# Cooperative Secondary Authorization Recycling

Qiang Wei, Matei Ripeanu, Konstantin Beznosov
Laboratory for Education and Research in Secure Systems Engineering
University of British Columbia
Vancouver, Canada
{qiangw, matei, beznosov}@ece.ubc.ca

## Abstract

*As distributed applications such as Grid and enterprise systems scale up and become increasingly complex, their authorization infrastructures—based predominantly on the request-response paradigm—are facing the challenges in terms of fragility and poor scalability. We propose an approach where each application server caches previously received authorizations at a secondary decision point (SDP) and shares them with other servers to mask authorization server failures and network delays.*

*The main contribution of this paper is the design of the cooperative secondary authorization recycling (CSAR) system and its evaluation using simulation and prototype implementation. The results demonstrate that our approach improves the availability and the performance of authorization infrastructures. Specifically, by sharing secondary authorizations among SDPs, the cache hit rate—an indirect metric of availability—can reach 70% even when only 10% of authorizations are cached. The performance is also improved, as the average time of authorizing a request to an application is reduced by up to 30%.*

## 1 Introduction

Architectures of modern access control solutions—such as [17, 10, 21, 26, 23, 9]—are based on the request response paradigm, as illustrated in the dashed box of Figure 1. In this paradigm, the policy enforcement point (PEP) intercepts application requests, obtains access control decisions (or authorizations) from the policy decision point (PDP), and enforces those decisions.

In large enterprise systems, PDPs are commonly implemented as centralized authorization servers, providing important benefits: consistent policy enforcement across multiple PEPs and reduced administration costs of authorization policies. As all centralized architectures, this architecture has two critical drawbacks: the PDP is a single point of failure and a potential performance bottleneck and.

The single point of failure leads to reduced availability: the authorization server responsible for making authorizations decisions may not be reachable due to a failure (transient, intermittent, or permanent) of the network, of the software located in the critical path (e.g., OS), of the hardware, or even a misconfiguration of the supporting infrastructure. A conventional approach to improving availability of a distributed infrastructure is failure masking through redundancy–either information, time, or physical redundancy [13]. However, redundancy and other general purpose fault-tolerance techniques for distributed systems scale poorly, and become technically and economically infeasible to employ when the number of entities in the system reaches thousands [14, 28].

In a massive-scale enterprise system with non-trivial authorization policies, making authorizations is often computationally expensive due to the complexity of the policies involved and large size of the resource and user populations. Thus the centralized PDP often becomes a performance bottleneck [22]. Additionally, the communication delay between the PEP and the PDP—added to the inherent cost of computing an authorization decision—can make authorization overheads prohibitively expensive.

The state-of-the-practice approach to improving overall system availability and reducing client observed processing delays is to cache authorization decisions at each PEP—what we refer to as *authorization recycling*. Existing authorization solutions commonly provide PEP-side caching [17, 10, 21]. These solutions, however, only employ a simple form of authorization recycling: a cached decision is reused only if the authorization request in question exactly matches the original request for which the decision was made. We refer to such reuse as *precise recycling*.

To improve authorization system availability and reduce the delay, Crampton et al. [8] introduce the secondary and approximate authorization model (SAAM) which adds a secondary decision point (SDP) to the traditional request-response paradigm ( Figure 1). The SDP is collocated with

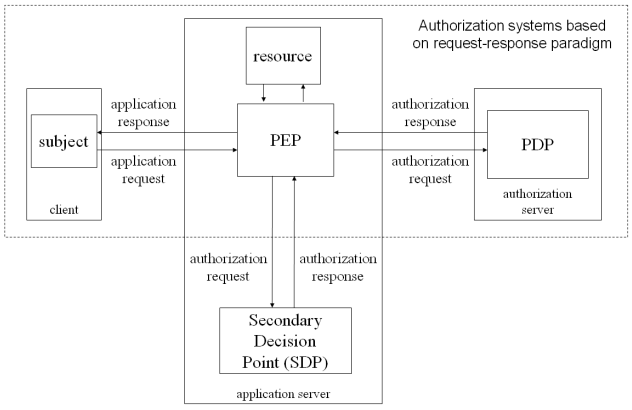



**Figure 1. SAAM introduces SDP in authorization systems based on request-response paradigm**

the PEP and can resolve authorization requests not only by reusing cached precise authorizations but also by inferring approximate authorizations from cached precise authorizations. The employment of approximate authorizations improves the availability and performance of the access control sub-system, which ultimately improves the observed availability and performance of the applications themselves.

In the solution proposed by Crampton et al., however, each SDP serves only its own PEP, which means that authorization decisions are reusable only for the requests made through the same SDP. In this paper, we propose an approach where different SDPs act jointly to serve all PEPs. Our main goal is to further improve the resilience of the authorization infrastructure to network and authorization server failures and reduce the delay of producing an authorization.

We believe that our approach is especially applicable for the distributed systems involving cooperating parties, such as Grid systems. First, different cooperating parties are likely to have the same set of users and the similar type of resources. Second, using centralized authorization servers in Grid [29, 24] implies that different cooperating parties are likely to enforce the same access control policies. These two observations leads to the conclusion that authorization decisions can often be shared among cooperating parties, thus bring the benefits to each other.

This paper makes a number of contributions as follows: we propose the concept of cooperative secondary authorization recycling (CSAR), analyze its design requirements, and propose a concrete architecture. We demonstrate its feasibility and benefit by simulation and evaluating a CSAR prototype.

The evaluation results show that our approach improves availability and performance. Specifically, by recycling secondary authorizations between SDPs, the hit rate can reach 70% even when only 10% of authorizations are cached at each SDP. This high hit rate results in more requests being resolved by the local and other cooperating SDPs, even when the authorization server is unavailable or slow, thus increasing availability of authorization infrastructures and reducing the load of the authorization server. Additionally, our experiments show that the request processing time can be improved up to 30% in our experimentation setup.

The rest of this paper is organized as follows. Section 2 presents the secondary and approximate authorization model. Section 3 describes the CSAR design. Section 4 evaluates CSAR's performance through simulation and a prototype implementation. Section 5 discusses related work. Finally, we summarize in Section 6.

## 2 SAAM

This section presents SAAM definitions and algorithms which build on the rest of this paper.

SAAM formally defines authorization request and authorization response. An authorization request is a tuple $(s, o, a, c, i)$, where $s$ is a subject, $o$ is an object, $a$ is an access right, $c$ is the request contextual information, and $i$ is a request identifier. Two requests are equivalent if they only differ in their identifiers. An authorization response for request $(s, o, a, c, i)$ is a tuple $(r, i, E, d)$, where $r$ is the response identifier, $i$ is the corresponding request identifier, $d$ is a decision and $E$ is the evidence, a list of response identifiers that were used for computing the response. The evidence can be used for verifying the correctness of a secondary response.

