Cooperative Secondary Authorization Recycling

Qiang Wei*, Matei Ripeanu†, Konstantin Beznosov‡

Laboratory for Education and Research in Secure Systems Engineering

lersse.ece.ubc.ca

University of British Columbia

Vancouver, Canada

Technical report LERSSE-TR-2008-02§

April 28, 2008

*qiangw@ece.ubc.ca

†matei@ece.ubc.ca

‡beznosov@ece.ubc.ca

§This and other LERSSE publications can be found at lersse-dl.ece.ubc.ca

Abstract

As enterprise systems, Grids, and other distributed applications scale up and become increasingly complex, their authorization infrastructures—based predominantly on the request-response paradigm—are facing challenges of fragility and poor scalability. We propose an approach where each application server recycles previously received authorizations 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 and performance of authorization infrastructures. 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 average time for authorizing an application request can be reduced by up to a factor of two compared with systems that do not employ cooperation.

Contents

1 Introduction

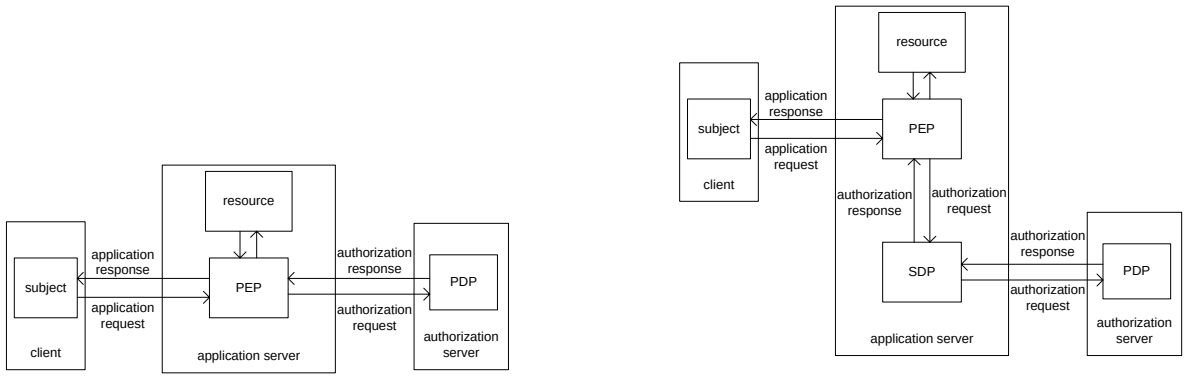
Architectures of modern access control solutions—[17, 11, 20, 24, 22, 10]—are based on the request-response paradigm, illustrated in Figure 1(a). In this paradigm, a policy enforcement point (PEP) intercepts application requests, obtains access control decisions (a.k.a. authorizations) from the policy decision point (PDP), and enforces those decisions.

In large enterprise systems, PDPs are commonly implemented as logically 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 as well as a potential performance bottleneck.

The single point of failure property of the PDP leads to reduced availability: the authorization server 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 [15]. 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 [16, 29]. (For instance, eBay has 12,000 servers and 15,000 application server instances [26].)

In a massive-scale enterprise system with non-trivial authorization policies, making authorization decisions 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 [21]. Additionally, the communication delay between the PEP and the PDP can make authorization overhead prohibitively high.

The state-of-the-practice approach to improving overall system availability and reducing authorization processing delays observed by the client is to cache authorizations at each PEP—what we refer to as *authorization recycling*. Existing authorization solutions



(a) Request-response paradigm.

(b) SAAM adds SDP to the request-response paradigm.

Figure 1: Architectures of modern access control solutions and SAAM.

commonly provide PEP-side caching [17, 11, 20]. 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. [9] propose the Secondary and Approximate Authorization Model (SAAM), which adds a secondary decision point (SDP) to the request-response paradigm (Figure 1(b)). The SDP is collocated with the PEP and can resolve authorization requests not only by precise recycling but also by computing approximate authorizations from cached authorizations. The use 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 SAAM, however, each SDP serves only its own PEP, which means that cached authorizations are reusable only for the requests made through the same PEP. In this paper, we propose an approach where SDPs cooperate to serve all PEPs in a community of applications. Our results show that SDP cooperation further improves the resilience of the authorization infrastructure to network and authorization server failures, and reduces

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 to enforce consistent access control policies. Therefore, authorizations can often be shared among parties or services, 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 evaluate CSAR's feasibility and benefits. Evaluation results show that by adding cooperation to SAAM, our approach further improves the availability and performance of authorization infrastructures. Specifically, the overall cache hit rate can reach 70% even with only 10% of authorizations cached. This high hit rate results in more requests being resolved locally or by cooperating SDPs, when the authorization server is unavailable or slow, thus increasing the availability of authorization infrastructure and reducing the load of the authorization server. Additionally, our experiments show that request processing time can be reduced by up to a factor of two.

