Deploy, Adjust and Readjust: Supporting Dynamic Reconfiguration of Policy Enforcement

Gabriela Gheorghe, Bruno Crispo, Roberto Carbone, Lieven Desmet, Wouter Joosen

Deploy, Adjust and Readjust: Supporting Dynamic Reconfiguration of Policy Enforcement

Gabriela Gheorghe[1], Bruno Crispo[1], Roberto Carbone[2],

Lieven Desmet[3], and Wouter Joosen[3]

[1] DISI, Università degli Studi di Trento, Italy, First.Last@disi.unitn.it
[2] Security and Trust Unit, FBK, Trento, Italy, carbone@fbk.eu
[3] IBBT-Distrinet, K.U.Leuven, 3001 Leuven, Belgium, First.Last@cs.kuleuven.be

Abstract. For large distributed applications, security and performance are two requirements often difficult to satisfy together. Addressing them separately leads more often to fast systems with security holes, rather than secure systems with poor performance. For instance, caching data needed for security decisions can lead to security violations when the data changes faster than the cache can refresh it. Retrieving such fresh data without caching it impacts performance. In this paper, we analyze a subproblem: how to dynamically configure a distributed authorization system when both security and performance requirements change. We examine data caching, retrieval and correlation, and propose a runtime management tool that, with external input, finds and enacts the customizations that satisfy both security and performance needs. Preliminary results show it takes around two seconds to find customization solutions in a setting with over one thousand authorization components.

Key words: configuration, policy, enforcement, middleware, cache

1 Introduction

Systems like Facebook, with hundreds of millions of users that add 100 million new photos every day, have data centres of at least thousands of servers [17, 20]. Similarly, eBay uses over 10000 Java application servers [23]. Such large applications need an infrastructure that can adapt to its usage constraints, since performance is a must for business. Performance usually refers to the number of user requests (e.g., clicked links on a web page) serviced per time unit. For Facebook, for instance, it has been stated that if website latency is reduced by 600ms, the user click-rate improves by more than 5%[24]. To achieve performance, Twitter, Google and eBay are using caching (page caching for user pages, local caching for scripts), replication, and data partitioning.

Application providers want security and privacy requirements to be satisfied. Application policies focus on user authentication and authorization, and data privacy. The problem of satisfying all these needs, i.e., *policy enforcement*, becomes complex in a distributed system with multiple constraints and multiple

users. When user data, profiles and action histories are spread across several domains, it is difficult to make this data available safely and consistently across the system. Maintaining this data is essential for correct security decisions, but is complicated by different domains introducing specific data access patterns.

Caching can impair policy enforcement. Commonly, user certificates, authorization decisions or user sessions are cached so that they are quickly retrieved. But when this data changes faster than it is cached, the decision of authenticating or authorizing may no longer be correct, in which case revenue or human lives can be at stake. For instance, on eBay when a buyer wants to bid on an item that is advertised by a seller, there can be a policy on the seller's side that would reject receiving bids when the buyer has a history of bad payment. If at one moment in time the customer's history is clean, by the time the bid request gets to the seller's side it will be accepted since it relies on a cached value of the payment history that is clean; this decision might not be true, in fact, if there is one bad payment reported before the cache is refreshed.

Attribute retrieval can also affect policy enforcement. Attributes refer to all data needed for authorization decisions (e.g. user profile, history, system state, user state). Retrieval can be done directly or using a mediator, in which case it runs the risk of staleness. Also, if attributes are semantically related, their retrieval should be the same way (we call it *correlation*). Attributes are not always retrieved directly, for two reasons: (1) it is costly to retrieve them every time, and (2) the data is private. For instance, eBay has multiple security domains (i.e., groups of providers with similar policies); to authorize a buyer operation in domain A, the eBay system needs to check the buyer history of bad payments with any seller for the current month, and buyer history is maintained in domain B. If domain B gives the freshest buyer history to domain A just for computing the bad payment events, all non-payment data would be disclosed.

When security enforcement and performance collide, such vulnerabilities occur in different systems tuned for performance. The Common Vulnerabilities and Exposures database [21] reveals numerous entries related to cache and configuration management. For instance, OpenSSL suffers from race conditions when caching is enabled or when access rights for cache modification are wrong (CVE-2010-4180, CVE-2010-3864, CVE-2008-7270). Similar problems of access restrictions to cached content are with IBM's DB2 or IBM's FileNet Content Manager (CVE-2010-3475, CVE-2009-1953), as well as ISC's BIND (CVE-2010-0290, CVE-2010-0218). Exploiting such issues is reported to lead to buffer overflows, downgrade to unwanted ciphers, cache poisoning, data leakages, or even bypassing authentication or authorization. Protocol implementations like OpenSSL and BIND, or servers like IBM's and RSA Access Manager, are examples of technologies that need to scale fast, but cannot scale *fast and safely*.

We argue that the techniques to improve a distributed application's performance must be tailored to the application's security needs. In our view, caching, retrieval and correlation of enforcement attributes require a management layer that intersects both the application and its deployment environment. To our knowledge, these aspects have not yet been approached in security enforcement.

To bridge this gap, we suggest a method and framework for adjusting at runtime the security subsystem *configurations*, i.e., the ways to connect components and tune connection parameters. This adjustment requires to find ways to connect security components so that their connections satisfy a set of constraints, specified by domain experts or administrators (e.g., eBay security architects or security domain administrators) and included as annotations in the XACML security policy. The system configuration that allows for both security and performance needs is *dynamic*. Constraints can change because tolerance to performance overheads or inaccurate enforcement decisions can vary; security domains can vary in dimension; network topology can change. Since varying runtime constraints imply re-evaluation rounds, we want to automate the reconfiguration of the authorization subsystem. Thus, our contributions are:

1. We show the need to consider security and performance restrictions together, rather than separately. We focus on attribute retrieval, caching, and correlation. To our knowledge, we are the first to look at caching from the perspective of the impact of stale attributes over the authorization verdict in the policy enforcement process.
2. We present a method to dynamically compute the correct configurations of policy enforcement services, by transforming system constraints into a logic formula solved with a constraint solver.
3. We suggest the first middleware tool to perform adaptive system reconfiguration on security constraints. Having split computation in two phases, preliminary results show that the heavier computation phase takes 2.5 seconds to compute a solution for over 1000 authorization components.

