A Privacy Preserving Authorisation System for the Cloud

David W Chadwick and Kaniz Fatema, University of Kent, Canterbury, CT2 7NF
d.w.chadwick@kent.ac.uk; kf66@kent.ac.uk

Abstract

In this paper we describe a policy based authorisation infrastructure that a cloud provider can run as an infrastructure service for its users. It will protect the privacy of users' data by allowing the users to set their own privacy policies, and then enforcing them so that no unauthorised access is allowed to their data. The infrastructure ensures that the users' privacy policies are stuck to their data, so that access will always be controlled by the policies even if the data is transferred between cloud providers or services. This infrastructure also ensures the enforcement of privacy policies which may be written in different policy languages by multiple authorities such as: legal, data subject, data issuer and data controller. A conflict resolution strategy is presented which resolves conflicts among the decisions returned by the different policy decision points (PDPs). The performance figures are presented which show that the system performs well and that each additional PDP only imposes a small overhead.

1. Introduction

Cloud computing has brought us a new era of Internet based data storage and processing power. The cloud offers enormous benefits to businesses such as reduced costs, since they no longer need to spend large amounts of capital on buying expensive application software or sophisticated hardware that they might never need. Instead they can perform their tasks using cloud services - either application, infrastructure or platform services (AaaS, IaaS and PaaS respectively) - and pay only for the resources they consume, when they need them. Cloud computing offers the opportunity to store a huge amount of data relatively cheaply. Using the cloud, users can access the services or applications regardless of their location or computing device/platform they use. However, despite all these benefits the cloud has to offer, privacy and security issues are still major challenges of cloud computing. There are many security issues to consider, including: fine grained access to cloud resources, privacy protection of data in the cloud, and auditing of cloud operations. Some threat models assume that the cloud provider cannot be trusted, and therefore propose storing only encrypted data in the cloud. Others assume that the cloud provider can be trusted, and that the threats come primarily from outside attackers and other cloud users. Given that most public cloud services are currently being run by large trusted organisations such as Amazon, IBM, Microsoft and Google, we believe that the latter threat model is reasonable for many users. Furthermore, organisations are now able to run their own private clouds, using open source software such as Eucalyptus (http://eucalyptus.com/) and hence the trusted provider model is the most appropriate one for this scenario. For example, the National Grid Service in the UK is already experimenting with private clouds for use by the academic community. Consequently the trusted

provider model is the one we adopt in this paper. Once we acknowledge this, we can start to design and build security and privacy protecting infrastructures that rely on a trusted cloud provider to operate parts of the infrastructure. We can then concentrate on the problems of designing and building a large scale federated privacy preserving infrastructure with fine grained access control and user accessible audit trails. If the technical infrastructure is strong enough and can provide users with the power to control access to their data, supported by appropriate legal contracts and audit trails, then cloud service providers should easily consolidate the trust of their users.

In this paper we describe an authorisation infrastructure that a cloud provider can run as a service for its users, that will allow the users to set their own privacy policies, thereby guaranteeing no unauthorised access to their data. The infrastructure ensures that these privacy policies are stuck to the users' data, so that access will always be controlled by them even if the data is transferred between cloud providers or services. This authorisation service accepts policy preferences from users e.g. whom they want to share their data with, or when they want their data to be deleted. These preferences are converted into policy rules and are stuck to the personal data within the service and when transmitting the data to another service. A receiving service will only store the data if it supports a policy decision point (PDP) that can evaluate the sticky policy language accompanying the data, otherwise it will refuse to accept it. Not only the user's preferences but also legal constraints will also be considered in order to assure privacy protection of the user's data. We have converted important sections of the EU data protection law into a policy which must be supported by each cloud service provider operating in the EU [30]. However, the structure of our system is generic so that it is possible to add any additional country specific legal constraints into the system, since we have separate policies for the law, data subject, data issuer and data controller. The system is capable of resolving conflicts between the decisions returned by the PDPs evaluating these different policies.

In cloud computing, one cloud provider may have to share data with other cloud providers, or release the data to authorized requestors. The provider will need to ensure that the receiving party is willing to enforce the same privacy policies for the data as itself, and will therefore need to send the sticky policies with the data. If the receiving party receives the data and sticky policies then it can only enforce these policies if it supports the same policy languages as the sender. Unfortunately there is no ubiquitously accepted standard for privacy policy languages. This is because different policy languages support different rule sets; hence it is not possible to construct every type of required policy using just one policy language. Therefore the same PDP cannot be used for evaluating all policies. Today we have many examples of different policy languages e.g. XACMLv2 [2], XACMLv3 [3], PERMIS [4], P3P [5], Keynote[6] etc. and hence many different PDP implementations. For example, XACMLv2 does not support delegation of authority whilst XACMLv3 and PERMIS do. The XACML policy language assumes a stateless PDP hence cannot support state based policy rules such as separation of duties (SoD), whilst PERMIS is state based and can support both dynamic and static SoD. Keynote on the other hand uses the same language to describe both credentials and policy rules whereas none of the other policy languages do. P3P is designed specifically to express privacy policies, whereas the others were designed as access control or authorization policy languages. Hence the cloud authorization service must support multiple PDPs and we have introduced a component (which we call the Master PDP) that can support multiple

subordinate PDPs and resolve any conflicts between their decisions.

Even though the policy handling authorisation infrastructure may be complex, we need to keep authorisation decision making as simple as possible for application developers when it is offered as an IaaS (Infrastructure as a Service). The complexity should be hidden behind a standard web services interface and orchestration of the different authorisation components should be done by the infrastructure itself. We propose this in our paper by using the OASIS SAML-XACML protocol [21] to provide this web service.

The rest of this paper is structured as follows. Section 2 reviews related research. Section 3 describes the architecture of our privacy preserving authorization infrastructure. Section 4 describes the sticky policy contents. Section 5 describes how conflict resolution is performed.. Section 6 describes the high level protocol exchanges for the receipt of data into the cloud, and the transfer of data between cloud providers. Section 7 gives details of our implementation whilst section 8 presents the validation and performance results.. Section 9 concludes and describes our future plans.

2. Related Research

Robert Gillman's report at the World Privacy Forum [1] focuses on privacy issues and legal compliance of sharing data in the cloud. He mentions various legal issues such as the possibility of the cloud being in more than one legal location at the same time with different legal consequences and such legal uncertainty makes it difficult to assess the privacy protection available to users. We cater for this by allowing different legal policies to be configured into our authorization service and by allowing legal policies to stop data being transferred to jurisdictions which do not have proper privacy protection.

Siani Pearson [10] has pointed out the key privacy requirements for clouds and a set of guidelines for designing the cloud service with privacy protection. Siani Pearson et al [11] have proposed to use accountability to enhance privacy protection in the cloud. They have identified the key elements for provisioning accountability within the cloud. Siani Pearson et al [12,13] have proposed a privacy manager which would obfuscate personal data in the client site before sending it to the cloud service provider. This approach would minimize the amount of sensitive data held within the cloud. Only the client will be able to obfuscate or de-obfuscate data with their chosen key. The problem with this approach is that the applications that are able to use the obfuscated data are limited and this will limit the services available to the user. Also the computation overhead of obfuscating and de-obfuscating the data is large and so it imposes constraints on the computing resources available to the user.

Dan et al [14] have proposed a data protection model by which a potential cloud user can find and choose a cloud service provider based on a user based ranking of the service providers. The user can then integrate their privacy policy with that of the service provider and any sub contractors and this combined policy can be coupled with the data, rather like our sticky policies. The work is still at a very preliminary stage and more design and implementation is needed.