The rest of this paper is organized as follows. Section 2 presents SAAM definitions and algorithms. Section 3 describes CSAR design. Section 4 evaluates CSAR's performance, through simulation and a prototype implementation. Section 5 discusses related work. We conclude our work in Section 6.

2 Secondary and Approximate Authorization Model (SAAM)

SAAM [9] is a general framework for making use of cached PDP responses to compute *approximate responses* for new authorization requests. 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 verify 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 an 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 general, the SDP infers approximate responses based on cached primary responses and any information that can be deduced from the application request and system environment. The greater the number of cached responses, the greater the information available to the SDP. As more and more PDP responses are cached, the SDP will become a better and better PDP simulator.

The above abstractions are independent of the specifics of the application and access control policy in question. For each class of access control policies, however, specific algorithms for inferring approximate responses—generated according to a particular access control policy—need to be provided. This paper is based on the SAAM_{BLP} algorithms [9] — SAAAM authorization recycling algorithms for the Bell-LaPadula (BLP) access control model [2].

The BLP model specifies how information can flow within the system based on security

labels attached to each subject and object. A security function λ maps $S \cup O$ to L, where S and O are sets of subjects and objects, and L is a set of security labels. Collectively, the labels form a lattice. The "$\leqslant$" is a partial order relation between security labels (read as "is dominated by"). Subject s is allowed `read` access to object o if s has discretionary read access to o and $\lambda(o) \leqslant \lambda(s)$, while `append` access is allowed if s has discretionary write access to o and $\lambda(o) \geqslant \lambda(s)$.

The SAAM_{BLP} inference algorithms use cached responses 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 and the corresponding responses are r_1, r_2 and r_3, 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 $(r_4, i_4, [r_1, r_2, r_3], allow)$. SAAM_{BLP} uses a special data structure called *dominance graph* to record the relative ordering on subject and object security labels, and evaluates a request by finding a path between two nodes in this directed acyclic graph.

Crampton et al. [9] present simulation results that demonstrate the effectiveness of their approach. With only 10% of authorizations cached, the SDP can resolve over 30% more authorization requests than a conventional PEP with caching. CSAR is, in essence, a distributed version of SAAM. The distribution and cooperation among SDPs can further improve access control system availability and performance, as our evaluation results demonstrate.

3 Cooperative Secondary Authorization Recycling (CSAR)

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 by locally making secondary authorization decisions, by involving other cooperative SDPs in the authorization process, and/or by passing the request to the 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 computational overhead.

- **Ability to deal with malicious SDPs.** As each PEP enforces responses that are possibly offered by non-local SDPs, the PEP should be prepared to deal with those SDPs that after being compromised become malicious. For example, it should verify the validity of each secondary response by tracing it back to a trusted source.

- **Consistency.** Brewer [7] conjectures and Lynch et al. [14] 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 that requests are resolved in minimal time. A deployment with applications generating a high volume of authorization requests,

on the other hand, should attempt to eagerly 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 infrastructures—i.e., PEPs and PDPs—in order to switch to CSAR.

3.2 System Architecture

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

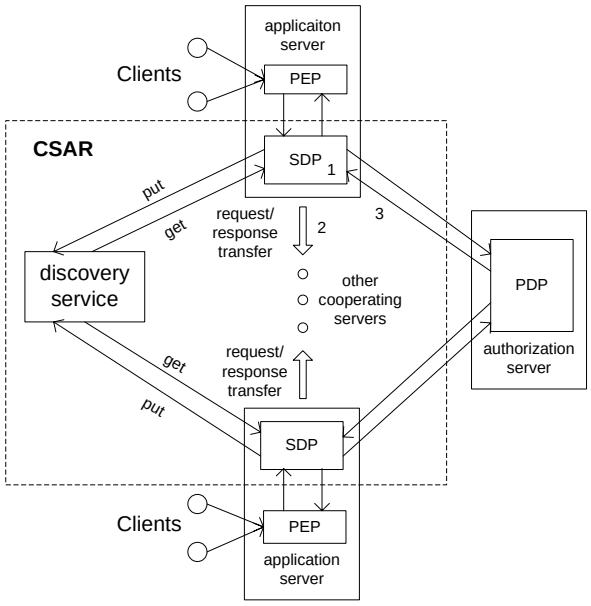


Figure 2: CSAR introduces cooperation between SDPs.

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 from application 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 PDP, to minimizing client-perceived latency, and to minimizing the network traffic generated.

Configurability is achieved by controlling the degree of concurrency 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 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 policy evaluation 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 caching but do not use CSAR, the SDP can offer the same interface and protocol as the legacy component.

3.3 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 [13], 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 cooperating SDPs. 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, we introduced the DS to achieve a selective requests distribution: an SDP in CSAR selectively sends requests only to those SDPs that are likely to be able to resolve them.