In addition, SAAM defines the *primary*, *secondary*, *precise*, and *approximate* authorization responses. The primary response is a response made by the PDP and the secondary response is a response produced by the SDP. A response is precise if it is a primary response for the request in question or a response for an equivalent request. Otherwise, if the SDP infers the response based on the responses to other requests, the response is approximate. In the rest of this paper, we use "request" as a short for "authorization request" and "response" as a short of "authorization response".

The inference algorithm for approximate responses depends on the access control model. In the case of the Bell-LaPadula (BLP) model [2], responses to requests enable us to infer information about the relative ordering on security labels associated with subjects and objects. If, for example, three requests: $(s_1, o_1, read, c_1, i_1)$, $(s_2, o_1, append, c_2, i_2)$, $(s_2, o_2, read, c_3, i_3)$ are allowed by the PDP, it can inferred that $\lambda(s_1) > \lambda(o_1) > \lambda(s_2) > \lambda(o_2)$, where $\lambda$ is the security function mapping an object or subject to its security label. Therefore, a request $(s_1, o_2, read, c_4, i_4)$ should also be allowed and the corre-

sponding response is $(r, i_4, [i_1, i_2, i_3]$, allow). Crampton et al. [8] present simulation results that demonstrate the effectiveness of this approach: with only 10% authorizations cached, the SDP can resolve over 30% more authorization requests than a conventional PEP.

The rest of the paper presents the CSAR design and evaluation results that demonstrate that cooperation among SDPs is possible and it can further improve access control system availability and performance.

# 3 CSAR Design

This section presents the design requirements for cooperative authorization recycling, the CSAR system architecture, and finally the detailed CSAR design.

## 3.1 Design Requirements

The CSAR system aims to improve the availability and performance of access control infrastructures by sharing authorization information among cooperative SDPs. Each SDP resolves the requests from its own PEP either by locally making secondary authorization decisions, by involving other cooperative SDPs in the authorization process, or, finally, by passing the request to the remote PDP.

As the system involves caching and cooperation, we consider the following design requirements:

**Low overhead** As each SDP participates in the authorization decision for some non-local requests, its load is increased. The design should minimize this additional overhead.

**Trustworthy** As each PEP enforces responses that are possibly offered by non-local SDPs, which might be malicious, the PEP should be able to verify the validity of each response, by for example tracing it back to a trusted source.

**Consistency** Brewer [4] conjectures and Lynch et al. [12] prove that distributed systems can not simultaneously provide the following three properties: availability, consistency, and partition tolerance. We believe that availability and partition tolerance are essential properties an access control system should offer. We thus relax consistency requirements in the following sense: with respect to an update action, various components of the system can be inconsistent for at most a user-configured finite time interval.

**Configurability** The system should be configurable to adapt to different performance objectives at various deployments. For example, a deployment with latency sensitive set of applications may require that requests

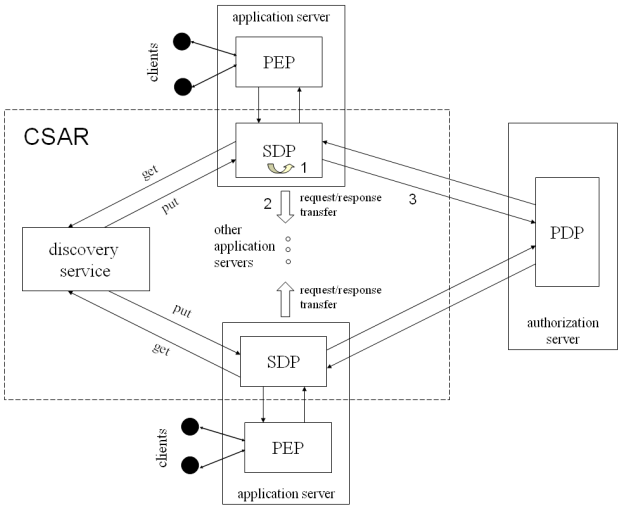



**Figure 2. CSAR introduces cooperation between SDPs. The discovery service is used for**

are resolved in minimal time, whereas a deployment with applications generating a high volume of authorization requests should attempt to aggressively exploit caching and the inference of approximate authorizations to reduce load on the PDP, the bottleneck of the system.

**Backward compatibility** The system should be backward compatible in the sense that minimal changes should be required to existing infrastructure—i.e., PEPs and PDP—in order to switch to CSAR.

## 3.2 Adversary Model

CSAR system involves multiple components: the PDP, PEPs, SDPs. The PDP is the ultimate authority for access control decisions. We only assume that the PDP cannot be compromised. The adversary can eavesdrop or spoof any network traffic or compromise an application server host with its PEP(s) and SDP(s). The adversary can also compromise the client computer(s). Therefore, there could always be one or more malicious clients, SDPs or PEPs in the system. Additionally, requests and responses could be eavesdropped, spoofed, or replayed.

## 3.3 System Architecture

This section presents an overview of the system architecture and discusses our design decisions to address the configurability and backward compatibility requirements.

As illustrated by Figure 2, a CSAR deployment contains multiple PEPs, SDPs, and one PDP. Each SDP is host-collocated with its PEP at an application server. Both PEP and SDP are either part of the application or the underlying middleware. The PDP is located at the authorization server and provides authorization decisions to all applications. The PEPs mediate the application requests from clients, generate authorization requests, and enforce the authorization decisions made by either the PDP or the SDPs.

For increased availability and lower load on the central PDP, our design exploits the cooperation between SDPs. Each SDP, computes responses for requests from its PEP, and can participate in computing responses to requests from other SDPs. Thus, authorization requests and decisions are transferred not only between the application servers and the authorization server, but also between cooperating application servers.

CSAR is configurable to optimize the performance requirements of each individual deployment. Depending on the specific application, geographic distribution and network characteristics of each individual deployment, performance objectives can vary from reducing the overall load on the central PDP, to minimizing client-perceived latency, to minimizing the generated network traffic.

Configurability is achieved by controlling the amount of parallelism in the set of operations involved in resolving a request: (1) the local SDP can resolve the request using data cached locally, (2) the local SDP can forward the request to other cooperative SDPs to resolve it using their cached data, and (3) the local SDP can forward the request to the central PDP. If the performance objective is reducing latency, then the above three steps can be performed concurrently and the SDP will use the first response received. If the objective is reducing network traffic and/or the load at the central PDP, then the above three steps are performed sequentially.

