Cooperative Secondary Authorization Recycling

Qiang Wei, Matei Ripeanu, and Konstantin Beznosov
Dept. of Electrical and Computer Engineering, University of British Columbia
Vancouver, BC, Canada
qiangw@ece.ubc.ca, matei@ece.ubc.ca, 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 challenges in terms of fragility and poor scalability. We propose an approach where each application server caches previously received authorizations at its secondary decision point and shares them with other application servers to mask authorization server failures and network delays.

This paper presents the design of our cooperative secondary authorization recycling system and its evaluation using simulation and prototype implementation. The results demonstrate that our approach improves the availability of authorization infrastructures while preserving their performance characteristics. Specifically, by sharing authorizations, the cache hit rate—an indirect metric of availability—can reach 70%, even when only 10% of authorizations are cached. Depending on the deployment scenario, the performance in terms of the average time for authorizing an application request can be reduced by up to 30%.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; C.4 [**Computer-Communication Networks**]: Performance of Systems—*Reliability, availability, and serviceability*

General Terms

Security, Design, Performance, Reliability

Keywords

CSAR, SAAM, authorization recycling, cooperation

1. INTRODUCTION

Architectures of modern access control solutions—such as [15, 10, 18, 22, 20, 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 with all centralized architectures, this architecture has two critical drawbacks: the PDP is a single point of failure (SPF) as well as a potential performance bottleneck.

The SPF aspect of PDP leads to reduced availability: the authorization server responsible for making authorizations 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 from a misconfiguration of the supporting infrastructure. A conventional approach to improving the availability of a distributed infrastructure is failure masking through redundancy of either information, or time, or through physical redundancy [13]. However, redundancy and other general purpose fault-tolerance techniques for distributed systems scale poorly, and become technically and economically infeasible when the number of entities in the system reaches thousands [14, 24].

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 the large size of the resource and user populations. Thus, the centralized PDP often becomes a performance bottleneck [19]. Additionally, the communication delay between the PEP and the PDP—added to the inherent cost of computing an authorization—can make authorization overhead prohibitively high.

The state-of-the-practice approach to improving overall system availability and reducing processing delays observed by the client is to cache authorizations at each PEP—what we refer to as *authorization recycling*. Existing authorization solutions commonly provide PEP-side caching [15, 10, 18]. These solutions, however, only employ a simple form of authorization recycling: a cached authorization is reused only if the authorization request in question exactly matches the original request for which the authorization was made. We refer to such reuse as *precise recycling*.

To improve authorization system availability and reduce delay, Crampton et al. [8], introduced 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 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 im-

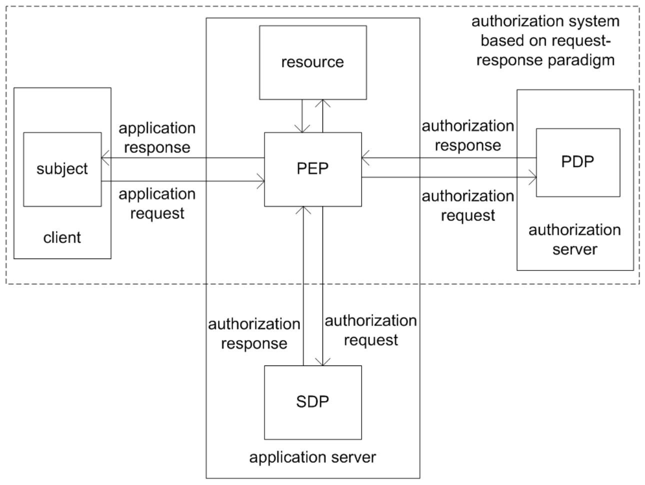


Figure 1: SAAM adds SDP to authorization systems based on request-response paradigm.

proves the availability and performance of the access control subsystem, 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 cached authorizations 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. We aim to further improve the resilience of the authorization infrastructure to network and authorization server failures, and to minimize the delay in producing authorizations.

We believe that our approach is especially applicable to the distributed systems involving either cooperating parties, such as Grid systems, or replicated services, such as load-balanced clusters. Cooperating parties or replicated services usually have similar users and resources, and use centralized authorization servers (e.g., in Grid [26]) to enforce consistent access control policies. Therefore, authorizations can often be shared among them, bringing benefits to each other.