Wassim Itani et al [15] have presented a security infrastructure for cloud providers to adopt. Privacy of the data in the cloud is ensured by the use of a secure cryptographic co-processor which provides a

trusted and isolated execution environment in the computing cloud. But the price of the co-processor may not make the solution feasible.

Anonymity based privacy preservation methods in the cloud have been proposed by Jian et al [16]. They have provided a simple example for anonymisation of some data based on the background knowledge of the service provider but their work lacks the automation of such anonymisation and is suitable for very limited services.

As can be seem most prior research assumes that the cloud provider cannot be trusted, whereas our approach is to assume that the cloud provider can be trusted to faithfully enforce the user's privacy policy. Therefore we believe that obfuscation or encryption of the user's data is not necessary for many cloud usage scenarios, especially private clouds.

3. Architecture of the Authorisation Service

The overall architecture of our authorization web service is shown in Figure 1 below. The AIPEP component offers an interface to cloud application developers, to allow them to easily call the service, passing it the user's sticky policy, obtaining authorization decisions, and being returned the sticky policies that should be attached to the data when it is transferred to another cloud service provider.

When the authorization service is offered as an IaaS service, then application or platform developers can build their application services to call the AIPEP service according to their need without worrying about implementing the complex authorization infrastructure which is offered as a service. For example, an enterprise wishing to provide privacy to their customer's personal data can build a user interface to their application, which will receive the privacy preferences of the user along with the user's private data, and can then submit this to their application service. The application service passes the user's policy and a request to store the user's data to the AIPEP service. If the data storage is allowed the AIPEP will ensure that the user's privacy policy is stored and ensure that it is consulted on all subsequent access requests for the data

Policy based systems typically rely on an application independent policy decision point (PDP) to make authorization decisions, and an application dependent policy enforcement point (PEP) to enforce these decisions. In our architecture the PEP is built into the application service, and it is expected to enforce the authorization decisions and obligations returned by the AIPEP service. We have introduced several new components into the cloud authorization service as described below.

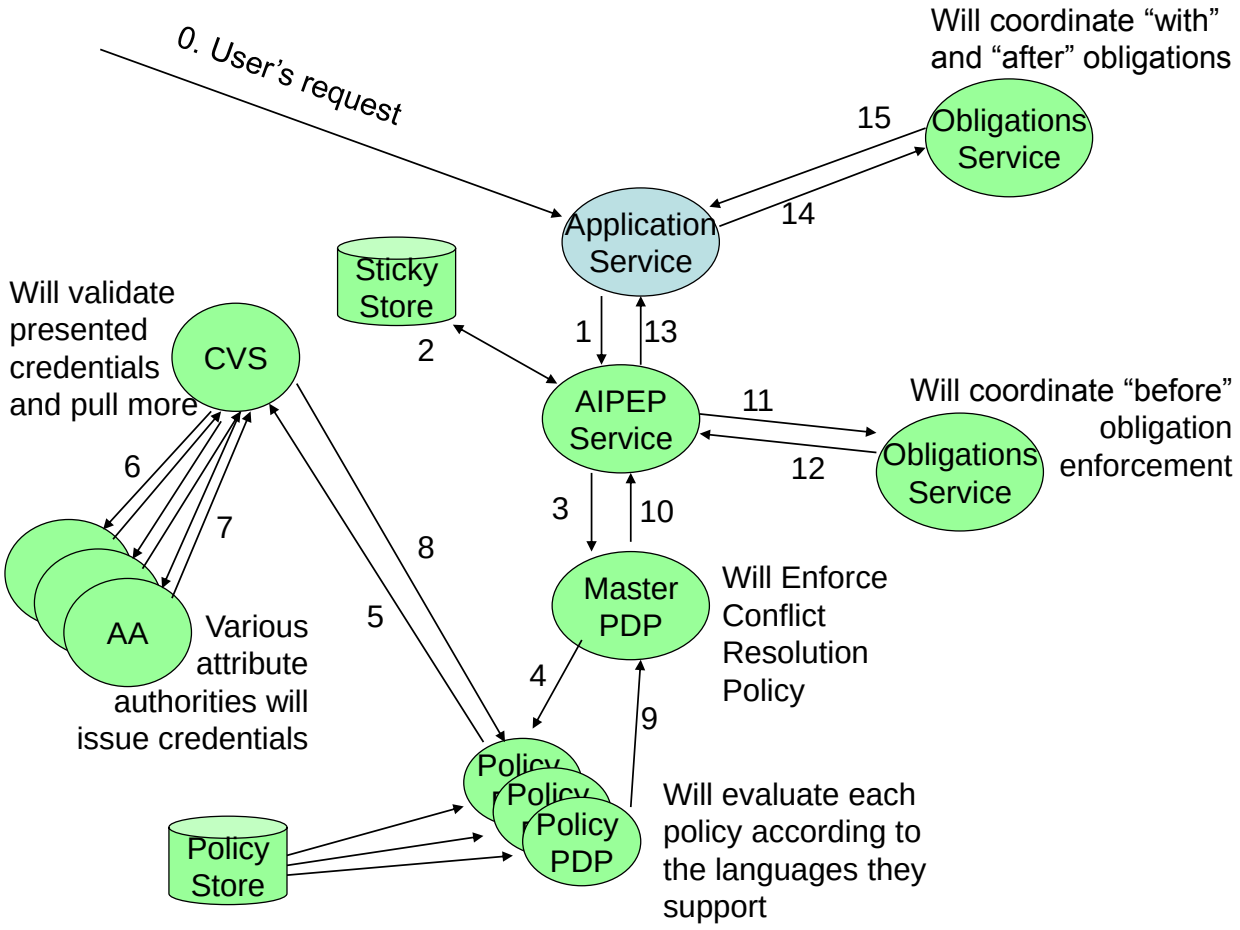

Figure 1. A Privacy Preserving Authorisation Service for the Cloud

3.1 AIPEP Service

Firstly we introduce the application **independent policy enforcement point service, the AIPEP service**. The AIPEP service is responsible for accepting web services requests from applications, coordinating the actions of the various components of the application independent authorization infrastructure, and then formulating web services decision responses to the applications. Each request should contain: a unique reference (RID) to the user's personal data, the sticky policy(ies) attached to this data, and details about the application request that was received in step 0. These latter details, called the request context in the XACML standard [2] typically comprise: the user's credentials, the type of application request being made by the user, expressed as the attributes of the action and the resource, and any supporting context parameters (such as time of day). Upon receiving this request, the AIPEP first stores the policy in the policy store and spawns a new PDP to handle it. It also binds the policy to the resource in the sticky store (step 2). The AIPEP retains a manifest which records which subordinate PDPs are currently spawned and which policies each is configured with. The AIPEP tells the Master PDP which set of spawned PDPs to use for a particular authorization decision request (step 3). The master PDP returns the overall decision (step 10) which may contain "before" obligations (see later). If so, the AIPEP calls the obligation service (step 11) to enforce these obligations, before returning the decision result to the

application (step 13). The application is then responsible for calling the obligations service for any "after" and "with" obligations to be enforced.