For an SDP to resolve a request, the SDP's cache must contain at least both the subject and object of the request. If either one is missing, there is no way for the SDP to infer the relationship between the subject and object, and thus fails to compute a secondary response. The role of the DS is to store and retrieve the mapping between subject/object and SDP addresses. In particular, the DS provides an interface with the following two functions: *put* and *get*. Given a subject or an object and the address of an SDP, the *put* function stores the mapping $(subject/object, SDPaddress)$. A *put* operation can be interpreted as "this SDP knows something about the subject/object." 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 cooperating 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 only logically centralized, but can have a scalable and resilient implementation. In fact, an ideal DS should be distributed and collocated with each SDP to provide high availability and low latency: each SDP can make a local *get* or *put* call to publish or discover cooperative SDPs, and the failure of one DS node will not affect others. Compared to the PDP, the DS is both simple—it only performs *put* and *get* operations—and general—it does not depend on the specifics of any particular security policy. As a result, a scalable and resilient implementation of DS is easier to achieve.

For instance, one can use a Bloom filter to achieve a distributed DS implementation, similar to the summary cache [12] approach. Each SDP builds a Bloom filter from the subjects or objects of cached requests, and sends the bit array plus the specification of the hash functions to the other SDPs. The bit array is the summary of the subjects/objects that this SDP has stored in its cache. Each SDP periodically broadcasts its summary to all cooperating SDPs. Using all summaries received, a specific SDP has a global image of the set of subjects/objects stored in each SDP's cache, although the information could be outdated or partially wrong.

For a small-scale cooperation, a centralized DS implementation might be feasible where various approaches can be used to reduce its load and improve its scalability. The first approach is to reduce the number of *get* calls. For instance, SDPs can cache the results from the DS for a small period of time. This method can also contribute to reducing the latency. The second approach is to reduce the number of *put* calls. For example, SDPs can update the DS in batch mode instead of calling the DS for each primary response.

3.4 Adversary Model

In our adversary model, an attacker can eavesdrop, spoof or replay 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) and the DS. Therefore, there could be malicious clients, PEPs, SDPs and DS in the system.

As a CSAR system includes multiple distributed components, our design assumes different degrees of trust in them. The PDP is the ultimate authority for access control decisions and we assume that all PEPs trust[1] the decisions made by the PDP. We also assume that the policy change manger (introduced later in Section 3.6) is trusted because it is collocated and tightly integrated with the PDP. We further assume that each PEP trusts those decisions that it receives from its own SDP. However, an SDP does not necessarily

[1]By "trust" we mean that if a trusted component turns to be malicious, it can compromise the security of the system.

trust other SDPs in the system.

3.5 Mitigating Threats

Based on the adversary model presented in the previous section, we now describe how our design enables mitigation of the threats due to malicious SDPs and DS.

A malicious DS can return false or no SDP addresses, resulting in threats of three types: (1) the SDP sends requests to those SDPs that actually cannot resolve them, (2) all the requests are directed to a few targeted SDPs, (3) the SDP does not have addresses of any other SDP. In all three cases, a malicious DS impacts system performance through increased network traffic, or response delays, or computational load on SDPs, and thus can mount a denial-of-service (DoS) attack. However, a malicious DS cannot not impact system correctness because every SDP can always resort to queering just the PDP if the SDP detects that the DS is malicious.

To detect a malicious DS, an SDP can track how successful the remote SDPs whose addresses the DS provides are in resolving authorization requests. A benign DS, which always provides correct information, will have a relatively good track record, with just few SDPs unable to resolve requests. Even though colluding SDPs can worsen the track record of a DS, we don't believe such an attack to be of practical benefit to the adversary.

A malicious SDP could generate any response it wants, for example, denying all requests and thus launching a DoS attack. Therefore, when an SDP receives a secondary response from other SDPs, it verifies the authenticity and integrity of the primary responses used to infer that response as well as the correctness of the inference.

To protect the authenticity and integrity of a primary response while it is in transit between the PDP and the SDP, the PDP cryptographically signs the response. Then, an SDP can independently verify the primary response's authenticity and integrity by checking its signature, assuming it has access to the PDP's public key. Recall that each secondary response includes an evidence list that contains the primary responses used for inferring this response. If any primary response in the evidence cannot be verified, that secondary

response is deemed to be incorrect.