CSAR is designed to be easily integrated with existing access control systems. Each SDP provides the same interface to its PEP as the PDP thus the CSAR system can be deployed incrementally without requiring any change to existing PEP or PDP components. Similarly, in systems that already employ authorization decision caching but do not use CSAR the SDP can offer the same interface and protocol as the legacy component.

### 3.4 Discovery Service

One essential component to enable cooperative SDPs to share their authorizations is the discovery service (DS) which helps an SDP to find other SDPs that might be able to resolve a request.

A naive approach to implement the discovery functionality (similar to a popular deployment configuration of Squid [11], a cooperative Web-page proxy cache) is request broadcasting: whenever an SDP receives a request from its PEP, it broadcasts the request to all other SDPs in the system. All SDPs attempt to resolve the request and the PEP enforces the response it receives first.

This approach is straightforward and might be effective when the number of cooperating SDPs is small and the cost of broadcasting is low. It has however two important drawbacks. First, it inevitably increases the load on all SDPs. Second, it causes high traffic overheads when SDPs are geographically distributed.

To address these two drawbacks, an SDP in CSAR distributes requests selectively: only to those SDPs that are likely to be able to resolve them. We introduced the DS to achieve this selective distribution. To be able to resolve a request, an SDP either caches the response or can infer an approximate response. For this purpose, both the subject and object of the request have to be present the SDP's cache.

The role of the DS is to store the mapping between entities and SDPs. In this paper we use $entity$ as a general term for either a subject or an object. The DS provides interface with the following two functions: $put(entity, SDPaddress)$ and $get(entities)$, where $entities$ is a pair of subject and object. Given an entity and the address of an SDP, the put function stores the mapping (entity, SDPaddress). A put operation can be interpreted as that "this SDP knows something about the entity." Given a pair of subject and object, the get function returns a list of SDP addresses which are mapped to both the subject and the object. The results returned by the get operation can be interpreted as "these SDPs know something about both the subject and the object thus might be able to resolve the request involving them".

This way we avoid broadcasting requests to all SDPs. Whenever an SDP receives a response for a request, it registers itself as a suitable SDP for the subject and the object of the request in the DS. When a new request comes, the SDP can retrieve a set of addresses of those SDPs that might be able to resolve this request.

Note that the DS is logically centralized, but can have a scalable and resilient implementation. Compared to the PDP, the discovery service is simple—it only performs put and get operations,— and general—it does not depend on any particular security policy. As a result a scalable and resilient implementation is easier to achieve. In fact, the discovery service can be easily implemented on top of a distributed hash table with proved reliability and scalability properties [25].

We do not assume that the DS is trusted by any component of the system. Our design limits a malicious DS to the ability to compromise only the performance but not the correctness of the system.

### 3.5 Response Verification

A malicious SDP could generate any response it wants, for example, deny all requests thus launching a denial of service (DoS) attack. Therefore, when an SDP receives a secondary response from other SDPs, it has to verify the response in terms of both response integrity and the correctness of the inference process that produced the response. To enable response verification, the PDP needs to sign every primary response. Each SDP can independently verify the integrity of each primary response assuming it has access to the PDP's public key.

As we mentioned in Section 2, an SDP can generate two types of secondary responses: precise responses and approximate responses. Because each precise response is generated from an equivalent primary response, it can be verified by simply validating its signature. For verifying approximate responses, we use the evidence part of the response.

The evidence part of an approximate response contains a list of primary responses that have been used to infer that response. When an SDP receives a response with a non-empty evidence $E$, the SDP first verifies the signature of each primary response in the evidence list. Second, the SDP uses the evidence list to prove the correctness of the response. If any of these two steps fails, the verification fails and the response is ignored.

Verification of each approximate response unavoidably introduces computational cost, which depends on the length of the response list in the evidence part. Therefore, we defined four execution scenarios. Based on the administration policy and deployment environment, the verification process can be configured differently to achieve various tradeoffs between security and performance.

**Total verification** All responses are verified.

**Allow verification** Only 'allow' responses are verified. This configuration protects resources from unauthorized uses but might be vulnerable to DoS attacks.

**Random verification** Responses are randomly selected for verification.

**Offline verification** No real-time verification is performed by SDPs, but there are offline audits.

In the above-mentioned procedure, we assume that each PEP trusts its own SDP. If it is not the case, the PEP needs to verify the response returned from its own SDP. To prove the correctness of the response, the PEP needs to implement the same verification algorithm. This, however, might be undesirable because the PEP implementation would start to depend on CSAR.

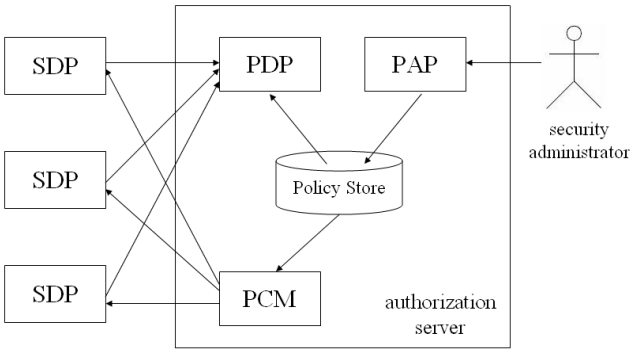



**Figure 3. The architecture of authorization server**

### 3.6 Cache Consistency

Similar to other distributed systems involving caches, CSAR needs to maintain cache consistency. In our system, the caches become inconsistent when access control policies change at the PDP but some SDPs do not update accordingly. Consequently, the SDPs may begin making stale decisions. In this section, we describe the mechanism used by CSAR to provide cache consistency.

#### 3.6.1 Assumptions

For context, we first state a few assumptions relevant to access control systems. We begin with the architecture of the authorization server.

As Figure 3 shows, we assume that access control polices are stored persistently in the policy store of the authorization server. The PDP makes authorization decisions against the policy store. In practice, the policy store can be a policy database or a collection of policy files.

We further assume that security administrators deploy and update policies through the policy administration point (PAP); this is consistent with the standard XACML architecture [30]. In addition, we assume that the PAP interface provides security administrators the option to specify the criticality of the ongoing change. We elaborate this aspect later when we describe the requirements.

Finally, we assume that there are only fail-stop failures in the system.

#### 3.6.2 Requirements and Design Decisions

The key requirement of our cache consistency mechanism is efficiency. Specifically, providing cache consistency should not add much server overhead and network traffic. To address this requirement, we divide all the policy changes into three categories: critical changes, time-sensitive changes,

and time-insensitive changes. The division is based on yet another assumption that not all policy changes are at the same level of criticality. By discriminating policy changes according to these types, we were able to employ different consistency techniques to achieve efficiency for each type. Our design allows a CSAR deployment to support any combination of the three types.