3.2 Credential Validation Service

In attribute based access controls (ABAC), the PDP makes it decisions based on the attributes of the subject, requested action, resource object and environment. In the XACML model all these attributes are provided by a Policy Information Point (PIP) and the subject's attributes are always assumed to be valid. In reality, subjects are issued with digitally signed credentials or claims by a number of different remote attribute authorities (AAs), some of which may be trustworthy, others which may not be. Whilst a PIP can usually reliably obtain the attributes of the resource object and the user's requested action, and in most cases those of the environment, reliably validating the credentials of the subject requires considerably more effort. This suggests there are different types of PIP. We thus need a type of PIP that is responsible for validating subject credentials, extracting the valid attributes from them and discarding the rest. We propose a credential validation service (CVS) for this [9].

The CVS is a specialized policy information point (PIP) that is configured with a credential validation policy. This policy tells it which credentials are valid, in terms of who the trusted authorities (IdPs) are and which attributes each are trusted to issue to which groups of cloud users. As pointed out in [29] delegation of authority is often an essential requirement in data-driven science projects where scientists often want to delegate access rights to a particular data set to an application/job running on their behalf. When delegation of authority is in operation, delegation chains of credentials may exist. The credential validation policy should therefore also contain a delegation policy that tells it which delegated credentials it can trust. The CVS should be able to work in either pull mode, push mode or pull and push mode. Pull mode means that the request did not contain any credentials and requires the CVS to pull them itself from its configured trusted attribute authorities, or a subset of them. In push mode the PEP pushes the credentials to the AIPEP, and in pull push mode some credentials are pushed and the remained have to be pulled. There currently isn't a standard policy language for a CVS policy, and in [9] we say why XACMLv2 is not sufficient for this (neither is XACMLv3). Consequently we have defined our own policy language for this.

Once the CVS has finished validating the subject's credentials, these are returned to the PDP as standard XACML formatted attributes (step 8).

3.3 Master PDP

In order to evaluate multiple authorization and privacy policies in different languages we introduce a new conceptual component called the Master PDP. The Master PDP is responsible for calling multiple PDPs (step 4) as directed by the AIPEP, obtaining their authorization decisions (step 9), and then resolving any conflicts between these decisions, before returning the overall authorization decision and any resulting obligations to the AIPEP (in step 10). Each of the policy PDPs supports the same interface, which is the XACML request-response context [2]. This allows the Master PDP to call any number of subordinate PDPs, each configured with its own policy in its own language. This design isolates this policy language from the rest of the authorization infrastructure, and the Master PDP will not be affected by any changes to any policy language as it evolves, or by the introduction of any new policy language.

3.4 Policy Store

PDPs need to obtain their policies from somewhere. The XACMLv2 standard proposes a functional component called the Policy Administration Point (PAP) which is responsible for creating the policies and making them available to the PDP through some back channel prior to the PDP making its decisions. The back channel could be, for example, an API to an integrated database, or a communications link to an external repository. However, using a back channel and previously prepared policies is too static and not sufficient for all use cases, especially for the privacy policies that originate from users, as these can't always be previously prepared. Consequently each user's data needs to be protected by the user's own policy and so the policy needs to be prepared by the user, stuck to the data and transferred via the front channel to the cloud at the same time as the user's data. Using this method for obtaining policies means that we can use the same method when transferring user data and sticky policies between cloud providers.

The policy store is the location where policies can be safely stored and retrieved. When the AIPEP stores a policy in the policy store, it is returned a unique reference to the policy, called the policy id (PID). The AIPEP can subsequently use this reference to retrieve the policy in order to spawn a new PDP.

3.5 Sticky Store

The sticky store holds the mapping between sticky policies (PIDs) and the resources (RIDs) to which they are stuck. This is a many to many mapping so that one policy can apply to many resources and one resource can have many sticky policies applied to it.

3.6 Obligations Service

Obligations are actions that must be performed when a certain event occurs. When the event is an authorization decision, then the obligations are actions that must accompany this decision. Enforcement of obligations, such as sending an email to a user when his/her data is accessed, or writing to an audit trail, or deleting data of a user after a certain time, is very important for a privacy preserving system. Some obligations need to be performed "before" the decision is enforced, some "after" the decision has been enforced, and some along "with" the enforcement of the authorization decision [7]. We call this the temporal type of the obligation. Examples are as follows: before the user is given access, increase the amount of logging to monitor what he is doing; after the user has been given access, record the amount of cpu that was used; simultaneously with the user's access, decrement his account balance. As described in [7] the failure semantics of each is as follows: a before obligation will be enacted even if the user's action subsequently fails, an after obligation may fail to be performed even if the user's action succeeded, and a with obligation should only succeed if the user's action succeeds, and should fail if the user's action fails. The presence of a "with" temporal type means that these obligations (at least) have to be atomic with the enforcement of the user's action, e.g. by supporting two-phase commit. Thus there are multiple places where obligations need to be enforced. Some obligations have been introduced to make up for deficiencies in the PDP's policy language, for example, [8] specifies how obligations can be used to specify over-ride policies in the XACML language, whilst [7] specifies how obligations can be used to support state based decision making in stateless PDPs such as XACML ones. Given that there are multiple places where obligations need to be enforced and that some have to be performed before the user's action is enforced, whilst others are extensions of the functionality of the

PDP, then it would be sensible to have these obligations enacted by a cloud infrastructure service, thereby reducing the burden on the application or platform developer. We propose an obligations cloud service with a standard interface that can be called from multiple places in an application work flow, with one of these places being in the application independent authorization infrastructure service.

According to the XACML model, each obligation has a unique ID (a URI). We follow this scheme in our infrastructure. Each obligations service is configured at construction time with the obligation IDs it can enforce and which obligation handling service is responsible for enacting each one. It is also configured with the temporal type(s) of the obligations it is to enforce. When passed a set of obligations by the AIPEP, or the application service, the obligations service will walk through this list, ignore any obligations of the wrong temporal type or unknown ID, and call the appropriate obligation handling service for the others. If any single obligation handling service returns an error, then the obligations service stops further processing and returns an error to the caller. If all obligations are processed successfully, a success result is returned. Each of the obligations enforced by the AIPEP must be of temporal type *before*.

4. Sticky Policy Contents

A sticky policy comprises:

- The policy author i.e. the authority which wrote the policy.
- The time of creation of the policy.
- The type(s) of resource(s) that are covered by this policy.
- The type of policy this is (see Figure 3).
- The policy language.
- The policy itself, written in the specified policy language.

Any number of sticky policies can be stuck to a data resource in either an application dependent manner e.g. as a <Condition> in a SAML attribute assertion, or by using the StickyPAD (**sticky p**olicy(ies) **a**nd **d**ata) XML structure that we have defined (see Appendix 1). Ideally the policies should be stuck to the data by using a digital signature, to ensure their strong binding. This could be by using the XML <ds:Signature> structure in the StickyPAD and SAML attribute assertion, or it could be externally provided e.g. by using SSL/TLS when transferring the data and policy across the Internet.
It is the responsibility of the sending application service to create the equivalent of the StickyPAD structure when sending data with a sticky policy attached to it, to another application service, and the receiving application to validate this signature when it receives the message in step 0 of figure 1. The application should then parse and unpack the contents and pass the sticky policy to the AIPEP along with the authorization decision request (step 1 of figure 1).

Various types of sticky policy may be defined:

- authorization policy – this says who is authorized to perform which actions on the associated data/resource. There may be several of these policies in a single StickyPAD (or its equivalent), with each policy being written by a different authority such as the data subject, the data issuer (e.g. IdP) and the legislative authority. Each authz policy has an author, so that it can be referred to by the conflict resolution policy (see later).