The paper is structured as follows. After an overview of the standard enforcement model (Section 2.1) and an illustrative example (Section 2.2), Section 3 overviews properties of enforcement attributes. Section 4 describes our approach and proposed architecture, while Sections 5 and 6 describe our prototype in more details. Related work is presented in Section 7 and Section 8 concludes.

2 Background and example

To be able to evaluate the configuration of the authorization system and how it can be adjusted, this section first describes the reference model in policy enforcement, and then presents an example that we find illustrative for our case.

2.1 The Reference Enforcement Model

The eXtensible Access Control Markup Language (XACML)[4] is the de facto reference in authorization languages and architectures. XACML has been widely adopted in industry, examples ranging from IBM's Tivoli Security Manager [13], to Oracle, JBoss and Axiomatics [3]. XACML has been an OASIS standard

[4] `http://www.oasis-open.org/committees/xacml/`

since 2003, and apart from the XML-based language specification, it proposes an authorization processing model based on the terminology of IETF's RFC3198[5]. Fig. 1 shows the three main components (greyed out) that we consider:

Policy Decision Point (PDP) is the entity that makes the authorization decision, based on the evaluation of the applicable policies;
Policy Enforcement Point (PEP) performs two functions: makes decision requests, and enforces (implements) the authorization decision of the PDP;
Policy Information Point (PIP) is the entity that acts like a source of attribute values to the PDP that makes the decisions.

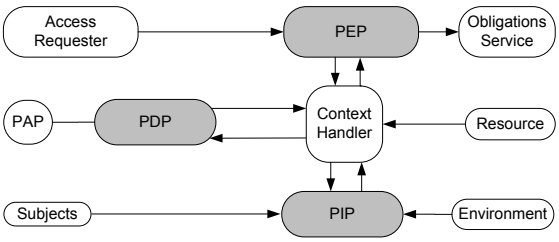


Fig. 1. The XACML reference architecture.

There are two variations of the XACML model, shown in Fig.3: the *attribute push model*, where the PEP collects the attributes offered by the PIPs and sends them to the PDP, and the *attribute pull model*, when the PDP collects all relevant attributes in order to make a decision. The pull model has a minimal load on the client and PEP and is designed for cases when the authorization process is entirely performed on the PDP; the push model, conversely, allows for more flexibility to the client and the PEP needs to ensure all required data reaches the PDP. In practice (e.g., PERMIS [5]), a *hybrid model* is used, whereby some attributes are pushed and some are pulled. To our knowledge, no distinction has been made as to when to pull and when to push an attribute.

There have been efforts to augment the XACML reference architecture: Higgins[6] proposes a new authorization architecture for managing online identity. A similar identity management variation was proposed by Djordjevic et al. [7], suggesting a common governance layer that incorporates the PEP but also manages interactions with an identity broker. Overall, such meta-models still keep the original XACML architecture. Thus, using the XACML reference as our model suffices to support a claim of general applicability.

2.2 Illustrative example

A *security subsystem* is the ensemble of components that perform authentication or authorization enforcement – a set of PIPs, PEPs, and PDPs. We follow

[5] `http://tools.ietf.org/html/rfc3198`
[6] `http://www.eclipse.org/higgins/`

some architectural points from eBay [16], whereby the security subsystem is divided into security domains (i.e., groups of components under similar security policies). User credentials, profile and state are spread across different domains; access patterns to such data can vary, and domains need to authenticate to other domains when user data is retrieved. We target policies whose evaluation require

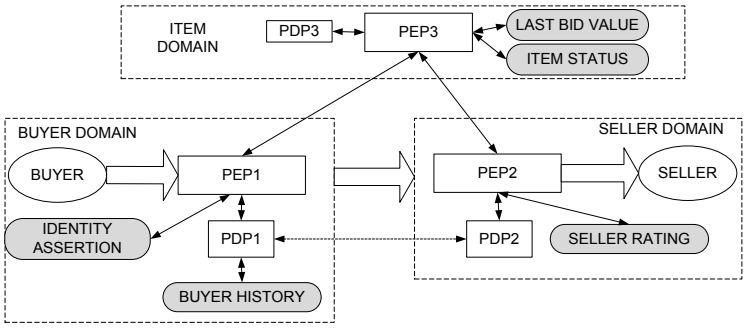


Fig. 2. Cross-domain authorization for a Buyer in BuyerDomain to bid in SellerDomain, with item information from ItemDomain. The greyed boxes are the bid authorization attributes: identity assertion, history, seller rating, item status and bid value.

scattered security data, e.g. the policy (**P1**)"*an authenticated buyer is allowed to make a bid on available items from a seller only if the buyer's history shows no delayed payment for the last month*". The buyer history for the current month, seller's available items on sale, identity tokens, and sellers' ratings – this data is kept at various locations where it may change at runtime, with an impact over allowing or disallowing further user actions (e.g., bids). If buyer attributes are not updated correctly or if fresh values are not retrieved in time, buyers will be blocked from certain sellers, or buyers will buy items that they should not normally access. Fig. 2 shows a multi-domain system where in order for a buyer in domain Buyer Domain to be allowed to bid to a Seller in the Seller Domain, its request must pass through the security subsystem of Buyer Domain, gather some data from Item Domain, and pass through the security subsystem of Seller Domain, which in its turn can also require item data from Item Domain.

For us, the problem of attribute management as shown above can be solved by finding a tradeoff between security correctness and performance requirements. This observation is confirmed by Randy Shoup, distinguished architect at eBay, who admits that an efficient caching system aims to "maximise cache hit ratio within storage constraints, requirements for availability, and tolerance for staleness. It turns out that this balance can be surprisingly difficult to strike. Once struck, our experience has shown that it is also quite likely to change over time" [22]. In this paper, we propose a framework to compute and recompute such balance at runtime and adapt the security subsystem accordingly: we do not want to change the entities of the security subsystem, but the connections between them that involve retrieval and caching of security attributes.

3 Attribute Configuration

Attributes are application-level assets, and have value in the enforcement process (that is aware of application semantics). In eBay, buyer history, item status and seller rating are attributes; in Shibboleth [14], the LDAP class *eduPerson* defines name, surname, affiliation, principal name, targetId as usual identity attributes.