In addition, the mechanism should maintain backward compatibility with existing authorization servers, which requires that the PDP is not aware of the existence of SDPs. Therefore, we can not modify the PDP to support cache consistency. To address this requirement, we add policy change manager (PCM), collocated with the policy store. The PCM monitors the policy store and detects policy changes, and informs the SDPs about the changes (Figure 3).

In the rest of this section, we define three types of policy changes and discuss cache consistency approaches for each.

#### 3.6.3 Support for Critical Changes

Critical changes in authorization policies are those changes that need to be propagated urgently throughout the enterprise applications, requiring immediate updates on all SDPs. For instance, an error found in the access control policy which, if not fixed immediately, will result in leaking customers' personal information to the public and possibly causing a substantial damage to the company reputation. Therefore, an immediate correction in the policy is critical.

When an administrator makes a *critical change* in the authorization policy, our approach requires that she also specifies a time period $t$, within which CSAR must $inform$ the administrator whether all the SDPs have been successfully updated. Period $t$ is assumed to be just few minutes. To accommodate variations in requirements for efficiency, we devised two approaches for updating SDPs with critical changes: "all-flush" and "selective-flush".

In the **all-flush approach**, the PCM broadcasts policy update message to all the SDPs. Upon receiving the message, each SDP flushes its cache and reports to the PCM success or failure of flushing. When all the SDPs have replied or the time period $t$ ends, the PCM reports the results to the administrator. For the SDPs without acknowledgment or with failed results, the administrator has to take out-of-band measures for flushing caches of those SDPs, e.g., by manually restarting corresponding machines. We assume that, given an IP address of an application server, the administrator can identify the physical location of the machine.

The above process requires the PCM to wait period $t$ if some SDPs are unavailable due to network partitions. To avoid waiting, the PCM can run a failure detection service like Google Chubby service [5]. With Chubby, each SDP can be viewed as a PCM client that maintains a Chubby session through periodic handshakes with the PCM. The PCM thus knows the live status of each SDP, which allows the PCM to avoid contacting dead SDPs.

The "all-flush" approach is simple. However it is inefficient for three reasons:

First, not all critical changes need to be propagated because they might have no affect on any cached response. For example, when a policy change is due to merely the addition of an object or user (a.k.a., subject), this change clearly can not invalidate any cached response. Therefore, the propagation for such policy changes is unnecessary.

Second, even when a change affects some cached responses, not all SDPs may have cached those responses. For example, if a user has been revoked access right(s), but only one SDP has ever cached the responses for this user, then other SDPs do not have to act upon the revocation.

Last but not least, not all cached responses get invalidated by every policy change. Using previous example, the SDP only needs to flush those cached responses that involve the user in question. The rest of the cached responses will remain valid.

Using the above observations, we developed a more efficient approach to propagating critical policy changes.

The second approach is called **selective-flush** because only some SDPs are updated and only selected cache entries are flushed. Compared to the "all-flush" approach, the "selective-flush" approach has the benefit of reducing the server overhead and network traffic while minimizing the impact on the system availability. We sketch the propagation process here.

The PCM first determines which subjects and/or objects are affected by the policy change. Since most modern enterprise access control systems make decisions by comparing security attributes (e.g., roles, clearance, sensitivity, groups) of subjects and objects, the PCM maps the policy change to the entities whose security attributes are affected. For example, if permission $p$ has been revoked from role $r$, then the PCM determines all objects of $p$ (denoted by $O^p$) and all users assigned to $r$ (denoted by $S^r$).

The PCM then finds out which SDPs need to be notified of the policy change. Given the entities affected by the policy change, the PCM uses the discovery service (DS) to find those SDPs that might have responses for the affected entities in their caches. The PCM sends the DS a policy change message containing the affected entities, $(O^p, S^r)$. Upon receiving the message, the DS first replies back with a list of the SDPs, that have cached the responses for the entities. Then it removes entries with those SDPs and entities from its map to reflect the flushing. After the PCM gets the list of SDPs from the discovery service, it multicasts the policy change messages to these affected SDPs.

In case that the discovery service is not available, the PCM simply broadcasts the policy change message to all

the SDPs in the system. However, this contingency tactic is still better than the "all-flush" approach because the message indicates the entities to be removed.

When an SDP receives a policy change message, it flushes those cached responses that contain the entities and then acknowledges the PCM the results. In the above example with revoking permission $p$ from role $r$, the SDP would flush those responses from its cache that contain both objects in $O^p$ and subjects in $S^r$. The rest of the process is similar to the "all-flush" approach.

In order for the "selective-flush" approach to be practical, the PCM should have the ability to identify quickly the subjects or the objects affected by the policy change. However, it may not be trivial due to the dynamics of modern access control systems. We have developed identification algorithms for policy based on Bell-LaPadula (BLP) model [2], and explore this issue for other access control models in the future research.

#### 3.6.4 Support for Time-sensitive Changes

*Time-sensetive* changes in authorization policies are less urgent than critical ones but still need to be propagated within a known period of time. For example, an employee is assigned to a new project or receives a job promotion. When an administrator makes a time-sensetive change, it is the PCM that computes the time period $t$ in which caches of all SDPs are guaranteed to become consistent with the change. As a result, even though the PDP starts making authorization decisions using the modified policy, the change becomes in effect throughout the CSAR deployment only after time period $t$. Notice that this does not necessarily mean that the change itself will be reflected in the SDPs' caches by then; only that the caches will use no responses invalidated by the change.

CSAR employs time-to-live (TTL), similar to the one in the Internet's domain naming system (DNS) [20], to process time-sensitive changes. Every primary response is assigned a TTL which determines how long the response remains valid in the cache, such as one day or one hour. The assignment can be performed by either the SDP, the PDP itself, or a proxy, through which all responses from the PDP pass before arriving to the SDPs. The choice depends on the deployment environment and the backward compatibility requirements. Every SDP periodically purges from its cache those responses whose TTL elapses.

TTL value can also vary from response to response. Some responses (say, authorizing access to more valuable resources) can be assigned a smaller TTL than others. For example, for a BLP-based policy, the TTL for the responses concerning *top-secret* objects could be shorter than for *confidential* objects.

#### 3.6.5 Support for Time-insensitive Changes

When the administrator makes a *time-insensitive* change, the system guarantees that all SDPS will eventually become consistent with the change. No promises are given, however, about how long it will take. Support for time-insensitive changes is necessary because some systems may not afford the cost of, or just are not willing to support, critical or time-sensetive changes.