- conflict resolution policy – says how conflicts between the different authorization policy decisions are to be resolved. This policy may contain additional obligations that are to be returned along with any obligations returned by the subordinate PDPs. There may be several of these policies in a single StickyPAD, with each policy being written by a different authority. Which one takes precedence is determined by the authority hierarchy (see later).
- audit policy – says what information should be audited when the associated data/resource is accessed. There may be more than one audit policy written by different authorities, and the final audit policy used by the audit system will be the union of all individual audit policies contained in the StickyPAD.
- privacy policy – contains privacy specific rules such as retention periods and purposes of use. These may be combined in the authorization policy depending upon the specific authz policy language used, but not all authorization policy languages can support all privacy specific rules. There may be more than one privacy policy in a StickyPAD and the final privacy policy will be the intersection of all the individual privacy policies.
- authentication policy – says what level of authentication/assurance is required of requesting subjects who are to be allowed to access the associated data. Whilst it is possible to represent this policy in the authorization policy through the use of Level of Assurance, as for example as described in [28], by keeping this as a separate policy it allows the PEP to short circuit the whole authorization process if the requesting subject has not been authenticated sufficiently.
- data manipulation policy – provides rules for how the associated data (usually PII) can be transformed, enriched or aggregated with other personal data of the same data subject or of other data subjects.

5. Conflict Resolution Policy

Our system includes many different PDPs each with policies from different authorities and possibly written in different languages. As a consequence a mechanism is needed to combine the decisions returned by these PDPs and resolve any conflicts between them. The Master PDP is the component responsible for combining the decision results returned by multiple subordinate PDPs and resolving the conflicts among their decisions.

The Master PDP has a conflict resolution policy (CRP) consisting of multiple conflict resolution rules (CRRs). The default CRP is read in at program initialisation time and additional CRRs are dynamically obtained from the subjects' and issuers' sticky policies. Each conflict resolution rule (CRR) comprises:

- a condition, which is tested against the request context by the Master PDP, to see if the attached decision combining rule should be used,
- a decision combining rule (DCR),
- optionally an ordering of policy authors (to be used by FirstApplicable DCR)
- an author and
- a time of creation.

A DCR can take one of five values: FirstApplicable, DenyOverrides, GrantOverrides, SpecificOverrides or MajorityWins which applies to the decisions returned by the subordinate PDPs. The DCRs will be discussed shortly.

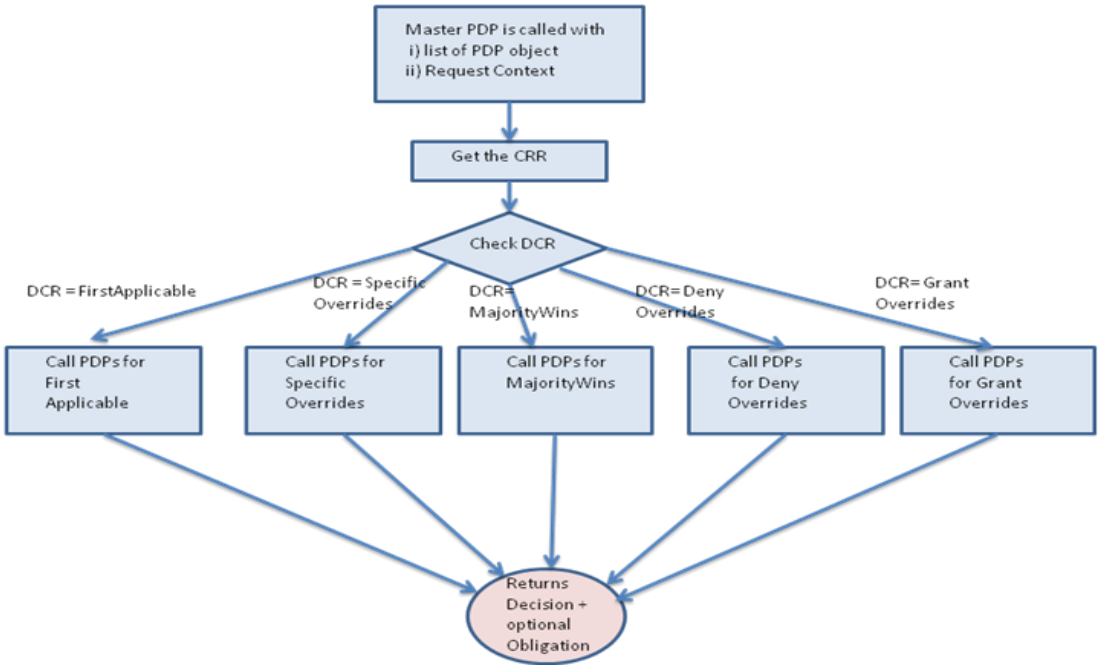

Figure. 2. Flow chart of the Master PDP

The Master PDP is called by the AIPEP and is passed the list of PDPs to call and the request context. From the request context it obtains the information such as requester, requested resource type, issuer and data subject of the requested resource. The Master PDP has all the CRRs defined by all the different authors as well as a default one. From the request context it knows the issuer and data subjects and so can determine the relevant CRRs. It will order the CRRs of law, issuer, data subject and data controller sequentially. For the same author the CRRs will be ordered according to their time of creation.

All the conditions of a CRR need to match with the request context for it to be applicable. The CRR from the ordered CRR queue are tested one by one against the request context. If the CRR conditions match the request context the CRR is chosen. If the CRR conditions do not match the request context the next CRR from the queue will be tested. The default CRR (which has DCR=DenyOverrides) is placed at the end of CRR queue and it will only be reached when no other CRR conditions match the request context. The PDPs are called according to the DCR of the chosen CRR.

Each PDP can return 5 different results: Grant, Deny, BTG, NotApplicable and Indeterminate. NotApplicable means that the PDP has no policy covering the authorisation request. Indeterminate means that the request context is either mal-formed e.g. a String value is found in place of an Integer, or is missing some vital information so that the PDP does not currently know the answer. BTG (Break the Glass) [17] means that the requestor is currently not allowed access but can break the glass to gain access to the resource if he so wishes. In this case his activity will be monitored and he will be made accountable for his actions. BTG provides a facility for over-riding access controls in an emergency.

If DCR=FirstApplicable the CRR is accompanied by a precedence rule (OrderOfAuthors) which says the order in which to call the PDPs. For example, if (resourceType=PII, requestor=data subject) DCR=FirstApplicable, OrderOfAuthor=law, dataSubject, holder. The Master PDP calls each subordinate PDP in order (according to the order of authors), and stops processing when the first Grant or Deny decision is obtained.

For SpecificOverrides all PDPs are interrogated and the decision returned by the PDP containing the rule with the most specific subject/ resource has priority over all the other PDPs. Note that in general rules will only be returned with Grant, Deny and BTG responses, and not with NotApplicable or Indeterminate, but if they are, they will be ignored in preference to the former. As the Master PDP does not know any of the rules, then each PDP needs to return the rule that caused its response together with its decision, in order for the Master PDP to determine which PDP has the most specific subject/resource rule. In cases of conflict, specific subjects are deemed to override specific resources. So the Master PDP will consult a subject ontology to determine which of the returned rules has the most specific subject. If multiple PDP rules have the most specific subject the Master PDP will choose the most specific resource among the rules having the most specific subject (the resource with the longest path name is deemed to be the most specific resource, for example C:/ MyDocument/MyFile is more specific than the C:/ MyDocument). If multiple PDP rules have the most specific subject and resource the decision of the PDP with the latest creation time will be chosen. If any of the PDPs does not return a rule with its decision then it is not possible to implement SpecificOverrides for this PDP as there is no way of determining whether it had the most specific subject or not. In this case we can either assume that the rule was the most generic and that this PDP comes last in the order of precedence, or we can replace the SpecificOverrides rule with a default rule. Deny Overrides is implemented as the default fallback strategy in this case.