To verify the correctness of a response, the SDP needs to use the knowledge of both the inference algorithm and evidence list. A secondary response is correct if the PDP would compute the same response. The verification algorithm depends on the inference algorithm. In the case of SAAM_{BLP}, it is simply the inverse of the inference algorithm. Recall that the SAAM_{BLP} inference algorithm searches the cached responses and identifies the relative ordering on security labels associated with the request's subjects and objects. In contrast, the verification algorithm goes through the evidence list of primary responses by reading every two consecutive responses and checking whether the correct ordering can be derived. To illustrate, consider the example from Section 2. A remote SDP returns a response $(r_4, i_4, [r_1, r_2, r_3], \mathit{allow})$ for request $(s_1, o_2, \mathit{read}, c_4, i_4)$, where r_1 is the primary *allow* response for $(s_1, o_1, \mathit{read}, c_1, i_1)$, r_2 is the primary *allow* response for $(s_2, o_1, \mathit{append}, c_2, i_2)$ and r_3 is the primary *allow* response for $(s_2, o_2, \mathit{read}, c_3, i_3)$. From these responses, the verification algorithm can determine that $\lambda(s_1) > \lambda(o_1) > \lambda(s_2) > \lambda(o_2)$. Therefore, s_1 should be allowed to read o_2, and thus r_4 is a correct response.

Verification of each approximate response unavoidably introduces additional computational cost, which depends on the length of the evidence list. A malicious SDP might use this property to attack the system. For example, a malicious SDP can always return responses with a long evidence list that is computationally expensive to verify. One way to defend against this attack is to set an upper bound to the time that the verification process can take. An SDP that always returns long evidence lists will be blacklisted.

We defined four execution scenarios, listed below, to help manage the computational cost caused by response verification. Based on the administration policy and deployment environment, the verification process can be configured differently to achieve various trade-offs between security and performance.

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

- **Random verification.** Responses are randomly selected for verification. This configuration can be used to detect malicious SDPs but cannot guarantee that the system is perfectly correct, since some false responses may have been generated before the detection.

- **Offline verification.** There is no real-time verification, but offline audits are performed.

3.6 Consistency

Similar to other distributed systems employing caching, CSAR needs to deal with cache consistency issues. In our system, SDP caches may become inconsistent when access control policy changes at the PDP. In this section, we describe how consistency is achieved in CSAR.

We first state our assumptions relevant to the access control systems. We assume that the PDP makes decisions using an access control policy stored persistently in a policy store of the authorization server. 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 XACML architecture [31]. To avoid modifying existing authorization servers and maintain backward compatibility, we further add a policy change manager (PCM), collocated with the policy store. The PCM monitors the policy store, detects policy changes, and informs the SDPs about the changes. The refined architecture of the authorization server is presented in Figure 3.

Based on the fact that not all policy changes are at the same level of criticality, we divide policy changes into three types: critical, time-sensitive, and time-insensitive changes. By discriminating policy changes according to these types, system administrators can choose to achieve different consistency levels. In addition, system designers are able to provide different consistency techniques to achieve efficiency for each type. Our design allows a CSAR deployment to support any combination of the three types. In the rest of this

section, we define each type of policy change and discuss the consistency properties.

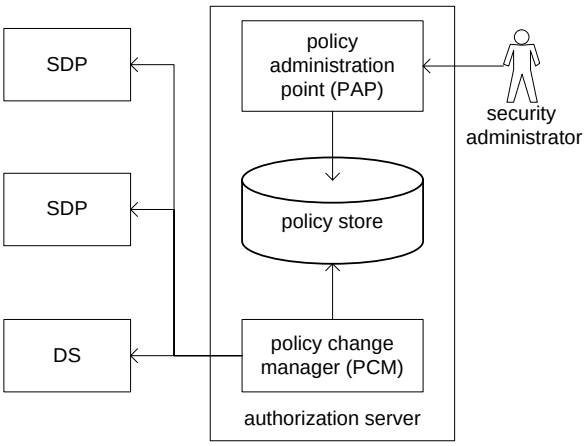


Figure 3: The architecture enabling the support for policy changes.

Critical changes of authorization policies are those changes that need to be propagated urgently throughout the enterprise applications, requiring immediate updates on all SDPs. When an administrator makes a critical change, our approach requires that she also specifies a time period t for the change. CSAR will attempt to make the policy change by contacting all SDPs involved, and must inform the administrator within time period t either a message that the change has been successfully performed or a list of SDPs that have not confirmed the change.

We developed a **selective-flush** approach to propagating critical policy changes. In this approach, only selected policy changes are propagated, only selected SDPs are updated, and only selected cache entries are flushed. We believe that this approach has the benefits of reducing server overhead and network traffic. In the following we sketch out the propagation process.

The PCM first determines which subjects and/or objects (a.k.a. entities) 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 subjects 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 corresponding entries 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.

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.

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 complexities 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.

Time-sensitive changes in authorization policies are less urgent than critical ones but still need to be propagated within a known period of time. 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 a time-to-live (TTL) approach to process time-sensitive changes. Every primary response is assigned a TTL that determines how long the response should remain valid in the cache, e.g., 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, autho-

rizing 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.

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. A simple approach to supporting time-insensitive change is for system administrators to periodically restart the machines hosting the SDPs.