One approach to supporting time-insensitive change is by flushing SDP's cache, either passively or actively. The passive approach is achieved when the application server reboots for maintenance. For the active approach, each SDP can flush responses older than a pre-configured age, which is the same as the time-sensitive approach with each SDP assigning TTL to the arriving responses.

## 4 Evaluation

In evaluating CSAR, we wanted to first demonstrate that our design works, and second to understand how much gain in availability and performance can be achieved as well as how these characteristics depend on the the number of cooperating SDPs and the frequency of policy changes.

We used both simulation and a prototype implementation. The simulation enables us to study availability by hiding the complexity of underlying communication, while the prototype enables us to study both performance and availability in more dynamic and realistic environment.

Both simulation and implementation used a similar setup. Without further notices, the PDP made decisions based on a BLP [2] policy that contained four security levels and three categories. The policy was enforced by all the PEPs. Each SDP instance contained 100 subjects and 100 objects, and implemented same inference algorithm.

### 4.1 Simulation-based Evaluation

We used simulation to evaluate the benefits of cooperation on system availability and on reducing load at the PDP. We used the cache hit (when an authorization request is resolved by the SDPs without contacting the PDP) rate as an indirect metric for these two characteristics. A high cache hit rate results in masking transient PDP failures (thus improving the availability of the access control system) and reducing the load on the PDP (thus effectively improving the scalability of the system).

CSAR involves multiple SDPs. In the experiments, we inspected one of the SDPs and explored the influence of the following three factors on its hit rate: (a) the number of cooperating SDPs, (b) cache warmness at each SDP, which is defined as the ratio of cached request-response pairs to the total possible request-response pairs, (c) the overlap rate between the resource spaces of the cooperating SDPs, which

is defined as the ratio of the objects owned by two cooperating SDPs to the inspected SDP. The overlap rate provides a unique measure of similarity between the resource of two SDPs.

A simulation engine was responsible for running the experiment and gathering the results. It read requests from the training set and testing set, and submitted each request to the PDP and the SDP. Each request was made up of a subject, object, and access right (*read* and *append*). The training set was a randomized list of every possible request in the request space, while the testing set was a random sampling of requests.

The simulation engine operated in two different modes: *warming* and *testing*. In the cache warming mode, the engine submitted requests from the training set to the PDP. The engine used the responses from the PDP to update the SDPs, warming their caches to a specified level, the percentage of authorizations cached. Once a desired cache warmness was achieved, the engine switched to testing mode. The SDP caches were not updated in this mode which is only used to estimate cache hit rates. The engine submitted requests from the testing set to the SDPs, recorded their responses, and calculated the hit rate as the ratio of the resolved testing requests by SDPs to all testing requests. This process was repeated for different levels of cache warmness, from 0 to 100% with the increment of 5%.

Simulation results were gathered on a commodity PC with one 2.8 GHz Intel Pentium 4 processor and 1 GB of RAM. The simulation framework was written in Java and ran on the Sun's 1.5.0 JRE. In all experiments, we used same cache warmness for each SDP and same overlap rate between the inspected SDP and each other cooperative SDP.

#### 4.1.1 Results and Discussion

In the first experiment, we studied how the hit rate depends on the the cache warmness and the overlap rate. Figure 4(a) compares the hit rate for the case of one SDP, representing SAAM (bottom curve) with the hit rate achieved by cooperating SDPs. Here, five cooperating SDPs collectively resolve responses and have their resource spaces overlap at either 10%, 50%, or 100%.

Figure 4(a) indicates that, when cache warmness is low (around 10%), the hit rate is still larger than 50% for overlap rates of 50% and up. Especially, when the overlap rate is 100%, CSAR can achieve almost 70% hit rate at 10% cache warmness. Low cache warmness can be caused by the characteristics of the workload, by limited storage space, or by frequently-changed access control policies.

For 10% overlap rate, however, CSAR outperforms SAAM by a mere 10%, which might not warrant the cost of CSAR complexity.

In the second experiment, we studied the impact of the number of cooperating SDPs on the hit rate under various overlap rates. We varied the number of SDPs from 1 to 10 while maintaining 10% cache warmness at each SDP. Figure 4(b) presents the results for 10%, 50%, and 100% overlap rates.

As expected, increasing the number of SDPs leads to higher hit rates. At the same time, the results shown in Figure 4(c) indicate that additional SDPs provide diminishing returns. For instance, the first SDP brings a 15% improvement in the hit rate, while the 10th SDP contributes only 2%. One can thus limit the number of cooperating SDPs to control the overhead traffic without losing the major benefits of cooperation. The results also imply that in a large system with many SDPs the impact of a single SDP's failure on the overall hit rate is negligible.

### 4.2 Prototype-based Evaluation

This section describes our prototype design and the results of the experiment with the prototype. The prototype system consisted of PEPs, SDPs, the discovery service (DS), a PDP, and a test driver, all of which communicated with each other using Java RMI. Each PEP received randomly-generated requests from the test driver and called its local SDP for authorizations. Upon an authorization request from its PEP, each SDP attempted to resolve this request first locally, then by querying other SDPs, and, if nothing worked, by calling the PDP. Each SDP maintained a dynamic pool of worker threads that concurrently queried other SDPs. The PDP made access control decisions using a BLP-based policy stored on disk in an XML file. The DS used a hash map.

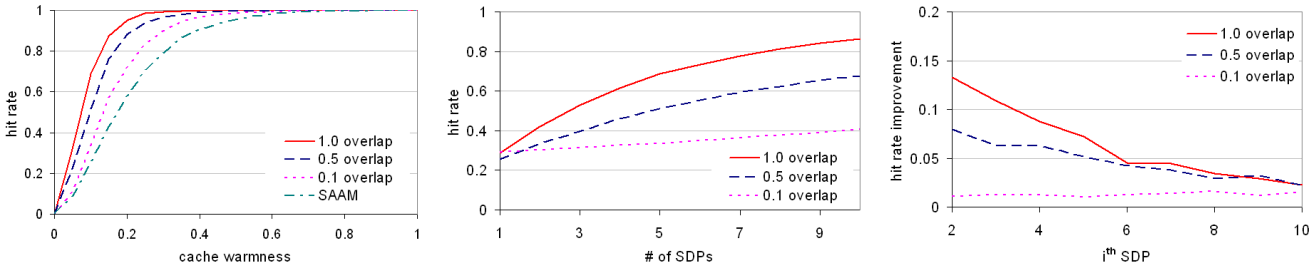
We implemented the PAP and the PCM according to the design described in Section 3.6. To simplify the prototype, the two components were process-collocated with the PDP.