How to obtain attributes is specified by *configuration (meta)data*. For instance in Shibboleth, identity and service providers publish information about themselves: descriptors of provision, authentication authority and attribute authority, etc. Such metadata influences the enforcement process, and can cover:

- visibility (e.g., is the assertion server reachable? is the last bid value public?),
- location or origin restrictions (e.g., for EU buyers use EU server data),
- access pattern of security subsystem to contact third-party (e.g., if the item data can be encrypted, can be backed up or logged),
- connection parameters (e.g., how often should buyer history data be cached or refreshed? who should retrieve it?).

Our point is that this configuration level, including attribute metadata, complements the enforcement process and can influence it. We examine three aspects: (1) push/pull models for attribute retrieval, (2) caching of attributes, and (3) attributes to be handled in the same way (coined 'correlation').

3.1 Attribute Retrieval

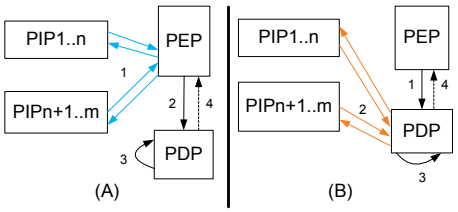
In our eBay example in Fig. 2, *attribute push* from PEP1 to PDP2 means that some attributes – 'identity assertion' , 'buyer history' and item information from Item Domain – are pushed to the Seller Domain and then to PDP2, which will make the final access decision. The 'buyer history', 'last bid value' and 'item status' can also be *pulled* by PDP2 at the moment when it needs such data.

From the point of view of performance, both attribute push and attribute pull can be problematic. In the attribute pull case, an excessive load on PDP2 can make it less responsive for other buyer requests. Performance is linked with security: low PDP2 performance can be made into a Denial of Service attack by saturating PDP2 with requests that require intensive background work. The case of all attributes being retrieved by PEP1 and given to PDP2 is also delicate: for example, if PEP1 knows that PDP2 requires the number of delayed payments of the buyer, it means that PEP1 will do the computation instead of PDP2; this scheme can put too much load on PEP1, that can have a negative effect on PEP1's fast interception of further events.

From the point of view of trust and intrusiveness, the push scheme is problematic. In Fig. 2, PDP2 must trust what PEP1 computed as the highest customer rating this month, and this potentially sensitive data would need to travel between PEP1 and PDP2. This decision bears several risks, of which: PEP1 might compute the wrong rating value; the decision might be delayed; if the rule changes from "any delayed payments", to "under five", then PEP1's logic should

be updated. The security subsystem should decide if exporting the computation of the delayed payment count to PEP1 costs more than the privacy of such local data to Buyer Domain. In either case, PEP1 and the communication channel should be trusted not to tamper with the data, and the policy should be fixed.

To determine when to use the push or the pull scheme, we have analysed a number of possible policy enforcement configurations and several existing scenarios and their solutions – PERMIS [4, 5], Shibboleth [14], and Axiomatics [3]. In particular, PERMIS looks at role-based authorization in multiple Grid domains. There are two kinds of regulations on subject credentials in PERMIS: policies on what credentials can be attached to the authorization request leaving from Buyer Domain (policies enforced by PDP1 in Fig. 2) and can reach Seller Domain; and policies on what credentials can be trusted by Seller Domain as the destination of an authorization request from Buyer Domain. Validating a buyer credential in Seller Domain – done by PDP2 – involves a chain of trust among issuers that have different rights to issue attributes for different sets of subjects. On a similar note, Shibboleth [14] enforces attribute release policies whereby once a user has successfully authenticated to a remote website, certain user attributes can be provided on demand via back-channels by requesting Web sites while other attributes cannot. These examples confirm that user or system attributes are sensitive, and should not always be pushed to the PDP.



Push scheme	Pull scheme
stateless attributes	state attributes
environment attributes	behaviour history
not sensitive user data	sensitive user data
request details	obligations
destination details	domain decisions
stateless context	stateful context

Fig. 3. Diagrams A and B show the classic attribute push and pull models. The table to the right shows how different attributes fit the push or pull scheme better.

Based on the security considerations above, we separate the applicability of the push and pull schemes based on types of attributes (see Fig. 3, right). The push scheme is more appropriate for those attributes that can be safely collected by the PEP or can only be retrieved by the PEP rather than the PDP (e.g., message annotations whose semantics is known by the PEP: IP address of a buyer and payment server, description of a token service, location and country of a certificate server). A similar treatment may be for attributes unlikely to change frequently: user identity, group membership, security clearances or roles, authentication tokens, or any constant data about system resources or the environment that the PEP can retrieve easily. Conversely, the PDP directly interacts with PIPs when it needs specific application logic and related state (e.g.,

customer rating, payment obligations), history of a user (e.g., bidding history of a user), or data that only the PDP is allowed to access (e.g., highest bid).

3.2 Attribute Caching

Caching in policy enforcement can be of three types: of (1) attributes, (2) decision, and (3) policy [25]. The PDP may cache policies that change little in terms of content, but caching attributes with different change rates is difficult. For instance, bid history, while expected to change a lot, can be constant if the user does not bid; conversely, feedback rating, while expected to change slowly, can change fast for a seller who does not ship items to buyers over a period of time. For similar reasons, the PDP or the PEP might maintain a cache of the decisions already made. Unlike attributes, that tend to change values the most frequently, decisions and policies are more static. We hereafter focus on attribute caching, but our framework can apply to decision and policy caches too.

Table 1. Different caching concerns in enforcement.

Scheme	**What to cache**	**Where to cache**	**How to invalidate**
Attribute caching	attributes	PEP or PDP	time limit
Decision caching	policy decisions	PDP	explicit
Policy caching	entire policy	PEP	explicit

Since cached attributes are attributes too, the push and pull scheme applies to caches as well as to attribute retrieving. New values of the attributes needed to make a policy decision can be either (1) pushed to the entity that needs them, be it the PEP or the PDP, or (2) pulled by the same entity that needs them. There are two cases that combine both caching and attribute retrieving issues:

1. *push to PEP cache, push to PDP*: a PEP pushes attributes to the PDP, and whenever an attribute is updated, the PEP cache is notified and updated. The scheme relies on an external entity to notify the PEP of attribute changes. Inaccurate notifications can compromise decision correctness.
2. *pull to PEP cache, push to PDP*: a PEP that pushes attributes to the PDP, and periodically queries attributes for fresh values (pull). This case does not use a third-party but puts more load on the PEP. Also, the PEP should have poll times that depend on the change rate of the attributes to be cached.