This paper makes the following contributions:

- We propose the concept of cooperative secondary authorization recycling (CSAR), analyze its design requirements, and propose a concrete architecture.
- We use simulations and a prototype to demonstrate CSAR feasibility and evaluate its benefits. Evaluation results show that by combining the cooperation and inference, our approach improves the availability of authorization infrastructures while preserving their performance. Specifically, the overall cache hit rate can reach 70%, even with only 10% of authorizations 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 the availability of authorization infrastructures and reducing the load of the authorization server. Additionally, our experiments show that request processing time can be improved by up to 30%.

The rest of this paper is organized as follows. Section 2 presents the SAAM definitions and algorithms. 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 our work in Section 6.

2. BACKGROUND

This section briefly describes the SAAM definitions and the algorithms for the Bell-LaPadula (BLP) access control model. More detail can be found in [8].

SAAM formally defines an authorization request and authorization response. An authorization request is a tuple (s, o, a, c, i), where s is the subject, o is the object, a is the access right, c is the request contextual information, and i is the request identifier. Two requests are *equivalent* if they only differ in their identifiers. An authorization response to 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 the decision, and E is the evidence. The evidence is a list of response identifiers that were used for computing a response, and can be used to prove the correctness of the 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 to the request in question or a response to 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 shorthand for "authorization request" and "response" as shorthand for "authorization response".

The inference algorithm for approximate responses depends on the access control model. For example, the BLP model [2] specifies how information can flow within the system based on labels attached to each subject and object. Let λ be the security function mapping an object or subject to its security label. A subject s can read an object o if $\lambda(s) \geqslant \lambda(o)$; a subject s can write to an object o if $\lambda(s) \leqslant \lambda(o)$. The SAAM$_{\text{BLP}}$ inference algorithm uses the responses to past requests 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 be inferred that $\lambda(s_1) > \lambda(o_1) > \lambda(s_2) > \lambda(o_2)$. Therefore, a request $(s_1, o_2, read, c_4, i_4)$ should also be allowed, and the corresponding response is $(r_4, i_4, [i_1, i_2, i_3], allow)$.

Crampton et al. [8] present simulation results that demonstrate the effectiveness of this approach. With only 10% of authorizations cached, the SDP can resolve over 30% more authorization requests than a conventional PEP with caching. In the rest of the paper we present the CSAR design, as well as evaluation results that suggest cooperation among SDPs is possible and can further improve access control system availability.

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 of access control infrastructures while controlling their overhead, by sharing authorization information among cooperative SDPs. Each SDP resolves the requests from its own PEP by locally making secondary authorization decisions, by involving other cooperative SDPs in the authorization process, or by passing the request to the remote PDP.

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

Low overhead. As each SDP participates in making authorizations for some non-local requests, its load is increased. The design should therefore minimize this additional overhead.

Trustworthiness. 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 [5] conjectures and Lynch et al. [12] prove that distributed systems cannot simultaneously provide the following three properties: availability, consistency, and partition tolerance. We believe that availability and partition tolerance are essential properties that 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 a set of latency-sensitive applications may require requests are resolved in minimal time. A deployment with applications generating a high volume of authorization requests, on the other hand, 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 so that minimal changes are required to existing infrastructure—i.e., PEPs and PDP—in order to switch to CSAR.

3.2 Adversary Model

The CSAR system involves multiple components: the PDP, PEPs, and SDPs. The PDP is the ultimate authority for access control decisions. We assume that the PDP cannot be compromised. We further assume that each PEP trusts decisions received from its own SDP. However, 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 in addressing 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 the PEP and SDP are either part of the application or of 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 SDPs.