When the PDP generated a primary response in our prototype, the PDP signed the response by computing a SHA1 digest of the response and signing the digest with its private key. When an SDP received the primary response, it verified the signature and stored the signature locally.

When an SDP made a secondary response for other SDPs, the SDP associated each primary responses inside the evidence list with its signature. When the remote SDP received the secondary response, it verified the signature of each primary response in the evidence list as well as proved the correctness of the evidence list.

#### 4.2.1 Experiments on Response Time

First we used the prototype to study the performance of CSAR in terms of client-perceived response time. We compared response times for four settings:




(a) Hit rate as a function of cache warmness for 5 SDPs compared to 1 SDP (bottom curve).

(b) Hit rate as a function of number of SDPs at cache warmness of 10%.

(c) The contribution of a new SDP to hit rate improvement at cache warmness of 10%.

**Figure 4. Impact of cache warmness, overlap rate, and the number of cooperating SDPs on hit rate.**

**No caching** SDPs were not deployed and PEPs sent authorization requests directly to the PDP.

**Non-cooperative caching (SAAM)** SDPs were deployed and available only to their local PEP. When a request was received, each SDP first tried to resolve the request locally. If no success, it sent the request to the PDP.

**Cooperative caching (CSAR)** In this scenario SDP cooperation was enabled. Requests were resolved sequentially for minimizing the load on the PDP, at the expense of the response time (see CSAR configurability in Section 3.3).

**CSAR with response verification** In this scenario response verification is enabled. Every primary response was signed by the PDP and every secondary response was verified by the SDP.

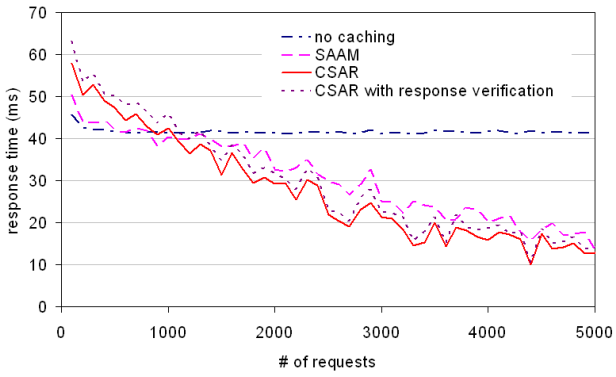



**Figure 5. Response time as a function of increasing cache warmness.**

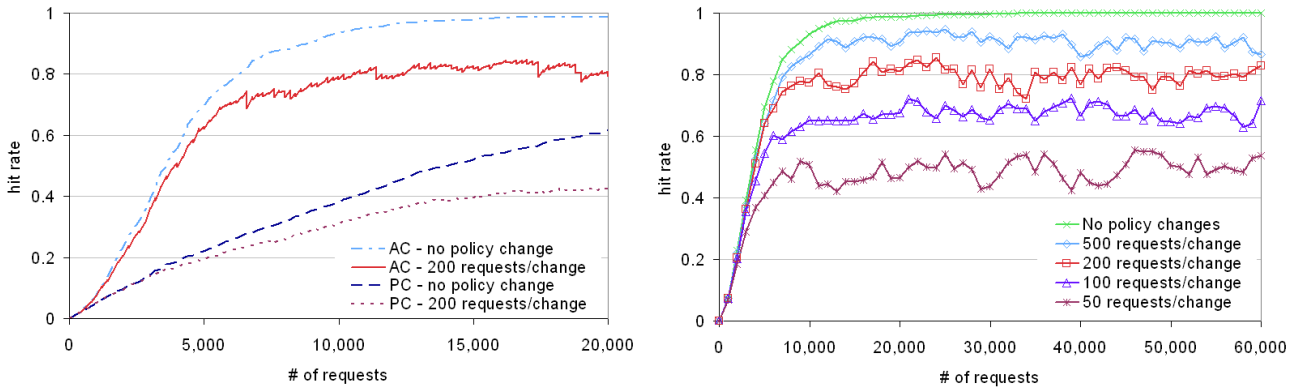
The experimental system consisted of four PEPs process collocated with their SDPs, a PDP and a DS. The overlap rates among SDPs' resources and subject populations were 100%. Each two PEPs and SDPs shared a commodity PC with 2.8 GHz Intel Pentium 4 processor and 1 GB of RAM. The DS and the PDP ran on one of the two machines, while the test driver ran on the other. The two machines were connected by a 100 Mbps LAN. As we aimed to evaluate CSAR specifically for scenarios when the PDP is deployed on a remote machine we introduced a 40ms network delay to each authorization request.

At the start of the experiment the caches were cold. The test driver maintained a thread for each PEP, simulating a client per PEP. Each thread sent requests to its PEP sequentially. The test driver recorded the response time for each request. After every 100 requests the test driver calculated the mean response time and used it as an indicator of the response time for this period.

We ran the experiment to answer the following question: when is the cooperation among SDPs most helpful for performance? Specifically, when cache warmness is low, each SDP does not have sufficient number of responses in its cache to resolve new requests on its own. When the cache warmness is high, each SDP can resolve most requests locally and therefore uses other SDPs or the PDP rarely. We chose to run the experiment when the cache size is still small, i.e. 5,000 requests for each SDP at the end.

Figure 5 plots the results. The following can be directly observed from the figure:

- For the "no caching" scenario, all the average response times are slightly larger than 40ms. This is because all requests have to be resolved by the PDP.
- When caching and approximate authorizations are enabled, response times decrease consistently with the number of requests because more requests are resolved locally.
- When cooperation is enabled, response times are further reduced after 1000 requests.
- When response verification is enabled, its impact to




(a) Hit rate drops with every policy change

(b) Hit rate as a function of number of requests at various frequency of policy change.

**Figure 6. The impact of policy changes on hit rate with single SDP.**

response times is small: the response time is increased less than 5%.

The result shows that while overall cooperation does not bring significant benefits in terms of improving response time, it does however improve average response time everywhere except cold or close to cold caches. The improvement is due to the requests resolved by the cooperating SDPs without going to the PDP. Particularly, we observe up to 30% improvement when the cache size is between between 2,500 requests and 3,000 requests.

Note that for the cooperation enabled scenario we used a sequential authorization process: the SDP first tried to resolve a request locally, then, if no success, contacted other SDPs, and only then the PDP. This has the advantage of maintaining the load on the PDP. If the PDP has enough resources to support higher loads, concurrent authorization process can be employed to further reduce the response time in the cooperative scenario.