3.7 Eager Recycling

In previous sections, we explained how cooperation among SDPs is achieved by resolving requests by remote SDPs. In this section, we describe an eager approach to recycling past responses. The goal is to further reduce the overhead traffic and response time.

The essence of cooperation is SDPs helping each other to reduce the cache miss rate at each SDP. Tewari et al. [27] divide all cache misses into four categories. In our context, *compulsory* misses and *communication/consistency* misses are most applicable. Compulsory misses are generated by a subject's first attempt to access an object. Communication/consistency misses occur when a cache holds a stale authorization. With cooperation, the SDP can avoid some of these misses by possibly getting authorizations from its cooperating SDPs.

Although cooperation helps to improve the overall cache hit rate, it leads to a small local cache hit rate at each SDP. The reason is that SAAM inference algorithms are based purely on cached primary responses. Without cooperation, the request is always sent to the PDP. The primary response returned from the PDP can then be cached and used for future inference. With cooperation, the request may be resolved by remote SDPs. The returned approximate response, however, is not useful for future inference, which eventually leads to a small local hit rate.

A small local cache hit rate results in two problems. First, it leads to increased response time, as requests have to be resolved by remote SDPs. Its impact is especially significant when SDPs are located in a WAN. Second, it unavoidably increases the computational load of SDPs because each SDP has to resolve more requests for other SDPs. Therefore, the cooperation should also lead to an increased local cache hit rate. To this end, we propose the eager recycling approach, which takes advantage of the information in the evidence list.

As stated before, if an SDP succeeds in resolving a request from another SDP, it returns a secondary response, which includes an evidence component. The evidence contains a list of primary responses that have been used to infer the secondary response. In eager recycling, the receiving SDP incorporates those verified primary responses into its local cache as if it received them from the PDP. By including these responses, the SDP's cache increases faster and its chances of resolving future requests locally by inference also increases. Our evaluation results show that this approach can reduce the response time by a factor of two.

4 Evaluation

In evaluating CSAR, we wanted first to determine if our design works. Then we sought to estimate the achievable gains in terms of availability and performance, and determine how they depend on factors such as the number of cooperating SDPs and the frequency of policy changes.

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. Additionally, we have integrated our prototype with a real application to study the integration complexity and the impact of application performance.

We used a similar setup for both the simulation and prototype experiments. The PDP

made access control decisions on either read or append requests using a BLP-based policy stored in an XML file. The BLP security lattice contained 4 security levels and 3 categories, 100 subjects and 100 objects, and uniformly assigned security labels to them. The total number of possible requests was 100x100x2=20,000. The policy was enforced by all the PEPs. Each SDP implemented the same inference algorithm. While the subjects were the same for each SDP, the objects could be different in order to simulate the request overlap.

4.1 Simulation-based Evaluation

We used simulations to evaluate the benefits of cooperation to system availability and 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 improving the scalability of the system).

We studied the influence of the following factors on the hit rate of one cooperating SDP: (a) the number of cooperating SDPs; (b) the *cache warmness* at each SDP, defined as the ratio of cached responses to the total number of possible responses; (c) the *overlap rate* between the resource spaces of two cooperating SDPs, defined as the ratio of the objects owned by both SDPs to the objects owned only by the studied SDP (The overlap rate served as a measure of similarity between the resources of two cooperating SDPs); (d) whether the inference for approximate responses was enabled or not; and (e) the popularity distribution of requests.

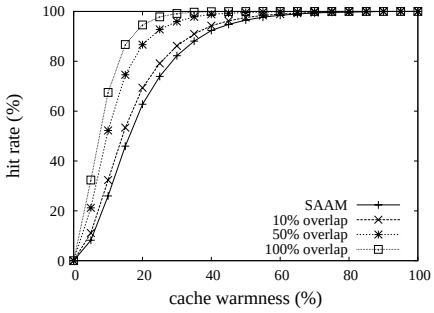
To conduct the experiments, we have modified the SAAM evaluation engine used in [9] to support cooperation. Each run of the evaluation engine involved four stages. In the first stage, the engine generated subjects and objects for each SDP, created a BLP lattice and assigned security labels to both subjects and objects. To control the overlap rate (e.g., 10%) between SDPs, we first generated the object space for the SDP under study (e.g., obj0–99). For each of the other SDPs, we then uniformly selected the corresponding

number of objects (e.g., 10) from the space of the SDP under study (e.g., obj5, obj23, etc.) and then generated the remaining objects sequentially (e.g., obj100–189).