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

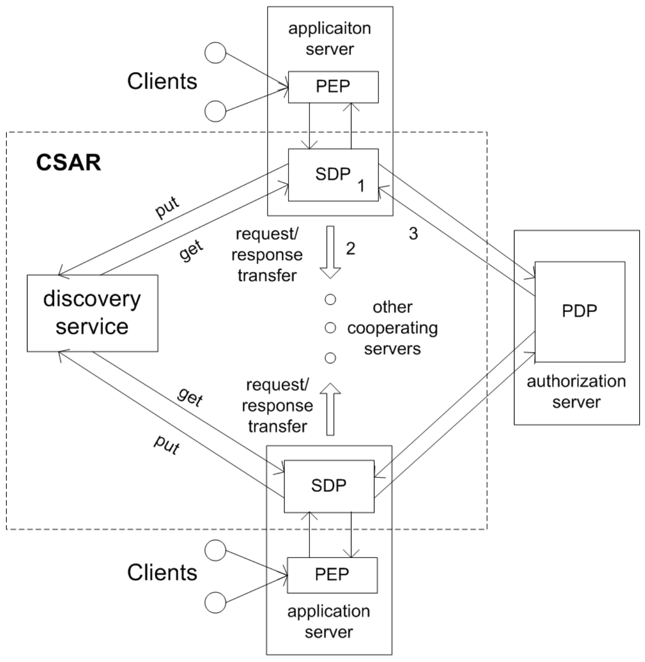


Figure 2: CSAR introduces cooperation between SDPs.

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 to reduce latency, then the above three steps can be performed concurrently, and the SDP will use the first response received. If the objective is to reduce 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 to 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 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 enabling cooperative SDPs to share their authorizations is the discovery service (DS), which helps an SDP find other SDPs that might be able to resolve a request.

A naive approach to implementing 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. However, it has two important drawbacks. First, it inevitably increases the load on all SDPs. Second, it causes high traffic overhead when SDPs are geographically distributed.

To address these two drawbacks, an SDP in CSAR selectively distributes requests 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 infers an approximate response. For this purpose, both the subject and object of the request have to be present in 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 an interface with the following two functions: $put(entity, SDPaddress)$ and $get(entities)$, where $entities$ is a subject and object pair. Given an entity and the address of an SDP, the put function stores the mapping $(entity, SDPaddress)$. A put operation can be interpreted as "this SDP knows something about the entity." Given a subject and object pair, the get function returns a list of SDP addresses that are mapped to both the subject and object. The results returned by the get operation can be interpreted as "these SDPs know something about both the subject and object and thus might be able to resolve the request involving them".

Using DS avoids broadcasting requests to all SDPs. Whenever an SDP receives a primary response to a request, it calls the put function to register itself in the DS as a suitable SDP for both the subject and object of the request. When cooperation is required, the SDP calls the get function to retrieve from the DS a set of addresses of those SDPs that might be able to resolve the 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 [21].

We do not assume that the DS is trusted by any component of the system. Our design limits a malicious DS to being able 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, denying all requests and 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 explained 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. To verify approximate responses, each SDP uses the evidence part of the 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. Second, the SDP uses the knowledge of both the inference algorithm and 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 evidence list. Based on the administration policy and deployment environment, the verification process can be configured differently to achieve various trade-offs between security and performance.

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.

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), which is consistent with the standard XACML architecture [27]. In addition, we assume that the PAP interface provides security administrators with the option to specify the criticality of the ongoing change. We elaborate on 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 or network traffic. To address this requirement, we divide all the policy changes into three categories: critical changes, time-sensitive changes, and time-insensitive changes. This 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 are 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 cannot modify the PDP to support cache consistency. To address this requirement, we add a 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.

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 authorization policy that, if not fixed immediately, will result in leaking customers' personal information to the public and possibly causing substantial damage to the company's reputation, must be handled immediately. Therefore, an immediate correction in the authorization policy is critical.

When an administrator makes a critical change in the authorization policy, our approach requires that he or 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 usually assumed to be just a few minutes. To accommodate variations in requirements for efficiency, we devised two approaches to updating SDPs with critical changes: all-flush and selective-flush.

In the **all-flush** approach, the PCM broadcasts every policy update message to all the SDPs. Upon receiving the message, each SDP flushes its cache and reports the success or failure of flushing to the PCM. 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 unsuccessful 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 for a period t if some SDPs are unavailable due to network partitions. To avoid waiting, the PCM can run a failure detection service like Google's Chubby [6]. 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 effect 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 cannot invalidate any cached response. Therefore, 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. Finally, not all cached responses get invalidated by every policy change. Using our 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.