#### 4.2.2 Experiments on Policy Changes

To evaluate the design of the cache consistency mechanisms, we studied their behavior in the presence of policy changes. Since the hit rate depends on the warmness of the SDPs' caches, and a policy change may result in flushing one or more responses from cache before they expire, we expected that continual policy changes at a constant rate would unavoidably result in a reduced hit rate. We wanted to understand by how much.

In order to measure the hit rate at run-time, each request sent by the test driver was associated with one of the two modes: *warming* and *testing*, used for warming the SDP caches or testing the cumulative hit rate respectively. The test driver switched between these two modes at predefined intervals. The overlap rates among SDPs' resources and subject populations were 100%.

The test driver maintained a separate thread responsible for firing a policy change and sending the policy change message to the PDP at pre-defined intervals, e.g., after every 100 requests.

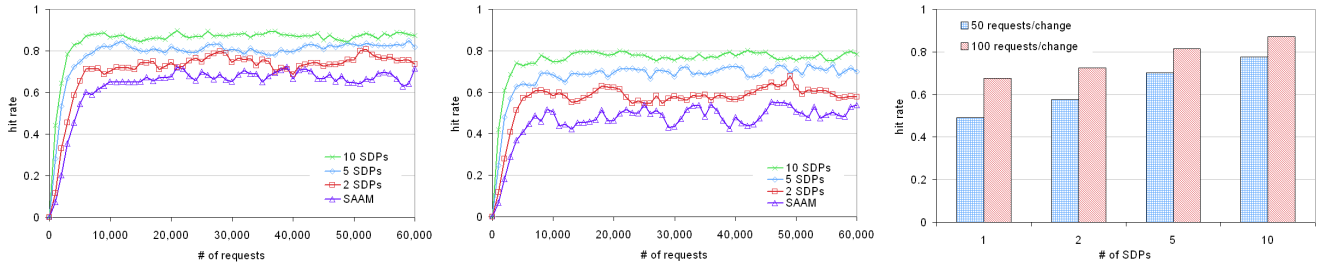
We first studied how the hit rate was affected by an individual policy change, i.e., the change of the security label for a single subject or object. We expected that SAAM inference algorithms were sufficiently robust and thus an individual change would result only in minor degradation of the hit rate. We used just one SDP for this experiment.

The test driver sent 20,000 requests in total. A randomly generated policy change message was sent to the PDP every 200 requests and the hit rate was measured right before and after each policy change, as well as at regular intervals. Results were gathered on a commodity PC with one 2.8 GHz Intel Pentium 4 processor and 1 GB of RAM.

Figure 6(a) presents how the hit rate drops with every policy change. We measured the hit rate for both the approximate recycling (the top two curves) and the precise recycling (the bottom two curves) of authorizations by SDP. For both of them, the figure shows the hit rate as a function of the number of observed requests, with or without policy changes. Because the hit rate was measured right before and after each policy change, every kink in the curve means a hit rate drop caused by a policy change.

Figure 6(a) demonstrates that the hit rate drop is small for both the approximate component and the precise component. For the approximate component, the largest hit rate drop is 5%, and most of other drops are around 1%. After the hit rate drops, the curve climbs up again because the cache size is increased with new requests.

It is also interesting to note that the curve for the approximate recycling with policy change is more ragged com-




(a) Hit rate as a function of number of requests observed with various numbers of cooperating SDPs at 100 requests/change.

(b) Hit rate as a function of number of requests observed with various numbers of cooperating SDPs at 50 requests/change.

(c) The average stabilized hit rate

**Figure 7. The impact of SDP cooperation on hit rate with policy changes.**

pared with the precise recycling. This suggests, not surprisingly, that the approximate recycling is more sensitive to the policy change. The reason is that the approximate recycling employs SAAM inference algorithm based on directed acyclic graph. A policy change could partition the graph resulting in larger reduction in the hit rate.

Although the hit rate drop is small, we can see that the cumulative effect of the policy change could be large. As Figure 6(a) shows, the hit rate of approximate recycling decreases about 12% in total when the request number reaches 20,000. This result leads to another interesting question: will the hit rate finally stabilize to some point or will it continuously drop?

To answer this question, we ran another experiment to study how the hit rate changes with continuous policy changes during a longer term. We used a larger number of requests (60,000), and measured the hit rate after every 1,000 requests. We varied the frequency of policy changes from 50 requests/change to 500 requests/change.

Figure 6(b) shows the hit rates as functions of the number of observed requests with each curve corresponding to a different frequency of random critical policy changes. Because of the continuous policy change, we do not see perfect asymptote of curves. However, the curves indicate that the hit rates stabilize after 20,000 requests. We calculated the averages of the hit rates after 20,000 requests and used them to represent the eventual stabilized hit rate. As we expected, the more frequent the policy changes the lower the stabilized hit rates are. The reason is that the responses are removed from SDPs' caches more frequently.

Figure 6(b) also shows that each curve has a knee. The steep increase in the hit rate below the knee implies that increased requests improve the hit rate dramatically in this period. Once the number of requests passes the knee, the benefit brought by caching further requests reaches the plateau of diminishing returns.

We finally studied how the cooperation between SDPs could benefit the hit rate under continuous policy changes. In the experiments, we varied the number of SDPs from 1 SDP to 10 SDPs.

Figures 7(a) and 7(b) show hit rates vs. number of requests observed when the policy changes at the rate of 50 requests/change and 100 requests/change. Figure 7(c) compares the average stabilized hit rate for the two frequencies of policy changes. As we expected, cooperation between SDPs improves the hit rate.

It is interesting to note that, when the number of SDPs increases, the curves after the knee become more smooth. This is a direct reflection of the impact of cooperation on the hit rate: the cooperation between SDPs compensate the hit rate drop caused by the policy changes at each SDP.

## 5 Related Work

CSAR is related to several research areas, including authorization caching, collaborative security, and cooperative caching. In this section, we review the work in each field and compare them with CSAR.

To improve the performance and availability of access control systems, caching authorization decisions has been employed in a number of commercial systems [17, 10, 21] as well as several academic distributed access control systems [1, 3, 15]. None of these systems involve cooperation between cache servers and most of them adopt the solution similar to TTL for cache consistency.

With SAAM, Crampton et al. [8] extend this caching mechanism, dubbed "precise caching", by introducing SDP and adding inference of "approximate" authorizations. CSAR builds on SAAM and extends it by enabling applications to share authorization responses. To the best of our knowledge, no previous research has proposed such cooperative recycling of authorizations.