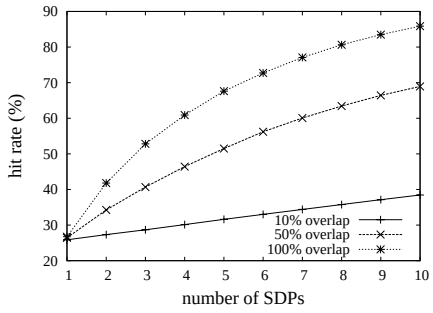
Second, the engine created the *warming set* of requests for each SDP: that is the set containing all possible unique requests for that SDP. We also created a testing set for all SDPs, which comprised a sampling of requests uniformly selected from the request space of the SDP under study. In our experiment, the testing set contained 5,000 requests. Next, the simulation engine started operating by alternating between *warming* and *testing* modes. In the warming mode (stage three), the engine used a subset of the requests from each warming set, evaluated them using the simulated PDP, and sent the responses to the corresponding SDP to build up the cache. Once the desired cache warmness was achieved, the engine switched into testing mode (stage four) where the SDP cache was not updated anymore. We used this stage to evaluate the hit rate of each SDP at controlled, fixed levels of cache warmness. The engine submitted requests from the testing set to all SDPs. If any SDP could resolve a request, it was a cache hit. The engine calculated the hit rate as the ratio of the test requests resolved by any SDP to all test requests at the end of this phase. These last two stages were then repeated for different levels of cache warmness, from 0% to 100% in increments of 5%.

Simulation results were gathered on a commodity PC with a 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. Experiments used the same cache warmness for each SDP and the same overlap rate between the inspected SDP and every other cooperating SDP. In particular, we simulated three overlap rates: 10%, 50%, or 100%. Each experiment was run ten times and the average results are reported. For a confidence level of 95%, the maximum observed confidence interval was 2.3% and the average was 0.5%.

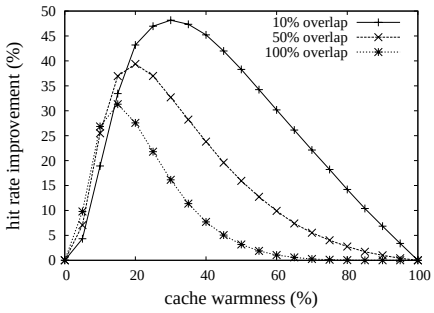
Figure 4 shows the results for requests that followed a uniform popularity distribution. Figure 4(a) shows the dependency of the hit rate on cache warmness and overlap rate. It compares the hit rate for the case of one SDP, representing SAAM (bottom curve), with the hit rate achieved by 5 cooperating SDPs. Figure 4(a) suggests that, when cache warmness was low (around 10%), the hit rate was still larger than 50% for overlap rates of



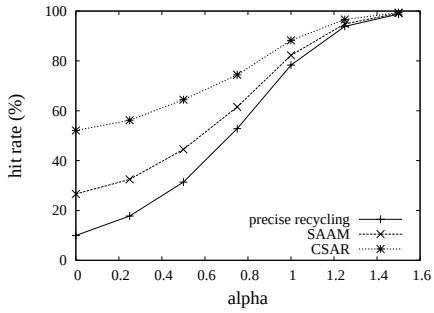
(a) Hit rate as a function of cache warmness for 5 SDPs compared to 1 SDP (i.e., SAAM).



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



(c) Hit rate improvement of approximate recycling over precise recycling as a function of cache warmness when cooperation is enalbed (5 SDPs).



(d) Hit rate as a function of Zipf coefficient. When alpha is 0, it is a uniform distribution.

Figure 4: The impact of various parameters on hit rate. The requests for subfigures (a)—(c) follow a uniform popularity distribution.

50% and up. In particular, when the overlap rate was 100%, CSAR achieved a hit rate of almost 70% at 10% cache warmness. In reality, low cache warmness can be caused by the characteristics of the workload, by limited storage space, or by frequently changing access control policies. For a 10% overlap rate, however, CSAR outperformed SAAM by a mere 10%, which might not warrant the cost of CSAR's complexity.

Figure 4(b) demonstrates the impact of the number of cooperating SDPs on the hit rate under three overlap rates. In the experiment, we varied the number of SDPs from 1 to 10, while maintaining 10% cache warmness at each SDP. As expected, increasing the number of SDPs led to higher hit rates. At the same time, the results indicate that additional SDPs provided diminishing returns. For instance, when the overlap rate was 100%, the first cooperating SDP brought a 14% improvement in the hit rate, while the 10th SDP contributed 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. On the other side, when the overlap rate is small, a large number of SDPs are still required to achieve a high hit rate.

Figure 4(c) shows the absolute hit rate improvement of approximate recycling over precise recycling when cooperation was enabled in both cases. We can observe from Figure 4(c) that the largest improvement occurred when the cache warmness was low. This was due to the strong inference ability of each SDP even at low cache warmness. When the overlap rate decreased, the tops of the curves shifted to the right, which implies that, for a smaller overlap rate, greater cache warmness is needed to achieve more improvement.