Considering the above observations, we developed a more efficient approach to propagating critical policy changes, called **selective-flush**, since only some SDPs are updated and only selected cache entries are flushed. Compared to the all-flush approach, the selective-flush approach has the benefits of reducing server overhead and network traffic while minimizing the impact on system availability. We sketch out 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 DS, it multicasts the policy change message to these affected SDPs.

In the case where the DS 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 results to the PCM. 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 quickly identify the subjects or objects affected by the policy change. However, this procedure may not be trivial due to the dynamics of modern access control systems. We have developed identification algorithms for the policies based on the BLP model, and will explore this issue for other access control models in future research.

3.6.4 Support for Time-sensitive Changes

Time-sensitive 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-sensitive 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 not use responses invalidated by the change.

CSAR employs time-to-live (TTL) to process time-sensitive changes. Every primary response is assigned a TTL that 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 backward compatibility requirements. Every SDP periodically purges from its cache those responses whose TTL elapses.

The 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.

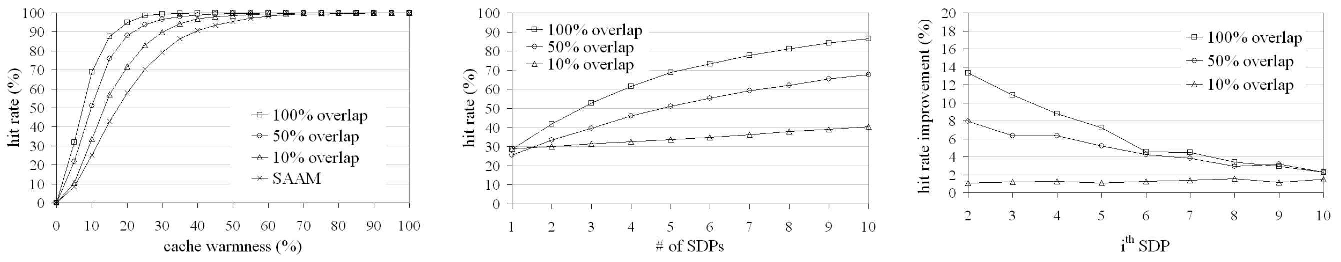
3.6.5 Support for Time-insensitive Changes

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 be able to afford the cost of, or are just not willing to support, critical or time-sensitive changes.

One approach to supporting time-insensitive change is by flushing the SDP's cache, either passively or actively. The passive approach is achieved when the application server reboots for maintenance. In the active approach, each SDP can flush responses older than a pre-determined 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 first to demonstrate that our design works. Then we sought to estimate the achievable gains in terms of availability and performance, and determine how these characteristics depend on factors such as the number of cooperating SDPs and the frequency of policy changes.



(a) Hit rate as a function of cache warmness for 5 SDPs compared to 1 SDP (i.e., SAAM).

(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 3: Impact of cache warmness, overlap rate, and the number of cooperating SDPs on hit rate.

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

We used a similar setup for both the simulation and implementation experiments. The PDP made access control decisions using a BLP-based policy in an XML file stored on disk. The policy was enforced by all the PEPs. Each SDP instance contained 100 subjects and 100 objects, and implemented the same inference algorithm. While the subjects were the same for each SDP, the objects could be different in order to simulate the resource overlap.

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 rate as an indirect metric for these two characteristics. A request resolved without contacting the PDP was considered a cache hit. 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) the *cache warmness* at each SDP (which is defined as the ratio of cached request-response pairs to the total possible request-response pairs); and (c) the *overlap rate* between the resource spaces of two cooperating SDPs (which is defined as the ratio of the objects owned by both SDPs to the objects owned only by the inspected SDP). The overlap rate provides a unique measure of similarity between the resources of two cooperating 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 SDPs. 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 testing requests resolved by SDPs to all testing requests. This process was repeated for different levels of cache warmness, from 0 to 100%, with an 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 Sun's 1.5.0 JRE. In all of the experiments, we used the same cache warmness for each SDP and the same overlap rate between the inspected SDP and every other cooperative SDP.