The cases when the PDP pulls attributes by itself and stores them are similar in that cached values need to be refreshed at a rate decided either by a (trusted) third-party or at the rate at which the PDP can query the data sources. From this last point of view – the polling rate of the PDP – cache management has the notion of cache invalidation policy, whereby the cached values have a validity time; when they become invalid, the manager will retrieve fresh copies. Table 1 shows how to invalidate caches depending on what security aspect is cached.

3.3 Attribute Correlation

In an interview with Gunnar Peterson (IEEE Security & Privacy Journal on Building Security), Gerry Gebel (president of Axiomatics Americas) pointed out that there can be different freshness values for related attributes [10]. The example given is that of a user with multiple roles in an organisation; whenever the user requests access to a system resource, the PDP needs to retrieve the current role of the user; some of the user's roles may have been cached (in the PDP, PEP or PIP) hence there might be different freshness values to be maintained for several role attributes. In our eBay scenario in Fig. 2, it can be that more than one item attribute changes from the moment a buyer reads it, and before the last bid. In order for the security subsystem to allow the buyer to bid, it must gather buyer information, item restrictions, *and* item information. If at this stage, the system can access the freshest bid value, but not the freshest item status flag, then the buyer could still be allowed to bid on a sold item.

We generalise such examples to the idea that correlated attribute need a common refresh rate for all the attributes in the group (e.g., all role attributes from an LDAP server, username and password attributes, source IP and port attributes, role and mutually exclusive role list, etc). Some attribute correlations are application-dependent (e.g., mutually exclusive roles, like bidder and item provider for the same item), others are application-independent (e.g., username and password for single sign-on). Hence, bundling attributes that should be used together is a must when enforcement decisions need to be accurate. With PEPs or PDPs likely to use overlapping attribute sets to enforce a cross-domain policy, synchronization over common attributes is essential for attribute consistency.

Having a control over the attributes used in policy enforcement implies the need for a management layer that we call 'configuration layer'. We continue by describing our solution to the issues above.

4 Approach and Proposed Architecture

We consider the setting of a distributed application on top of which there is a security subsystem in charge of enacting several security policies. This subsystem consists of multiple PEPs, PIPs and PDPs. In the state of the art so far, the connections between these components are fixed once a policy is deployed. In our view, they should change at runtime, since this way the system can *adapt* to varying security or performance constraints. How often these changes are incorporated into the security subsystem depends on which of security or performance is paramount, and on the disruption incurred when actually changing the connections among security components.

We will use the term *wiring* for enabling a configuration on the concrete infrastructure (realizing the connections among PIPs, PEPs and PDPs in order to support the attribute management features explained above). We want to rewire the different authorization components with hints from the application domain and knowledge of the enforced policies. We see such hints supplied with the

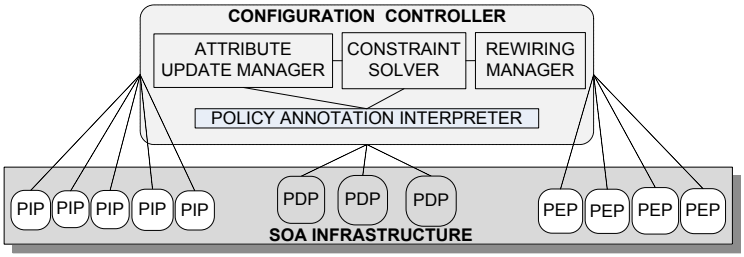


Fig. 4. The architecture of our solution.

XACML policy, since it is likely that the policy writer, aware of some application domain details, can formulate requirements for the treatment of attributes. We also assume that security components have standard features: a cache storage space (hence also cache size and cache refresh policies), permissions over enforcement attributes, or other properties (domain, location, etc).

The architecture of our solution is presented in Fig. 4. Instead of having a set of hardwired connections between the PEPs, PIPs and PDP, we suggest a configuration layer on top of the SOA infrastructure. This layer enables the dynamic rewiring according to varying runtime conditions of the application. Responsible of this task is a Configuration Controller (CC) consisting of four components: a Policy Annotation Interpreter (PAI), an Attribute Update Manager (AUM), a Constraint Solver (CS) and a Rewiring Manager (RM).

The **Policy Annotation Interpreter (PAI)** extracts the policy annotations relevant for the configuration of the security subsystem.

The **Attribute Update Manager (AUM)** monitors the value changes of security attributes and notifies or directly propagates the new values to PEPs and PDPs or to their respective caches. This component also manages attribute synchronization and consistency. It should be connected to all PIPs needed to enforce a policy. The purpose of the AUM is to propagate attribute changes to the security subsystem. The environment, user properties or security relevant state might change at runtime, so the security subsystem needs to be notified.

The **Constraint Solver (CS)** is the component that finds solutions to satisfy the connection constraints of PEPs, PIPs and PDPs. Such constraints cover attribute retrieval, caching and correlation, and are specified in the security policy. The CS processes the policy, searches the solution space, and selects the configurations of the security subsystem that do not violate the constraints.

The **Rewiring Manager (RM)** enacts the solutions found by the CS. The RM resides at the middleware layer and is dependent on the communication infrastructure (in this case, a SOA infrastructure). How this infrastructure is rewired is not within the scope of this paper, and was addressed before [12].

With this approach in mind, we continue by examining types of runtime constraints, how they can be specified, and how the CS can handle them. There are several assumptions for our running system: first, all PIPs are considered to provide fresh information to the other security components. Then, PEPs can cache PIP data, and PDPs can cache enforcement decisions and policies.