In addition, the peak in each curve decreased with the overlap rate. This lowering of the peak appears to be caused by the reduced room for improvement left to the approximate recycling. When the overlap rate increased, the hit rate of precise recycling was already high due to the benefit brought by cooperation.

In the above experiments, the request distribution follows the uniform popularity model, i.e., requests are randomly picked from the request space. However, the distributions of requests in the real world might not be uniform. Breslau et al. [6] demonstrate that

most Web request distributions follow Zipf's Law which expresses a power-law relationship between the popularity P of an item (i.e., its frequency of occurrence) and its rank r (i.e., its relative rank among the referenced items, based on frequency of occurrence). This relationship is of the form $P = 1/r^{\alpha}$ for some constant α.

To study how the request distribution affects hit rate, we also simulated the requests that follow a Zipf object popularity distribution. In the experiment, we varied the coefficient for α between 0 and 1.5. In the case of Zipf, the distribution of items becomes less and less skewed with the decrease of α, reaching a completely uniform distribution at $\alpha = 0$. We expect real-world distributions of requests to be somewhere in the above range. We fixed all other parameters—the cache warmness at 10%, the overlap rate at 50%, the number of SDPs at 5—and varied only α.

Figure 4(d) shows the hit rate as a function of the α for precise recycling, SAAM and CSAR. It suggests that the hit rate increases along with the α in all three cases. This is expected because requests repeat more often when alpha increases. When alpha was 1.5, all recycling schemes achieved a nearly 100% hit rate. It is also clear that the hit rate improvement due to cooperation only was reduced with the increase of α. The reason appears to be two-fold. First, with requests following Zipf distribution, the hit rate in the local cache of each SDP was already high, so that there was less room for improvement through cooperation. Second, unpopular requests had a low probability to be cached by any SDP. Therefore, the requests that could not be resolved locally were unlikely to be resolved by other SDPs either.

Summary: The simulation results suggest that combining approximate recycling and cooperation can help SDPs to achieve high hit rates, even when the cache warmness is low. This improvement in hit rate increases with SDPs' resource overlap rate and the number of cooperating SDPs. We also demonstrate that when the distribution of requests is less skewed, the improvement in hit rate is more significant.

4.2 Prototype-based Evaluation

This section describes the design of our prototype and the results of our experiments. The prototype system consisted of the implementations of PEP, SDP, DS, PDP, and a test driver, all of which communicated with each other using Java Remote Method Invocation (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 either sequentially or concurrently. Each SDP maintained a dynamic pool of worker threads that concurrently queried other SDPs. The DS used a customized hash map that supported assigning multiple values (SDP addresses) to a single key (subject/object).

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 propagating policy changes. To support response verification, we generated a 1024-bit RSA key pair for the PDP. Each SDP had 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. In the following, we present and discuss the results of evaluating the performance of CSAR in terms of response time, the impact of policy changes on hit rate, and the integration of CSAR with a real application.

4.2.1 Evaluating Response Time

First we compared the client-perceived response time of CSAR with that of the other two authorization schemes: without caching and SAAM (without cooperation). We studied three variations of CSAR: sequential authorization, concurrent authorization and eager recycling. We also evaluated the impact of response verification on response time in the case of sequential authorization. We ran experiments in the following three scenarios, which varied in terms of the network latency among SDPs, and between SDPs and the PDP:

(a) **LAN-LAN.** SDPs and the PDP were all deployed in the same local area network (LAN), where the round-trip time (RTT) was less then 1ms.

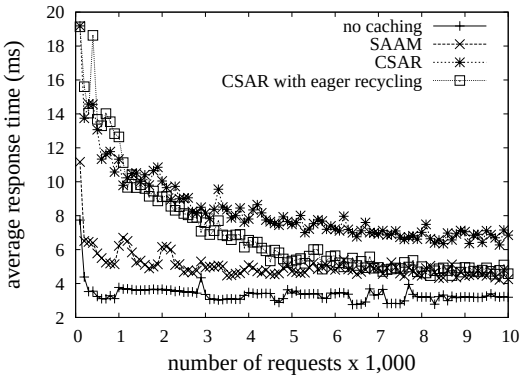
(b) **LAN-WAN.** SDPs were deployed in the same LAN, which was separated from the PDP by a wide area network (WAN). To simulate network delays between SDPs and the PDP, we added a 40ms delay to each authorization request sent to the PDP.

(c) **WAN-WAN.** All SDPs and the PDP were separated from each other by a WAN. Again, we introduced a 40ms delay to simulate delays that possibly occur in both the remote PDP and remote SDPs.