For DenyOverrides and GrantOverrides the Master PDP calls all the subordinate PDPs and combines the decisions using the following semantics:

- DenyOverrides – A Deny result overrides all other results. The precedence of results for deny override is Deny>Indeterminate>BTG>Grant>NotApplicable.

- GrantOverrides – A Grant result overrides all other results. The precedence of results for grant override is Grant>BTG>Indeterminate>Deny>NotApplicable

When a final result returned by the Master PDP is Grant (or Deny) the obligations of all the PDPs returning a Grant (or Deny) result are merged to form the final obligation.

For MajorityWins all the PDPs are called and the final decision (Grant/Deny/BTG) will depend on the returned decision of the majority number of PDPs. If the same number of PDPs return Grant /Deny and BTG then Deny will be the final answer. If fewer Denies are returned and same number of Grant and BTG is returned then BTG will be the final answer. If Deny (or Grant) is the final answer then all the obligations of this returned decision will be merged. If none of the PDPs return Grant/Deny/BTG then Indeterminate will override NotApplicable.

The law and controller PDPs are common for all request contexts. Based on the request context the issuer and the data subject's PDP policies may be overriden.

6. Protocol exchanges

6.1 User input of PII and sticky privacy policy

The user is presented with an application dependent GUI and is asked to enter their PII. This requires existing GUIs to be enhanced to invite the subject to choose or enter their privacy policy. We do not specify how this is done. Organisations may have a limited number of options that a user can choose from e.g. a set of purposes, retention periods, audit requirements etc., or they may provide the user with the ability to enter their own fully specified privacy policy encoded in some standard policy language such as XACML or EPAL. Our infrastructure service is not constrained in the privacy policy languages that it can support, subject to the availability of an appropriate PDP that can evaluate authorisation decision requests and return an authorisation decision response via the AIPEP interface.

When the application service receives the user's input, it must extract the subject's privacy policy from the application layer message and pass this to the AIPEP in the authorisation decision request "can this user submit this PII (with this unique RID) to the data cloud store, using this sticky policy in conjunction with the existing policies". In this way the application does not need to understand the contents of the subject's privacy policy since the authorisation infrastructure will handle it in an application independent way. The AIPEP takes the policy, stores it in the policy store and is returned the PID of the policy. It then constructs a new PDP that can process this policy, passing it either the policy or the PID depending upon how the PDP is constructed. The AIPEP has a manifest that records the number of PDPs that are currently active, along with their PIDs. These include the PDPs that were initialised when the authorisation infrastructure service was started, plus any additional PDPs that have been dynamically

constructed since then. The size of the manifest is an implementation configurable parameter depending upon the capacity of the cloud. Once the PDP has been constructed the AIPEP passes the authorisation decision request to the Master PDP along with the set of PDPs that should be used to process this request.

The Master PDP consults its conflict resolution policy to determine how to resolve the authorisation decisions from the multiple PDPs. The Master PDP then calls the subordinate PDPs, either sequentially (first applicable rule) or in parallel (override rules) and analyses the returned decisions according to the rule. If the CRR is deny or grant overrides, and there is more than one such response of the same type, then the obligations from all such similar responses are combined together in the response that is returned to the AIPEP by the Master PDP. If a grant is returned this will contain at least one "before" obligation which instructs the authorisation system to store the subject's sticky policy in the sticky store.

The AIPEP calls the obligations service passing it the set of received obligations. This obligations service is only configured to process "before" type obligations, one of which will be instructions to the sticky store obligation service to store the subject's sticky policy. Other "before" type obligations may be configured into any of the policies of the subordinate PDPs, such as "audit the authz decision" etc. Only if all "before" type obligations are successfully enacted will the AIPEP return granted to the PEP. If any of the obligations fail to be enacted, then the AIPEP will return a deny response and will rollback its actions i.e. remove the subject's privacy policy from the policy store, terminate the appropriate subordinate PDP and remove it from its manifest.

When the application service receives the granted response it will store the user's PII in its cloud storage.

6.2 Client Access to a User's Data

When a client wishes to access a user's data, the application service intercepts the client's request and makes an authorisation decision query to the authorisation infrastructure service asking if this client has permission to read (or other access mode depending upon the client's request) the data identified by its unique RID. The AIPEP queries the sticky store to ask which sticky policies are bound to the resource with this RID. The sticky store returns the set of policy PIDs that are applicable.

The AIPEP consults its manifest to see which of these policies are currently loaded into PDPs, and if some are not, retrieves the appropriate policies from the policy store and initialises the corresponding subordinate PDPs. It then passes the authorisation decision request to the Master PDP along with the set of PDPs to query. The Master PDP consults its conflict resolution policy to determine which strategy to use (ordered or unordered) and passes the authorisation decision request to the subordinate PDPs either sequentially or in parallel. It then analyses the returned responses according to the governing conflict resolution rule and passes the appropriate response plus the combined set of obligations to the AIPEP.

The AIPEP enforces any "before" obligations that are present, and if these are successfully enacted it returns the response to the application service, otherwise a deny is returned. In this way the user's privacy policy is enforced along with any legal and organisational policies for accessing the data.

6.3 Transfer of data to another cloud provider

This protocol flow is very similar to client access to a user's data, but with the following differences. The authorisation decision request now becomes "has this third party cloud permission to retrieve the data identified by this unique RID". When the subordinate PDPs are asked if the third party is allowed to retrieve the PII, then along with each grant decision there may be a "before" obligation which instructs the authorisation system to retrieve the PDP's policy from the sticky store and return it to the application service, which is now required to put this sticky policy in the relevant application protocol field. On the third party's side the sticky policy has to be extracted from the relevant protocol field.

Once the third party cloud receives the response to its request, the protocol flow is now very similar to that described in section 6.1. The cloud application extracts the sticky policy(ies) and passes this/these to the authorisation infrastructure service along with the authorisation decision request "can this third party receive this data (with this unique RID) into its data store, using this policy(ies) in conjunction with the existing policies". Providing the local legal policy grants permission with a CRR of deny overrides (or is silent on this issue), then the user's policy will be enforced as directed by their sticky policy. The latter policy at least will require the third party to store and enforce the user's policy, and this will cause a before obligation to be returned to the AIPEP, which will ensure that the sticky policy is safely stored in the authorisation infrastructure before returning grant to the receiving application. If the authorisation infrastructure is not able to enforce the sticky policy obligation then the application will be denied permission to receive the user's data.

7. Implementation Details

The authorization infrastructure is implemented in Java, and is being used and developed a part of the EC TAS³ Integrated Project (www.tas3.eu). The first beta version is available for download from the PERMIS web site at http://sec.cs.kent.ac.uk/permis/downloads/Level3/standalone.shtml. This contains the AIPEP, CVS, the Obligations Service, a Master PDP and the ability to dynamically spawn new subordinate PDPs.