4.1.1 Results and Discussion

In the first experiment, we studied how the hit rate depends on cache warmness and overlap rate. Figure 3(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 3(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. In particular, when the overlap rate is 100%, CSAR can achieve a hit rate of almost 70% 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 a 10% overlap rate, however, CSAR outperforms SAAM by a mere 10%, which might not warrant the cost of CSAR's 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 3(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 3(c) indicate that additional SDPs provide diminishing returns. For instance, the first SDP brings a 14% 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 suggest 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 our experiments with the prototype. The prototype system consisted of the implementations of PEPs, SDPs, a DS, a PDP, and a test driver,

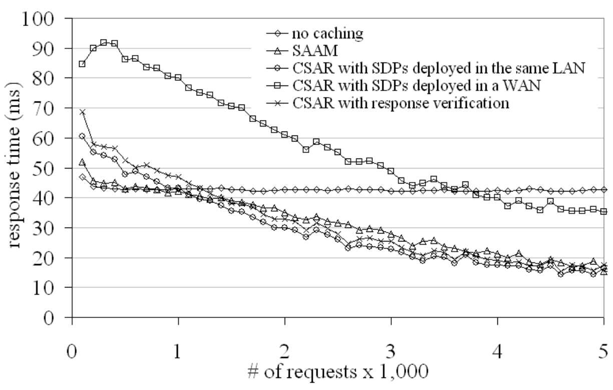


Figure 4: Response time as a function of number of requests observed by SDPs.

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 the DS and 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 DS used a customized hash map which supported assigning multiple values (SDP addresses) to a single key (entity).

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. Additionally, we implemented the selective-flush approach for propagation of policy changes.

To support response verification, we generated a 512-bit RSA key pair for the PDP. Each SDP maintained a copy of the PDP's public key. After the PDP generated a primary response, it signed the response by computing a SHA1 digest of the response and signing the digest with its private key.

4.2.1 Evaluating 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 the following five settings:

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

2. 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 this step was unsuccessful, the request was sent to the PDP.

3. CSAR with SDPs deployed in the same LAN. In this scenario, cooperation was enabled and SDPs were deployed in the same LAN. Requests were resolved sequentially for minimizing the load on the PDP, at the expense of response time (see CSAR configurability in Section 3.3).

4. CSAR with SDPs deployed in a WAN. This scenario was the same as the previous one except that SDPs were deployed in a WAN. In the experiment, we simulated this scenario by introducing a 40ms round-trip delay to each authorization request sent between SDPs.

5. CSAR with response verification. This scenario was the same as the third scenario except that response verification was enabled. Every primary response was signed by the PDP and every secondary response was verified by the SDP in terms of both response integrity and correctness of inference process.

The experimental system consisted of four PEP processes collocated with their SDPs, a PDP, and a DS. The overlap rate used among the SDPs' resource spaces was 100%. Each two collocated 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. We introduced a 40ms delay to each authorization request sent to the PDP in order to simulate the delay caused by network delays between application servers and the PDP, and the computational delay at the PDP necessary to compute authorizations with complex authorization polices.

At the start of each experiment, the caches were cold. The test driver maintained a thread for each PEP, simulating one 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 a 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 rarely uses other SDPs or the PDP. 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 4 shows the plotted results. The following can be directly observed from the figure:

1. For the "no caching" scenario, all of the average response times are slightly higher than 40ms. This is because all requests have to be resolved by the PDP.

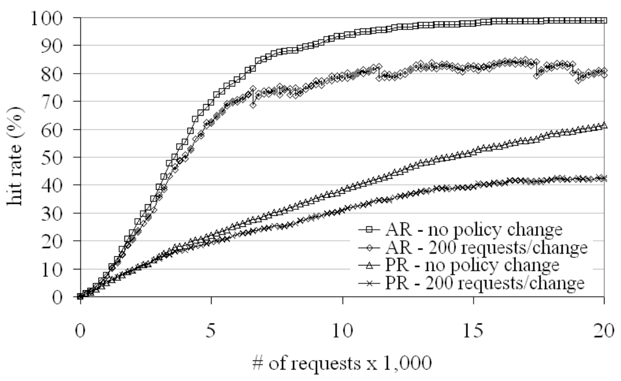
2. When caching and approximate authorizations are enabled, response times decrease consistently with the number of requests because more requests are resolved locally.

3. When cooperation is enabled for SDPs deployed in the same LAN, response times are further reduced after 1,000 requests.