A number of research projects propose cooperative access control frameworks that involve multiple, cooperative PDPs that resolve authorization requests. In Stowe's scheme [27], a PDP that receives an authorization request

from PEP forwards the request to other collaborating PDPs and combines their responses later. Each PDP maintains a list of other trusted PDPs to which it forwards the request. Mazzuca [19] extends Stowe's scheme. Besides issuing requests to other PDPs, each PDP can also retrieve policy from other PDPs and make decisions locally. These two schemes both assume that each PDP maintain different policies and a request need to be authorized by different parties. CSAR, on the other hand, focuses on the collaboration of PEPs and assumes that they enforce the same policy. This is why we consider Stowe's and Mazzuca's schemes orthogonal to ours.

Our research can be considered a particular case of a more general research direction, known as *collaborative security*. It aims at improving security of a large distributed system through the collaboration of its components. A representative example of collaborative security is Vigilante [7], which enables collaborative worm detection at end hosts, but does not require hosts to trust each other. Another example is application communities [18], in which members collaborate to identify previously unknown flaws and attacks and notify other members. Our research can also be viewed as a collaborative security mechanism because different SDPs collaborate with each other to resolve authorization requests to mask PDP failures or slow performance.

Another related research area, albeit outside of the security domain, is cooperative web caching. Web caching is a widely used technique for reducing the latency observed by Web browsers, decreasing the aggregate bandwidth consumption of an organization network, and reducing the load on web servers. Several teams [6, 16, 11] investigated decentralized, cooperative web caching. Our approach is different in the following aspects: first, an authorization usually can not directly be shared between different users if not for the inference; second, approximate authorizations can not be pre-cached and our discovery service needs to take this into account.

## 6 Summary

As distributed systems scale up and become increasingly complex their access control infrastructures are facing new challenges. Conventional request-response authorization architectures become fragile and scale poorly to massive-scale. Caching authorization decisions has long been used to improve access control infrastructure availability and performance. In this paper, we build on this idea and on the idea of inferring approximate authorization decisions at intermediary control points and propose a cooperative approach to further improve the availability and performance of access control solutions. Our cooperative secondary authorization recycling (CSAR) exploits the potential of an increased hit rate offered by a larger, distributed cooperative cache of access control decisions. We believe that this solution is especially practical in massive-scale distributed systems, such as Grid, because of the overlap in their user and resource spaces and the need for consistent policy enforcement across multiple parties.

We define CSAR system requirements and present a detailed design that meets these requirements. We introduce a response verification mechanism that does not require cooperating SDPs to trust each other. Responses are verified by tracing back to a trusted primary source, the PDP. We manage consistency by dividing all the policy changes into three categories and employing efficient consistency techniques for each type.

We evaluate CSAR through both simulations and a prototype implementation. Our experiments show that this approach improves the availability and performance of the access control sub-system. Specifically, by recycling secondary authorizations between SDPs, the hit rate can reach 70% even when only 10% of all possible authorization decisions are cached at each SDP. This high hit rate results in more requests being resolved by the local and other cooperating SDPs, even when the authorization server is unavailable or slow, thus increasing availability of the authorization infrastructure and reducing the load of the authorization server. This results imply that even with small caches (or low cache warmness) our cooperative authorization solution can offer significant benefits. In addition, request processing time is improved up to 30% in our experimentation setup.

## References


[1] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, Oakland, CA, 2005.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-74-244, MITRE, March 1973.

[3] K. Borders, X. Zhao, and A. Prakash. CPOL: high-performance policy evaluation. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 147–157, New York, NY, USA, 2005. ACM Press.

[4] E. A. Brewer. Towards robust distributed systems. In *(Invited Talk) Principles of Distributed Computing*, Portland, Oregon, 2000.

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation*, 11 2006.

[6] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 153–163, Jan 1996.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, Brighton, UK, 2005.

[8] J. Crampton, W. Leung, and K. Beznosov. Secondary and approximate authorizations model and its application to Bell-LaPadula policies. In *Proceedings of the Symposium on Access Control Models and Technologies (SACMAT)*, pages 111–120, Lake Tahoe, California, USA, June 7–9 2006. ACM, ACM Press.

[9] L. G. DeMichiel, L. Ü. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, 2001.

[10] Entrust. getaccess design and administration guide. Technical report, Entrust, September 20 1999.

[11] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy Squid. In *Proceedings of the 1998 Workshop on Internet Server Performance*, pages 129–136, June 1998.

[12] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[13] B. W. Johnson. *Fault-tolerant Computer System Design*, chapter An introduction to the design and analysis of fault-tolerant systems, pages 1–87. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[14] Z. Kalbarczyk, R. K. Lyer, and L. Wang. Application fault tolerance with Armor middleware. *IEEE Internet Computing*, 9(2):28–38, 2005.

[15] M. Kaminsky, G. Savvides, D. Mazieres, and M. F. Kaashoek. Decentralized user authentication in a global file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 60–73, New York, NY, USA, 2003. ACM Press.

[16] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the of 8th International World–Wide Web Conference*, Toronto, Canada, May 1999.

[17] G. Karjoth. Access control with IBM Tivoli Access Manager. *ACM Transactions on Information and Systems Security*, 6(2):232–57, 2003.

[18] M. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *Proceedings of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (NDSS 2006)*, pages 95–106, San Diego, CA, 2006.

[19] P. J. Mazzuca. Access control in a distributed decentralized network: an XML approach to network security using XACML and SAML. Technical report, Dartmouth College, Computer Science, Spring 2004.

[20] P. Mockapetris. Domain names - concepts and facilities, 1987.

[21] Netegrity. Siteminder concepts guide. Technical report, Netegrity, 2000.

[22] V. Nicomette and Y. Deswarte. An authorization scheme for distributed object systems. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 21–30, Oakland, CA, 1997.

[23] OMG. CORBAservices: Common object services specification, security service specification v1.8, 2002.

[24] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. In *POLICY '02: Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 50, Washington, DC, USA, 2002. IEEE Computer Society.

[25] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[26] Securant. Unified access management: A model for integrated web security. Technical report, Securant Technologies, June 25 1999.

[27] G. H. Stowe. A secure network node approach to the policy decision point in distributed access controlw. Technical report, Dartmouth College, Computer Science, June 2004.

[28] W. Vogels. How wrong can you be? Getting lost on the road to massive scalability. In *Middleware Conference*, Toronto, October 20 2004.

[29] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 48, Washington, DC, USA, 2003. IEEE Computer Society.

[30] XACML-TC. Oasis extensible access control markup language (xacml) version 1.0. OASIS Standard, 18 February 2003.