A number of different obligation services have been written that are called by the obligations service, and these can perform a variety of tasks such as write the authorization decision to a secure audit trail, send an email notification to a security officer, and update the internal state information (called retained ADI in ISO 10181-3 [18]. We have implemented state based Break The Glass policies [17] using the AIPEP, the obligations service and a stateless PDP. A live demo of BTG is available at http://issrg-testbed-2.cs.kent.ac.uk/.

The authorization infrastructure service has been tested with three different PDPs: Sun's XACML PDP [19], the PERMIS PDP and a behavioral trust PDP from TU-Eindhoven [20]. Each of these PDPs uses a different policy language. Sun's PDP uses the XACML language, the PERMIS PDP uses its own XML based language whilst TU-Eindhoven's PDP uses SWI-Prolog.

7.1 Protocol interfaces

We use standard web services protocols wherever possible. The AIPEP supports two different protocols, one for requesting an authorization decision and one for requesting credential validation.

The former uses the SAML-XACML protocol [21] as profiled by the OGF in [22]. This SAML profile allows an XACML formatted authorization decision request to be combined with the policy that is to be used by the PDP and both passed as a SAML query in a new element called an XACMLAuthzDecisionQuery. The SAML response allows an XACML response context, containing a response and optional obligations, to be returned in a newly defined SAML assertion, the XACMLAuthzDecisionStatementType. Other optional parameters can also be in the assertion, such as an altered request context which contains the set of attributes that were actually used in the authorization decision making. The OASIS XACML TC has recently agreed to enhance this protocol so that policies in any language can be transferred from the PEP to the PDP – the original version mandated that the policies had to be written in the XACML language. At the time of writing this has not been publicly released yet.

We have specified how various types of un-validated credentials such as SAML attribute assertions, X.509 public key certificates and X.509 attribute certificates can be passed to the AIPEP service in an Open Grid Forum profile [23] of [21].

The latter uses WS-Trust as profiled by the OGF in [24]. The protocol we use for communicating between the AIPEP and the CVS is based on WS-TRUST [25] and SAMLv2 [26] and the complete profile is specified as an Open Grid Forum profile [22]. We have enhanced this protocol to allow different credential validation policies to be dynamically transferred to the authorization infrastructure, but we have not yet had time to progress this through the standardization process. The WS-Trust protocol tells the CVS to work in either push or pull mode, or a combined push-pull mode. The CVS can dynamically retrieve (additional) user credentials from any number of external attribute authorities or repositories. We currently support two protocols for this: LDAP queries and SAMLv2 attribute queries. We support credentials formatted as SAML attribute assertions, LDAP attributes or X.509 attribute certificates. We have also specified an Open Grid Forum profile for the protocol to pull credentials [24], which is based on SAMLv2. Our CVS implementation however is more sophisticated than this, and can also pull X.509 attribute certificates from an LDAP repository or WebDav server [27], LDAP string attributes from a trusted LDAP server, and follow delegation chains of credentials. Push mode means that the request message contains the full set of credentials which are to be validated by the CVS, and none further need to be pulled. Note that the push/pull dialect field is advisory only, since the CVS is allowed to pull further credentials if it needs to. For example, if the AIPEP sent a delegated credential to be validated and the delegator is not a root of trust in the CVS's policy, then the credentials of the delegator will need to be pulled. Pull and push mode, as its name implies, is requesting the CVS to validate the credentials in the request message and pull any further credentials that it can find for the subject of the authorization decision query. Again an advice field can specify a subset of the AAs/IdPs to pull from.

The Master PDP currently talks to the subordinate PDPs using either the XACML request response context or the SAML-XACML protocol.

8. Validation and Performance Results

We validated the correct operation of the authorization infrastructure by running it on a small cloud server, details below. We first devised a test scenario, along with a series of tests , to validate that the correct authorization decisions were being returned by the authorization infrastructure, and then we carried out a series of performance tests to measure how quickly the correct authorization decisions could be made (along with storing and retrieving the user's sticky policies).

8.1 Validation Tests

In order to validate that the authorization infrastructure returns the correct authorization decisions we devised the following scenario:

> Patient M registers with the Local Health Centre and fills in a registration form with his personal details. He is also given a form where he enters his policy about who can access his medical records and personal details. This is mainly a tick box form so that the patient does not need to be worried about knowing any specific policy language. The patient's privacy preferences are automatically translated into a privacy policy which is stuck to his personal data and submitted into the system. Once registered, a doctor will perform a series of medical tests and enter these into the patient's electronic medical record which will also be stuck to the same privacy policy.

Suppose that initially M's policy has only one rule saying that researchers are allowed to view his medical data if the data can first be anonymized. The mandatory legal policy that is already encoded into the system contains 7 policy rules, summarized in Table 1 below (these translate into 13 rules in the legal PDP because the data subject can be identified in several different ways such as name and address or National Health Number). The Health Centre (data controller) also has its own policy in a separate PDP which states "Medical Professional s of this Health Centre are allowed to READ/ WRITE Medical data for any patient".

No.	Policy
1.	A data subject can send update requests for his/her personal data
2.	A data subject is rejected to access his/her personal data if LegalObjection=true
3.	A data subject is rejected to access his/her personal data if NationalSecurityIssue=true
4.	A data subject is rejected to access his/her personal data if the data is classified as a therapeutic exception or as Dr's personal notes.
5.	A data subject can view his/her personal data if the data is not classified as a therapeutic exception or as Dr's personal notes, LegalObjection=false and NationalSecurityIssue=false
6.	Legal Authority and National SecurityAuthority can read any personal data for the purpose of legal proceedings / prospective legal proceedings / obtaining legal advice/ establishing a legal claim / exercising a legal claim / defending a legal claim / administration of justice / prevention of real danger / suppression of a criminal offence / satisfying an important public interest.
7.	Medical professionals can BreakTheGlass to access to medical data for purpose of medical diagnosis/ the provision of care and treatment / preventive medicine / management of health care services.

Table 1. Configured Legal policies

We then postulated a set of 25 test cases for different people wishing to access M's medical records, including M himself, his doctor, a nurse, a relative, a researcher etc. and manually determined what the correct results should be. We input these requests into the authorization system, which was running 3 PDPs (the legal, data subject's and data controller's) and analyzed the responses to see if they concurred with the correct results. Once the correct results were consistently obtained from the authorization system, we then carried out a series of performance tests as described below.

8.2 Performance Tests

The authorization infrastructure was up and running on a small cloud server, whose configuration was: 2 CPUs each with 4 core and each core with hyper-threading support (equivalent to 16 core) with each core's speed being 2.53 GHz; cache size for each core is 12 MB; memory is 25 GB in total. The configuration for each Virtual Machine inside the cloud is: cpu MHz= 2527; cache=4096 kb; memory=~256MB. We ran the authorization infrastructure as one VM, although further performance improvements might be expected if the different components of the system, such as the obligations service and subordinate PDPs are run in their own VMs.

The configuration of the client machine, which performed the role of the cloud medical application running in a different cloud was: cpu MHz = 2993.589 MHz; cache=2048 KB; memory= 2052024 KB. Note that there was no underlying medical application, as all the client machine did was make authorization decisions requests across a LAN and receive authorization decisions. Because the client was operating across a local area network we realize that the results will be slower than if the client was running inside another VM of the same cloud service, but they will be faster than if the cloud application was being run by a different cloud provider accessing the storage service over the Internet.