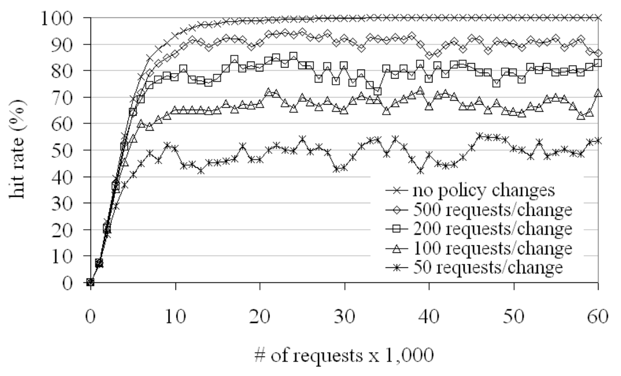
4. When cooperation is enabled for SDPs deployed in a WAN, response times are almost double compared to the previous scenario.

5. When response verification is enabled, its impact on response times is small: response time is increased less than 5%. Although we only show the results for SDPs deployed in the same LAN, this also applies to the scenario when SDPs are deployed in a WAN.

The results for the third scenario indicate that while overall cooperation does not bring significant benefits in terms of improving response time, it does improve average response time everywhere except with 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 between 2,000 requests and 3,000 requests have been processed. On the other hand, when SDPs are distributed over a WAN, the performance gain by recycling is lost due to the latency caused by cache misses.



(a) Hit-rate drops with every policy change for both approximate recycling (AR) and precise recycling (PR).



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

Figure 5: The impact of policy changes on hit rate with a single SDP.

Note that for the cooperation-enabled scenarios we used a sequential authorization process: the SDP first tried to resolve a request locally, then, if unsuccessful, contacted other SDPs, and only then the PDP. This procedure has the advantage of reducing the load on the PDP. If the PDP has enough resources to support higher loads, a concurrent authorization process can be employed to further reduce the response time in all cooperative scenarios.

4.2.2 Evaluating Effects of Policy Changes

To evaluate the design of the cache consistency mechanism, 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 caches 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.

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 rate used was 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 that an individual change would result in only 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 just before and after each policy change.

Figure 5(a) shows how the hit rate drops with every policy change. We measured the hit rate for both approximate recycling (the top two curves) and precise recycling of authorizations by the SDP. For both types of recycling, the figure shows the hit rate as a function of the number of observed requests, with policy change (lower curve) or without policy changes (upper curve). Because the hit rate was measured just before and after each policy change, every kink in the curve indicates a hit-rate drop caused by a policy change.

Figure 5(a) indicates that the hit-rate drops are small for both the approximate component and precise component. For the approximate component, the largest hit-rate drop is 5%, and most of other drops are around 1%. After each drop, 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 than it is for precise recycling. This result suggests, not surprisingly, that approximate recycling is more sensitive to the policy change. The reason is that approximate recycling employs a SAAM inference algorithm based on a directed acyclic graph. A policy change could partition the graph, resulting in a larger reduction in the hit rate.

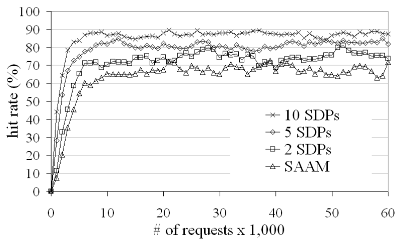
Although the hit-rate drop for each policy change is small, we can see that the cumulative effect of policy changes could be large. As Figure 5(a) shows, the hit rate of approximate recycling decreases about 20% in total when the request number reaches 20,000. This result leads to another interesting question: Will the hit rate finally stabilize at some point or will it continue to drop?

To answer this question, we ran another experiment to study how the hit rate varies with continuous policy changes over 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 to 500 requests per change.