4.1 Annotating XACML policies

For the eBay authorization policy that requires attributes 'user token' and 'user feedback rating', our approach is qualitative: if there are multiple providers of user tokens, or if the rating attribute has to be fresh, our aim is to ensure such quality considerations are respected. This additional data (where an attribute can be retrieved from, if it can be stale or pushable) belongs in the authorization policy. The reason is that the policy writer usually has the correct idea over attribute usage and invalidation with respect to the correctness of the policy decision; the policy writer knows if the user token does not change a lot so that it can be pushed to the PDP, or if the buyer/seller feedback rating is critical and volatile so should not be cached. The XACML 3.0 syntax does not natively

```
<xs:element name="AttrProps" type="att-xacml:AttrPropsType"/>
<xs:complexType name="AttrPropsType">
   <xs:attribute name="AttrId" type="xs:anyURI" use="required">
   <xs:sequence>
     <xs:element ref="att-xacml:Providers" minOccurs="1" maxOccurs="unbounded">
     <xs:element name="AttrNotCached" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrCacheable" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrPushable" minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="att-xacml:CorrelAttr" minOccurs="0" maxOccurs="unbounded">
   </xs:sequence>
</xs:complexType>
```

Fig. 5. Attribute meta-data elements in the `att-xacml` schema.

support attributes about subject, environment or resource attributes. We suggest to enrich the default XACML syntax with annotations specific to the following aspects: (1) what PIP/PEP provides an attribute, (2,3) if an attribute can be cached and where, (4) if an attribute is pushable or pullable and where, (5) the correlation among different attributes. Until a XACML profile bridges this gap, a solution to this problem is to specify attribute metadata as an element in an enhanced schema we call `att-xacml` and part of which is shown in Fig. 5. The `AttrPropsType` type consists of the properties of attributes that we are interested in: `attribute-id`, and a series of elements to indicate on what PEPs or PDPs they can be pushed, cached or not cached, what PIPs provide them, and correlated attributes. We assume that these features, attached to each attribute in a policy, are interpreted by an extension of the XACML engine. The PAI processes the semantics of these attributes for the CS.

4.2 Satisfying configuration constraints

The restrictions on attributes that were specified in a XACML-friendly syntax in Section 4.1 will reach the Constraint Solver (CS). The CS has to match these constraints against its runtime view over the authorization subsystem, which we

call *runtime topology*. The result is a number of solutions that satisfy all constraints; we call these solutions *configuration solutions*, or *configurations*. If the CS finds no solution, then the constraints are not satisfiable and the security subsystem remains unchanged. The CS is shown in Fig. 6. Satisfying configura-

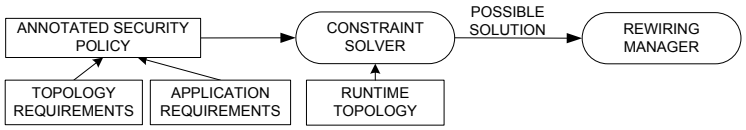


Fig. 6. Model for configuration and rewiring in our approach.

tion constraints is complicated by the existence of different and interdependent constraint types. It is not in the scope of this paper to analyse the impact of each of these constraints over the resulting configurations. Yet, for now, we acknowledge that reconfiguration depends on how often and what triggers the CS to re-evaluate its previously found configurations and issue new ones. Here we see two possibilities: the re-evaluation can be per policy, or per policy subpart. In the first case, the trigger of the reconfiguration is a new policy version, and the changes required for the new configuration might be scarce and far apart in the topology. In the second case, the trigger of a reconfiguration is a change in the runtime topology that relates to an attribute provider or consumer referred to by a security policy; here, the reconfiguration changes are likely to be closer together in the topology. It is hard to say which one happens more often, since this is application dependent. A tradeoff must be made between a system that reconfigures itself either too frequently, or never.

4.3 (Re)Wiring

Rewiring of PEPs, PDPs, and PIPs refers to changing the connections among these components. These connections are established at the SOA *middleware* level, that enables component intercommunication. In our previous work [11], we envisioned the Enterprise Service Bus (ESB) as the manager of the security subsystem. The main reason is that, by design, the ESB controls the deployed security components and their connections; when runtime conditions change (e.g., components appear or disappear, security policies are updated), the ESB can modify message routing on the fly (e.g., validate credentials before they reach an authorization server), trigger attribute queries (e.g., check the feedback rating attribute every minute and not every hour) or change attribute propagation to security domains (e.g., what entities are entitled to receive bid product updates). In particular, the dynamic authorization middleware in [12] applies the dynamic dispatch system of an ESB to enable run-time rewiring of authorization components across services. Each authorization component (PEP, PDP, PIP and PAP) is enriched with an authorization composition contract, and a single administration point allows the wiring and rewiring of authorization components. Using lifecycle and dependency management, the architecture guarantees that

authorization wirings are consistent (i.e., all required and provided contract interfaces match), and that they remain consistent as the rewiring happens. Since rewiring was addressed before, here we focus on the constraint solving problem.

5 Configuration Prototype

From the components shown in Fig. 6, we concentrated on the constraint solver component. While aspects about the RM and the AUM have been approached in previous work, runtime constraint solving is the most challenging aspect of our solution. We assume the CS receives runtime data about the components of the security subsystem, and attribute constraints from the deployed policies (Section 3), and produces configuration solutions that satisfy the constraints. We have prototyped the constraint solving in Java with the SAT4J[7] SAT solver.

5.1 Constraint Solving with a SAT solver

We used a SAT solver because we wanted to obtain configuration solutions for a wide range and number of constraints (Section 5.2 explains why it is difficult to do this without a SAT solver). Given a propositional formula in conjunctive normal form (CNF) describing both the problem and its constraints, a SAT solver is designed to find an assignment of the propositional variables that satisfy the formula. The solver can also address partial maximum (PMAX) SAT problems: given a set of clauses S (a 'maximization set'), find assignments that maximise the number of satisfied clauses in S. This approach naturally maps to the configuration problem that administrators are faced with. For a security-aware application, satisfying all security constraints is paramount, while performance constraints should be maximized if possible. The reciprocal is treated similarly.

Fig. 7 shows two ways to use a SAT solver in our problem: in the first case, an encoded input is fed to the solver to find a valid configuration. If some part of the input changes at runtime, the whole input will be encoded again, and the solver will recompute all solutions from scratch. This method requires a massive effort to re-encode the entire problem, even if only a small part of it has changed. We employed an alternative incremental approach. The SAT solver receives an initial set of clauses, and if a subset of these clauses change at runtime, its internal mechanisms recompute only the subpart of the problem that has changed. A maximisation set can be provided in both cases. Next, we describe the encoding, offline and runtime input in our approach.