Two series of tests were performed. The first set tested how long it took for the authorization system to store and retrieve a user's sticky policy and make an authorization decision. The second set tested the reduction in performance for an increasing number of PDPs running inside the authorization service.

For the second set of tests we had 1 to 10 PDPs, and each PDP had 1 rule in it. The PDPs were Sun's XACML PDP [19].

8.2.1 Making Authorisation Decisions

For the first set of tests, the legal PDP had 13 rules in it and the data controller's PDP had 1 rule in it. The user's sticky policy also had 1 rule in it and this policy was added in test 1 and transferred in test 3.

Test 1 was a request to store personal information along with a sticky policy. Three policies were evaluated (user's, legal and controller's) and the request was granted. The user's policy was stored by the authorization system. Test 2 was a request by a third part to read the user's personal information. The same three policies were interrogated and the decision was granted. The third test was a request to move the user's personal data to another cloud provider, the three policies were interrogated, the request was granted, and the user's sticky policy was returned with the decision.

Each test was run sequentially 500 times and then the mean time and standard deviation were calculated. Any results that varied over 3 times the standard deviation from the mean time were

removed as outliers. This resulted in up to 3% of the results being discarded. The results are presented in Table 2 below.

Test	Mean	Std Dev	% Discarded
1. Request to store personal data and sticky policy		2.0	1.9
2. Authz decision with 3 PDPs	7.41	0.82	1.2
3. Request to transfer data (and retrieve sticky policy)	11.38	0.9	3.0

Table 2. Time (in ms) to make an authorization decision and/or store/retrieve a sticky policy

From the results one can see that:

- the time to retrieve a sticky policy for transfer is approximately 4 ms.

- the time to store a sticky policy in a new PDP is approximately 6.4ms

A significant amount of the time is spent in processing the SAML-XACML request and response messages and transferring them across the LAN.

8.2.2 Increasing the number of PDPs

In this second series of tests the authorization server was configured with an increasing number of policies/PDPs, each containing 1 rule. In the first test the authorization server only had 1 policy configured into it (the legal PDP with 13 rules). In the second test the authorization server was configured with the legal policy and the data controller's policy. In the third test the authorization server had 3 policies: the user's sticky policy, and the legal and controller's configured policies. In the subsequent tests an additional sticky policy was added.. The number of rules in each additional PDP is kept the same for this set of tests so that the results can't be affected by various numbers of rules in different PDPs.

In each case we measured the time taken for an authorization decision to be made when a third party asked to read M's medical record, and a grant result was obtained. This necessitated all configured policies being interrogated and the Master PDP determining the overall decision, using a GrantOverrides combining rule. The results are shown in Table 3 below.

Test	Mean	Std Dev	% Discarded	Mean PDPi – MeanPDPi-1
1 PDP	4.11	0.31	5.4	
2 PDPs	5.40	0.73	3.8	1.29

3 PDPs	7.19	1.35	5.8	1.79
4 PDPs	10.17	0.49	5.6	2.98
5 PDPs	13.06	0.95	0.8	2.89
6 PDPs	15.63	1.11	1.0	2.57
7 PDPs	18.57	1.08	2.12	2.94
8 PDPs	21.34	1.2	1.9	2.77
9 PDPs	23.93	1.4	0.79	2.59
10 PDPs	25.54	1.5	1.0	1.61

Table 3. Time (in ms) to make an authorization decision for different number of PDPs

9. Discussion, Conclusions and Future Plans

We have built a sophisticated privacy protecting authorization infrastructure that provides a web service interface for cloud providers to use. It can be used by an IaaS provider as part of the infrastructure offering to platform and application developers. The latter may then use this to further develop privacy preserving applications. Our authorization infrastructure does not obviate the need for trust, but rather is built on the assumption that cloud providers can be trusted to the extent that they wish to provide an automated infrastructure that can easily enforce each other's policies reliably and automatically. If they cannot support an incoming sticky policy they will inform the sender of the fact. Our system provides IaaS providers with an application independent authorisation infrastructure that makes it easy for them to enforce users' privacy policies without having to write a significant amount of new code themselves. Furthermore the user has the potential for more complete control over his/her privacy than now, in that the infrastructure allows the user to specify a complete privacy policy including a set of obligations which can notify the user when his/her data is accessed or transferred between organizations e.g by using an *after* obligation when giving permission for the transfer of her data to go ahead or a *before* obligation before giving permission for the data to be read. Of course, an untrustworthy cloud provider can always discard any sticky policies it receives and never need access the authorisation infrastructure to ask for permission to receive the data, but we assume that legally binding contracts between the cloud providers and their users will require them to support the sticky policies that are transferred to them. Our authorisation infrastructure makes it much easier for them to do this.

Future work will look at how the authorization infrastructure can be partitioned into separate services running in different VMs, so as to increase its performance, as well as looking at how scalability can be achieved through horizontal elasticity.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh

Framework Programme (FP7/2007-2013) under grant agreement n° 216287 (TAS³ - Trusted Architecture for Securely SharedServices)[1].

References:

[1] Robert Gellman, "WPF REPORT: Privacy in the Clouds: Risks to Privacy and Confidentiality from Cloud Computing", February 23, 2009.

[2] OASIS "eXtensible Access Control Markup Language (XACML) Version 2.0"OASIS Standard, 1 Feb 2005

[3] OASIS. "eXtensible Access Control Markup Language (XACML) Version 3.0". Committee Specification 1. 10 August 2010

[4] David Chadwick, GansenZhao, Sassa Otenko, Romain Laborde, Linying Su and Tuan Anh Nguyen. "PERMIS: a modular authorization infrastructure". Concurrency And Computation: Practice And Experience. Volume 20, Issue 11, Pages 1341-1357, 10 August 2008.

[5] W3C: The Platform for Privacy Preferences 1.0 (P3P 1.0). Technical Report. 2002

[6] Blaze, M., Feigenbaum, J., Ioannidis, J. "The KeyNote Trust-Management System Version 2", RFC 2704, September 1999.

[7] David W Chadwick, Linying Su, Romain Laborde. "Coordinating Access Control in Grid Services". Concurrency and Computation: Practice and Experience, Volume 20, Issue 9, Pages 1071-1094, 25 June 2008.

[8] Alqatawna, J.; Rissanen, E.; Sadighi, B. "Overriding of Access Control in XACML". Proc. 8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '07) 13-15 June 2007. Page(s):87 – 95

[9] David W Chadwick, Sassa Otenko and Tuan Anh Nguyen. "Adding Support to XACML for Multi-Domain User to User Dynamic Delegation of Authority". International Journal of Information Security. Volume 8, Number 2 / April, 2009 pp 137-152

[10] Siani Pearson "Taking Account of Privacy when Designing Cloud Computing Services", in Proceedings of ICSE- Cloud '09, Vancouver, 2009.

[11] Siani Pearson, Andrew Charlesworth, "Accountability as a Way Forward for Privacy Protection in the Cloud", Proceedings of the 1st International Conference on Cloud Computing,Beijing, China, 2009.

[12] Siani Pearson, Yun Shen and Miranda Mowbray, "A Privacy Manager for Cloud Computing", HP Laboratories , HPL-2009-156

[13] Miranda Mowbray, Siani Pearson. A client-based privacy manager for cloud computing, In COMSWARE '09. ACM, 2009

[14] Dan Lin, Anna Squicciarini, "Data Protection Models for Service Provisioning in the Cloud", SACMAT '10, Pittsburgh, Pennsylvania, USA, 2010.