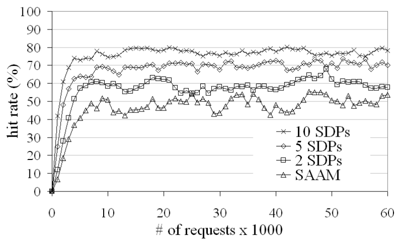
Figure 5(b) shows the hit rates as functions of the number of observed requests, with each curve corresponding to a different frequency of random policy changes. Because of the continuous policy change, we do not see a perfect asymptote of curves. However, the curves indicate that the hit rates stabilize after 20,000 requests. We can thus calculate the averages of the hit rates after 20,000 requests and use them to represent the eventual stabilized hit rate. As we expected, the more frequent the policy changes, the lower the stabilized hit rates are, since the responses are removed from the SDP caches more frequently.

Figure 5(b) also shows that each curve has a knee. The steep increase in the hit rate before 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.

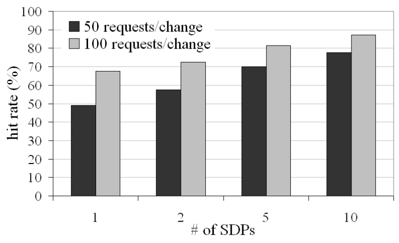
Finally, we studied how the cooperation between SDPs could benefit the hit rate under continuous policy changes. In the experiments, we varied the number of SDPs from 1 to 10. Figures 6(a) and 6(b) show hit rates versus the number of requests observed



(a) Hit rate as a function of number of requests observed when policy changes every 100 requests.



(b) Hit rate as a function of number of requests observed when policy changes every 50 requests.



(c) Comparison of stabilized hit rates.

Figure 6: The impact of SDP cooperation on hit rate when policy changes.

when the policy changes at the rate of 50 and 100 requests per change. Figure 6(c) compares the eventual 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 smoother. This trend is a direct reflection of the impact of cooperation on the hit rate: the cooperation between SDPs compensates for the hit-rate drops 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. This section reviews the work in each field and compare it to CSAR.

To improve the performance and availability of access control systems, caching authorization decisions has been employed in a number of commercial systems [15, 10, 18] as well as several academic distributed access control systems [1, 4]. None of these systems involves cooperation between cache servers, and most of them adopt the solution similar to TTL for cache consistency.

To further improve the performance and availability of access control systems, Beznosov [3] introduces the concept of recycling approximate authorizations, which extends the precise caching mechanism. Crampton et al. [8] develop SAAM 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 for authorizations.

A number of research projects propose cooperative access control frameworks that involve multiple, cooperative PDPs that resolve authorization requests. In Stowe’s scheme [23], 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 [17] 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 maintains different policies and that a request needs 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 to be orthogonal to ours.

Our research can be considered a particular case of a more general research direction, known as *collaborative security*. This research 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 [16], 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 and 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 projects have investigated decentralized, cooperative web caching [25]. Our approach is different from them in the following two aspects: first, unlike Web documents, an authorization usually cannot be directly shared among users; second, approximate authorizations cannot be pre-cached, which our discovery service needs to take 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 have built on this idea and on the idea of inferring approximate authorization decisions at intermediary control points, and have proposed a cooperative approach to further improve the availability of access control solutions. Our cooperative secondary authorization recycling approach 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 the distributed systems involving cooperating parties or replicated services, because of the overlap in their user and resource spaces and the need for consistent policy enforcement.

We have defined CSAR system requirements and presented a detailed design that meets these requirements. We have introduced 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. Cache consistency is managed by dividing all of the policy changes into three categories and employing efficient consistency techniques for each type.

We have evaluated CSAR through both simulations and a prototype implementation. Our results suggest that even with small caches (or low cache warmness), our cooperative authorization solution can offer significant benefits. 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 cooperating SDPs, thus increasing availability of the authorization infrastructure and reducing the load on the authorization server. In addition, depending on the deployment scenario, request processing time is improved by up to 30%.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive comments on this paper, Wing Leung for designing the SAAM evaluation algorithms, Kyle Zeeuwen for the initial development of the SAAM simulation framework, and Craig Wilson for improving the readability of this paper.

7. 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. Beznosov. Flooding and recycling authorizations. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 67–72, Lake Arrowhead, CA, USA, 20-23 September 2005.

[4] 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.

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

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation*, pages 335–350, Seattle, WA, USA, November 6-8 2006.

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

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] J. Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, 1999.

[26] 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.

[27] XACML-TC. OASIS eXtensible Access Control Markup Language (XACML) version 1.0. OASIS Standard, 18 February 2003.