Offline Input. The administrator sets up the general settings. The set $PIPs$ of all possible PIPs, the set $PEPs$ of PEPs, and the set $PDPs$ of PDPs, say which attributes can be provided by each component. Specifically, for each $pip \in PIPs$, $pep \in PEPs$, and $pdp \in PDPs$, the following data is provided:

– `provide`(pip, a) iff PIP pip can provide attribute a;

[7] `http://www.sat4j.org/`

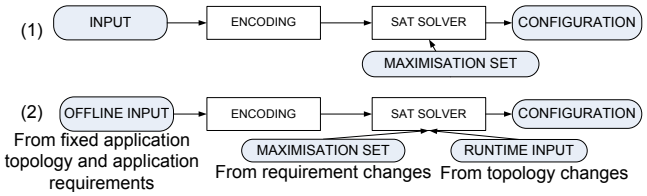


Fig. 7. Two SAT solver methods to solve constraints (inputs and outputs shown darker)

- `provide`(pep, a) iff PEP pep can provide attribute a;
- `needs`(pdp, a) iff PDP pdp needs attribute a;
- `pull`(a, pdp) iff PDP pdp needs attribute a directly from some PIP;
- `correlation`(pdp) iff PDP pdp either pulls or pushes all attributes;
- `not_cacheable`(a, pdp) iff PDP pdp needs freshness for attribute a;
- `cached`(a, pep) iff PEP pep can tolerate a stale attribute a.

These predicates become propositional variables for the solver, along with:

- `arch`(pdp, pip, a) means that a is pulled directly by pdp from pip.
- `arch`(pdp, pep, a) means that a is pushed from pep to pdp.
- `arch`(pep, pip, a) means that a is pushed from pip to pep.

The solver assigns the truth values to the `arch`() variables; these give the active connections of each component, and each set of assignments is a configuration.

We also use the notation $PIPs_a \subset PIPs$ (and $PEPs_a \subset PEPs$) as the set of PIPs (PEPs) such that `provide`(pip, a) (`provide`(pep, a), resp.). Similarly, $PDPs_a \subset PIPs$ is the set of PDPs such that `needs`(pdp, a) is true. In the attribute retrieval model, we consider the following formulae:

$$\texttt{pull}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pip, a) | pip \in PIPs_a\}$$
$$\texttt{push}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pep, a) | pep \in PEPs_a\}$$

saying that an attribute a is pulled by (pushed to) the PDP pdp if and only if there is an arch between pdp and a PIP pip (PEP pep, resp.). Thus, if the pdp asks for a, and `pull`(a, pdp) is true, then a must be retrieved directly from a pip.

We use the notation $\oplus\{v_1, v_2, \dots\}$ meaning that exactly one of $v_1, v_2, \dots$ is true, and $\ominus\{v_1, v_2, \dots\}$ meaning that at most one of $v_1, v_2, \dots$ is true. Encoding the possible paths and constraints means a conjunction of the following formulae:

- For each a required by pdp, exactly one connection must exist between pdp and either a PIP providing a or a PEP providing a. This is expressed by the following formula, for each `needs`(pdp, a) in the Offline Input:

$$\texttt{needs}(pdp, a) \supset \oplus\big\{\texttt{arch}(pdp, x, a) | x \in PIPs_a \cup PEPs_a\big\} \quad (1)$$

  Moreover, in case there is an arch between pdp and pep, providing a, then there must be also a connection between pep and a pip providing a (formula (2)). Otherwise (formula (3)) no arch between pep and pip providing a

is necessary. For each $\texttt{needs}(pdp, a)$ and $pep \in PEPs_a$:

$$\texttt{arch}(pdp, pep, a) \supset \oplus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \quad (2)$$

$$\neg\,\texttt{arch}(pdp, pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \quad (3)$$

– If a *pip* does not provide a then no connection providing a can exist between *pip* and other components. For each $\texttt{provide}(pip, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\}$$

$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pep \in PEPs_a\}$$

Similarly for the PEPs: for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$

$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

– To show the advantage of this approach, we consider another constraint: we suppose that each component can provide (e.g. for performance reasons) an attribute only to a finite number of components (in this case, one). To model that, for each $\texttt{provide}(pip, a)$ in the Offline Input, we consider the following:

$$\texttt{provide}(pip, a) \supset \ominus\{(\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\} \quad (4)$$

$$\texttt{provide}(pip, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pep \in PEPs_a\} \quad (5)$$

Formula (4) (same for (5)) says that if *pip* provides a then there must be at most one connection between *pip* and the PDPs (PEPs, resp.) requesting a. It is similar for the PEPs, so for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$

$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

– Without data showing that PDPs cache attributes, we considered only PEPs to do that. This choice does not limit generality. We had to encode the constraint: if *pdp* asks for a, and $\texttt{not\_cacheable}(a, pdp)$ is true, then a cannot be retrieved from a *pep* that cached a. So for each $\texttt{not\_cacheable}(a, pdp)$ in the Offline Input, we have the conjunction:

$$\texttt{not\_cacheable}(a, pdp) \supset \bigwedge\{\neg(\texttt{cached}(a, pep) \wedge \texttt{arch}(pdp, pep, a)) | pep \in PEPs_a\}$$

– The attribute correlation constraint can be described as follows: if *pdp* has correlated attributes, then all its attributes must be either pulled or pushed. For each $\texttt{needs}(pdp, a)$ in the Offline Input, this can be modeled as follows:

$$\texttt{correlation}(pdp) \supset (\texttt{pull}(pdp) \vee \texttt{push}(pdp))$$

where: $\texttt{pull}(pdp) \equiv \bigwedge\{\texttt{pull}(a, pdp)\}$, and $\texttt{push}(pdp) \equiv \bigwedge\{\texttt{push}(a, pdp)\}$

Notice that by defining PMAX-SAT problems it is also possible to impose constraints like: if possible, preference should be given to attribute provisioning via indirect paths, i.e., from PIP to PEP to PDP, over direct paths, i.e., from PIP to PDP. This can be obtained by maximising the number of truth values for the variables corresponding to the arches from PEPs to PDPs.