[15] Wassim Itani, Ayman Kayssi, Ali Chehab, "Privacy As A Service: Privacy-Aware Data Storage and Processing in Cloud Computing Architectures", Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing DASC '09, 2009.

[16] Jian Wang, Yan Zhao, Shuo Jiang, Jiajin Le, "Providing Privacy Preserving in Cloud Computing", International Conference on Test and Measurement, ICTM '09 , 2009.

[1] The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.

[17] Ana Ferreira, David Chadwick, Pedro Farinha, Ricardo Correia., Gansen Zhao, Rui Chilro, Luis Antunes. "How to securely break into RBAC: the BTG-RBAC model", Annual Computer Security Applications Conference, Honolulu, Hawaii, December 2009

[18] [ITU-T Rec X.812 (1995) | ISO/IEC 10181-3:1996 "Security Frameworks for open systems: Access control framework"

[19] Sun's XACML PDP, available from http://sunxacml.sourceforge.net/

[20] Böhm, K., Etalle, S., Hartog, J.I. den, Hütter, C., Trabelsi, S., Trivellato, D. & Zannone, N. "A flexible architecture for privacy-aware trust management". Journal of Theoretical and Applied Electronic Commerce Research, 2010, vol 5(2), 77-96"

[21] OASIS "SAML 2.0 profile of XACML, Version 2.0". OASIS committee specification 01, 10 August 2010

[22] David Chadwick, Linying Su. "Use of WS-TRUST and SAML to access a Credential Validation Service". OGF GWD-R-P, 25 June 2009.

[23] David W Chadwick, Linying Su, Romain Laborde. "Use of XACML Request Context to access a PDP". OGF GWD-R-P. 25 June 2009

[24] V. Venturi, T. Scavo, D.W. Chadwick, "Use of SAML to retrieve Authorization Credentials", OGF GWD-R-P, 25 June 2009

[25] OASIS, "WS-Trust 1.3", OASIS Standard, 19 March 2007

[26] OASIS. "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard, 15 March 2005

[27] David W Chadwick, Sean Anthony. "Using WebDAV for Improved Certificate Revocation and Publication". In LCNS 4582, "Public Key Infrastructure. Proc of 4th European PKI Workshop, June, 2007, Palma de Mallorca, Spain. pp 265-279

[28]

[29] Palankar, M. R., Iamnitchi, A., Ripeanu, M., and Garfinkel, S. "Amazon S3 for science grids: a viable solution?." In Proc. 2008 Int. W'shop on Data-Aware Distributed Computing (Boston, MA, USA, June, 2008). DADC '08. ACM, New York, NY, pp55-64

[30] Kaniz Fatema, David W Chadwick, Brendan Van Alsenoy. "Legal Policies to Protect Privacy". Summer School. To be published in LNCS

Appendix 1. The StickyPAD – an XML schema for carrying data and sticky policies together

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="tas3:to:be:decided:namespace"
    targetNamespace="tas3:to:be:decided:namespace"
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <!--
<xs:import namespace="urn:oasis:names:tc:SAML:2.0:assertion"
schemaLocation="http://docs.oasis-open.org/security/saml/v2.0/saml-schema-
assertion-2.0.xsd"/>
-->
    <!-- Use a schema on local file store as there seem to be problems with
the
ones available on the net. -->
<xs:import namespace="http://www.w3.org/2000/09/xmldsig#"
schemaLocation="http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/xmldsig-
core-schema.xsd"/>
```

```
<xs:element name="StickyPad" type="StickyPADType"/>
<xs:complexType name="StickyPADType">
    <xs:annotation>
        <xs:documentation>
     This is the TAS3 Sticky Policy and Data/resource type definition.
     Version 8. 2 December 2010.
     The DataResource can be any data or resource which requires a sticky
policy.
     Resource Types holds the type(s) of resource that are contained
     in the DataResource e.g. it could be a computer system or an email
message
     or some PII.
     Any number of policies can be stuck to a DataResource.
     The XML signature is optional because applications may choose to
     secure the PAD using alternate means, e.g. SSL/TLS.
        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:choice>
          <xs:element ref="DataResource"/>
          <xs:element ref="DataResourceRef"/>
        </xs:choice>
        <xs:element name="DataResourceTypes" type="ResourceTypes"
form="qualified"/>
        <xs:element ref="StickyPolicy" maxOccurs="unbounded"/>
        <xs:element ref="ds:Signature" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>


<xs:complexType name="ResourceTypes">
   <xs:sequence>
      <xs:element name="ResourceType" type="xs:anyURI" maxOccurs="unbounded"
form="qualified"/>
   </xs:sequence>
</xs:complexType>


<xs:element name="StickyPolicy" type="StickyPolicyType"/>
<xs:complexType name="StickyPolicyType">
    <xs:annotation>
        <xs:documentation>
     The Policy ID specifies the globally unique ID of the policy.
     The PolicyLanguage specifies the language the policy is written in.
     The PolicyAuthor specifies attributes of the person who wrote the
policy.
     Time of Creation specifies when the policy was written.
     Expiry time specifies after what time the policy should be ignored.
Infinity is the default.
     Resource Type specifies the type(s) of resource this policy refers to.
     The PolicyContents contains the policy written in the language
           specified in PolicyLanguage.
     The PolicyType specifies what type of policy this is.
        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
```

```
        <xs:element name="PolicyAuthor" type="PolicyAuthorType"
form="qualified"/>
        <xs:element name="PolicyResourceTypes" type="ResourceTypes"
form="qualified"/>
        <xs:element ref="PolicyContents"/>
    </xs:sequence>
    <xs:attribute name="PolicyID" type="xs:anyURI" use="required"/>
    <xs:attribute name="PolicyLanguage" type="xs:anyURI" use="required"/>
    <xs:attribute name="PolicyType" type="xs:anyURI" use="required"/>
    <xs:attribute name="TimeOfCreation" type="xs:dateTime" use="required"/>
    <xs:attribute name="ExpiryTime" type="xs:dateTime" use="optional"/>
</xs:complexType>

<xs:element name="PolicyContents" type="AnyXMLType"/>
<xs:element name="DataResource" type="AnyXMLType"/>
<xs:element name="DataResourceRef" type="xs:anyURI"/>

<xs:complexType name="AnyXMLType" mixed="true">
    <xs:sequence>
        <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any"
processContents="lax">
            <xs:annotation>
                <xs:documentation>
                    Any xml content is allowed in this element.
                </xs:documentation>
            </xs:annotation>
        </xs:any>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="PolicyAuthorType">
      <xs:annotation>
            <xs:documentation>
             The Author is identified by any number of Author Attributes.
             AuthorType indicates the author's relationship to the Resource
in this StickyPAD
            </xs:documentation>
      </xs:annotation>
       <xs:sequence>
          <xs:element name="AuthorAttribute" type="AuthorAttributeType"
minOccurs="0" maxOccurs="unbounded" form="qualified"/>
          <xs:element name="AuthorType" type="xs:anyURI" form="qualified"/>
       </xs:sequence>
</xs:complexType>

<xs:complexType name="AuthorAttributeType">
        <xs:attribute name="AttributeId" type="xs:anyURI" use="required"/>
        <xs:attribute name="Issuer" type="xs:anyURI" use="optional"/>
        <xs:attribute name="IssueInstant" type="xs:dateTime" use="optional"/>
        <xs:attribute name="Value" type="xs:string" use="required"/>
</xs:complexType>

</xs:schema>
```