In satisfiability problems, coding the at-most-one operator $\ominus$ is often problematic. If there are $n > 1$ variables, an improved SAT encoding of $\ominus\{v_1, \ldots, v_n\}$ is the logarithmic bitwise encoding in [9]. This operator's CNF encoding requires $\lceil \log_2(n-1) \rceil$ auxiliary variables and $n\lceil \log_2(n-1) \rceil$ clauses. With this encoding, the CNF formula corresponding to (1) has a complexity (in terms of number of clauses) of $O(|PDPs| \cdot \mathrm{N_{a\_pdp}} \cdot (\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}}) \cdot \log_2(\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}})$, where $\mathrm{N_{a\_pdp}}$ is the average number of attributes requested by each PDP, and $\mathrm{N_{pips\_a}}$ ($\mathrm{N_{peps\_a}}$) is the average number of PIPs (PEPs, resp.) providing a specific attribute. These constraints generate a large number of clauses. This is why the encoding is computed offline and fed to the solver "on stand-by mode".

Runtime Input. The offline input abstracts a global view of the scenario. The runtime input specifies a current scenario, indicating the security components currently being used, their current set of attributes provided/needed, and any other current constraints. The runtime input is an instantiation of the offline input. For instance, if the Offline Input file contains `provide(pip1, a1)` and `provide(pip2, a2)`, but in the current scenario `pip1` is not available, then the Runtime Input will contain the following assignment: `provide(pip1, a1) = false`.

5.2 Constraint Solving without a SAT solver

For independent constraints over components, an algorithm to choose configuration solutions may be faster than the SAT solver. Such an algorithm can perform a graph traversal to select the first PIP or PEP that satisfies the requirements of a PDP and some boolean constraints. Graph traversal works best in scenarios where existing connections have no impact on choosing further connections; yet it may become problematic when the graph is dynamic and local connections change global state, that in turn changes other local connections. This can happen, for example, when bandwidth restrictions limit the number of possible connections of a component; or when a user cannot re-rate a seller until the next time they win the auction. Designing such an algorithm is not easy, since it is tightly bound to the constraints: whenever constraints appear or disappear, the algorithm needs to be rewritten. With a SAT solver, the set of constraints can vary without influencing the process of producing a solution, but performance might suffer. Still, it is worth to design such a generic algorithm as future work.

6 Evaluation and Discussion

The performance of our prototype depends on several aspects: the size and semantics of the offline input file, the size and semantics of the runtime input, and

the internal performance of the SAT solver. Since each SAT solver uses a different solving technique, the performance of our prototype depends on the choice of solver. Apart from this dependency, we considered that the most relevant aspect to test is the encoding of the offline input, that need be recomputed several times along the lifetime of an application. This recomputing can be triggered by components that appear or disappear, or by changes in what components provide or require. The runtime input, on the other hand, is only a (moderately small) subset of the maximal offline one, and its impact over the solution finding time depends on the SAT technique used by each individual solver.

Therefore, we wanted to measure the offline CNF generation and the constraint solving time for a number of configuration topologies, while keeping the runtime input minimal. To obtain different offline inputs, there were several parameters to vary: the number of authorization components, the number of attributes and constraints on attributes, the distribution of attributes per authorization component. In choosing which of these parameters to vary and which to keep constant, we considered that the number of attributes is fixed in each scenario, since the administrator should know beforehand the attributes required for all security decisions in a particular application. Hence, we generated over 100 offline input files to describe topologies with varying numbers of PIPs, PDPs, and PEPs, with a fixed number of attributes provided/needed by each. In order to assign attributes to each component, we implemented an algorithm for unbiased random selection as described by Knuth [18, Sec.3.4.2].

To test our Java encoding against the SAT4J solver with the aims described above, we configured the JVM to run with 500 to 1000MB of memory, and we used JRE1.5.0. We ran each of the input files against a minimal runtime configuration file of one constraint, with the following parameter changes: 25, 50, 75 PDPs; 50, 100, 250, 500, 750 PIPs; 25, 50, 75, 100, 150, 200, 250 PEPs. The number of attributes was constant to 100, with 3 attributes per component in all cases. We measured the time (in milliseconds as per the Java call $System.currentTimeMillis()$) for the static CNF generation, as well as the time it takes SAT4J to compute the first solution from the already generated encoding (the solve time). The results for 50 PDPs can be seen in Fig. 8; those for 25 and 75 PDPs are similar. The figures show a linear increase in the offline CNF generation in the number of authorization components (Fig. 8, right) with a similar linear increase in solve time (Fig. 8, left).

These preliminary results are very encouraging in that the time taken to compute the constraints from the offline input is very short for a moderate application size (e.g., about 2.5 seconds for 750 PIPs, 250 PEPs and 50 or 75 PDPs), and also that they grow linearly with the number of components, for a fixed number of attributes. Even though satisfiability is generally known to be an NP-complete problem (and on a general case, it can be expected to obtain an exponential growth in problem complexity and hence performance), the linear relation that we have obtained is justified in that we are using a generic tool to solve a particular instance of a satisfiability problem.

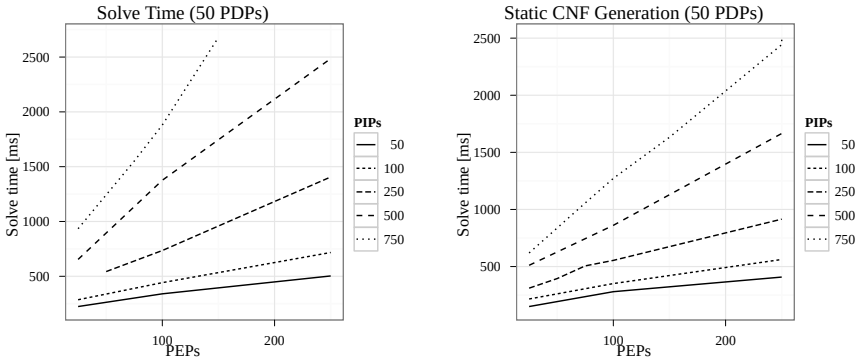


Fig. 8. The time to solve our constraint problem with 50PDPs, and varying numbers of PEPs and PIPs (left) and the time to generate the offline CNF clauses (right)

7 Related Work

Some policy configuration features come with commercial products: Tivoli Access Manager [13], Axiomatics Policy Server [3], Layer 7 Policy Manager [19], Shibboleth [14] and Globus Toolkit 4 [1]. Tivoli provides central management and enforcement of data security policies across multiple PEPs, with partial policy replication and decision caching. Axiomatics offers central administration of policies, and can manage multiple PDP instances. Shibboleth focuses on federated identity management, allows for session caching and attribute backup, but does not consider attribute freshness. Globus considers resource and policy replication. All these products address different policy management aspects differently, and never consider together cache consistency, synchronisation and security guarantees. The users of these products cannot change such features from a single console, and hence cannot understand how one impacts another.

Two previous works are particularly relevant to our approach. First, Colombo et al. [6] observe that attribute updates may invalidate PDP's policy decisions. The work only focuses on XACML's expressive limitations, does not consider system reconfigurations and is not supported by any implementation. Second, the work of Goovaerts et al. [12] focuses on matching of provisions in the authorization infrastructure. The aim is to make the system that enforces a policy seamless and scalable when those components that provide security attributes change or disappear. They do not discuss how attribute provisioning impacts the correctness of the enforcement process.

Several other works are related to ours. Policy management is also tackled in Ponder [8] but aspects such as caching consistency and synchronisation are not considered. PERMIS [5] proposes an enforcement model for the Grid that uses centralised context data stores and PDP hierarchies with visibility restrictions over attributes provided across domains. Like us, Ioannidis [15] separates policy enforcement from its management, suggests a multi-layer architecture to enforce locally some part of a global policy, but does not discuss consistency of policy

data at different locations and propagation of updates. The thesis of Wei [25] is the first to look at PDP latency in large distributed applications for role based access control. He does authorization recycling by caching previous decisions, but does not consider cache staleness. Atluri and Gal [2] offer a formal framework for access control based on changeable data, but the main difference is that their work is on changing the authorization process rather than configuring security components to manage such data appropriately.

8 Conclusions and future work

Our work is the first to consider the impact of properties about authorization components and their connections, over the correctness of the enforcement process. We believe that performance tuning uncovers security flaws in distributed applications and we concentrate on security attribute management. Our management solution can help administrators in two ways: it can generate system configurations where a set of security constraints need always be satisfied (along with maximising performance constraints), or can check an existing configuration of the system against a given set of (security or performance) constraints. The configuration solutions can be recomputed at runtime and preliminary results show an overhead of a few seconds for a system with one thousand components. To our knowledge, this is the first tool to perform a fully verified authorization system reconfiguration for a setting whose security constraints would otherwise be impossible to verify manually.

Acknowledgements. This work was partially supported by EU-FP7 NESSoS, the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, the Research Fund K.U.Leuven, and by the "Automated Security Analysis of Identity and Access Management Systems (SIAM)" project funded by Provincia Autonoma di Trento within "team 2009-Incoming" FP7-COFUND. We also thank Wouter De Borger and Stephan Neuhaus for technical comments.

References

1. Globus Alliance: Globus Toolkit 4 API. `http://www.globus.org/toolkit/docs/4.2/4.2.1/security/` (Nov 2010)
2. Atluri, V., Gal, A.: An authorization model for temporal and derived data: securing information portals. ACM Trans. Inf. Syst. Secur. 5, 62–94 (February 2002)
3. Axiomatics: Axiomatics Policy Server 4.0. `http://www.axiomatics.com/products/axiomatics-policy-server.html` (Nov 2010)
4. Chadwick, D., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: PERMIS: a modular authorization infrastructure. Concurr. Comput. : Pract. Exper. 20, 1341–1357 (2008)
5. Chadwick, D.W., Su, L., Laborde, R.: Coordinating access control in grid services. Concurrency and Computation: Practice and Experience 20(9), 1071–1094 (2008)

6. Colombo, M., Lazouski, A., Martinelli, F., Mori, P.: A Proposal on Enhancing XACML with Continuous Usage Control Features. In: Grids, P2P and Services Computing. pp. 133–146. Springer US (2010)
7. Djordjevic, I., Dimitrakos, T.: A note on the anatomy of federation. BT Technology Journal 23, 89–106 (October 2005)
8. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A policy deployment model for the ponder language. In: Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on. pp. 529 –543 (2001)
9. Frisch, A.M., Peugniez, T.J., Doggett, A.J., Nightingale, P.W.: Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. J. Autom. Reason. 35, 143–179 (October 2005)
10. Gebel, G., Peterson, G.: Authentication and TOCTOU. http://analyzingidentity.com/2011/03/18/ (2011)
11. Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In: IFIPTM. vol. 321, pp. 63–78 (2010)
12. Goovaerts, T., Desmet, L., Joosen, W.: Scalable authorization middleware for service oriented architectures. In: ESSoS, Madrid, Spain (Feb 2011)
13. IBM: IBM Tivoli Access Manager. http://www-01.ibm.com/software/tivoli/products/access-mgr-e-bus/ (Nov 2010)
14. Internet2MiddlewareInitiative/MACE: Shibboleth 2. https://wiki.shibboleth.net/confluence/display/SHIB2/Home (2011)
15. Ioannidis, S., Bellovin, S.M., Ioannidis, J., Keromytis, A.D., Anagnostakis, K.G., Smith, J.M.: Virtual private services: Coordinated policy enforcement for distributed applications. I. J. Network Security 4(1), 69–80 (2007)
16. Kassaei, F.: eBay identity assertion framework. http://www.slideshare.net/farhangkassaei/ebay-identity-assertion-framework-iaf (May 2010)
17. Kerner, S.M.: Inside Facebook's Open Source Infrastructure. http://www.developer.com/open/article.php/3894566/ (July 2010)
18. Knuth, D.E.: The art of computer programming, vol. 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
19. Layer 7 Technologies: Policy Manager for XML Gateways. http://www.layer7tech.com/products/policy-manager-for-xml-gatewaysl (Nov 2010)
20. Miller, R.: The Facebook Data Center FAQ. http://www.datacenterknowledge.com/the-facebook-data-center-faq/ (September 2010)
21. Mitre: Common Vulnerabilities and Exposures. http://cve.mitre.org/ (2011)
22. Shoup, R.: Scalability best practices - Lessons from eBay. InfoQ http://www.infoq.com/articles/ebay-scalability-best-practices (May 2008)
23. Shoup, R.: More Best Practices from Large Scale Websites - Lessons from eBay. Talk at QCon San Francisco 2010, http://qconsf.com/sf2010 (November 2010)
24. Wei, D., Jiang, C.: Frontend Performance Engineering in Facebook. O'Reilly Velocity, Web Performance and Operations Conference (June 2009)
25. Wei, Q.: Towards Improving the Availability and Performance of Enterprise Authorization Systems. Ph.D. thesis, University of British Columbia (2009)