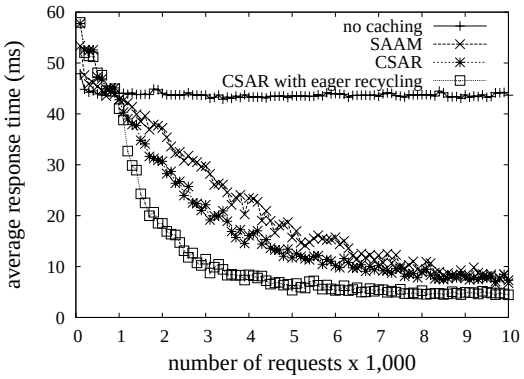
In the experiments we did not intend to test every combination of scenarios and authorization schemes, but to test those most plausibly encountered ones in the real world. For example, concurrent authorization and response verification were only enabled in the WAN-WAN scenario when SDPs were remotely located. Using concurrent authorization in this scenario can help to reduce the high cost of cache misses on remote SDPs due to communication costs. In addition, since the requests in such a scenario are obtained from remote SDPs that might be located in a different administrative domain, response verification is highly desirable.

The experimental system consisted of a PDP, a DS, and four PEP processes collocated with their SDPs. Note that although the system contained only one DS instance, this DS simulated an idealized implementation of a distributed DS where each DS had up-to-date global state. This DS instance could be deemed to be local to each SDP because the latency between the DS and the SDPs was less than 1ms and the DS was not overloaded. Each two collocated PEPs and SDPs shared a commodity PC with a 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. In all experiments, we made sure that both machines were not overloaded so that they were not the bottlenecks of the system and did not cause additional delays.

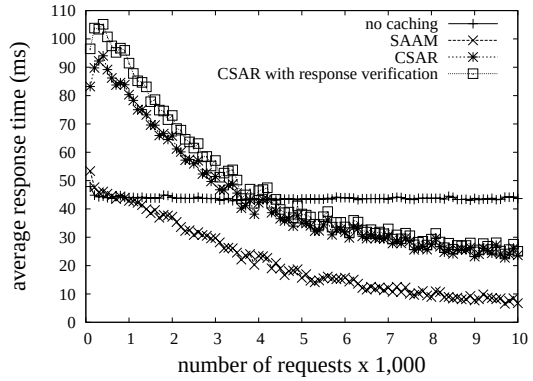
At the start of each experiment, the SDP caches were empty. The test driver maintained one thread per PEP, simulating one client per PEP. Each thread sent randomly generated

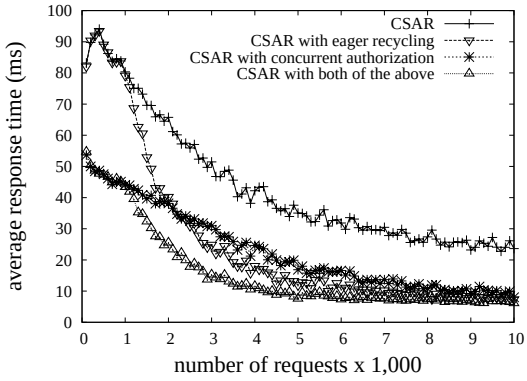


(a) LAN-LAN: SDPs and the PDP are located in the same LAN.



(b) LAN-WAN: SDPs are located in the same LAN while the PDP is located in a WAN.





(c) WAN-WAN: SDPs and the PDP are separated by a WAN.

Figure 5: Response time as a function of the number of requests observed by SDPs. The requests follow a uniform distribution.

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 that period. We ran the experiment when the cache size was within 10,000 requests for each SDP at the end, which was one-half the total number of possible requests. Figures 5(a)–5(c) show the plotted results for 100% overlap rate. For the sake of better readability, we present the results for WAN-WAN scenario in two graphs. The following conclusions regarding each authorization scheme can be directly drawn from these figures:

(i) In the **no-caching** scheme, SDPs were not deployed and PEPs sent authorization requests directly to the PDP. In the LAN-LAN scenario, this scheme achieved best performance since it involved the least number of RMI calls in our implementation. In the other two scenarios, however, all of the average response times were slightly higher than 40ms because all requests had to be resolved by the remote PDP.

(ii) In the **non-cooperative caching (SAAM)** scheme, SDPs were deployed and available only to their own PEPs. When a request was received, each SDP first tried to resolve the request locally and then by the PDP. For the LAN-LAN scenario, this method did not help reduce the latency because in our prototype SDP was implemented as a separate process and each SDP authorization involved an RMI call. In the other two scenarios, response times decreased consistently with the number of requests because more requests were resolved locally. Note that the network distance between SDPs does not affect the results in this and the previous scenario, since either no caching or no cooperation was involved.

(iii) In the **CSAR** scheme, SDPs were deployed and cooperation was enabled. When a request was received, each SDP resolved the request *sequentially*. For the LAN-LAN scenario, the response time was the worst because this scenario involved most RMI calls in our implementation. For the LAN-WAN scenario, using cooperation helped to slightly reduce the response time compared with the SAAM method, because resolving requests by other SDPs is cheaper than by the remote PDP. However, this

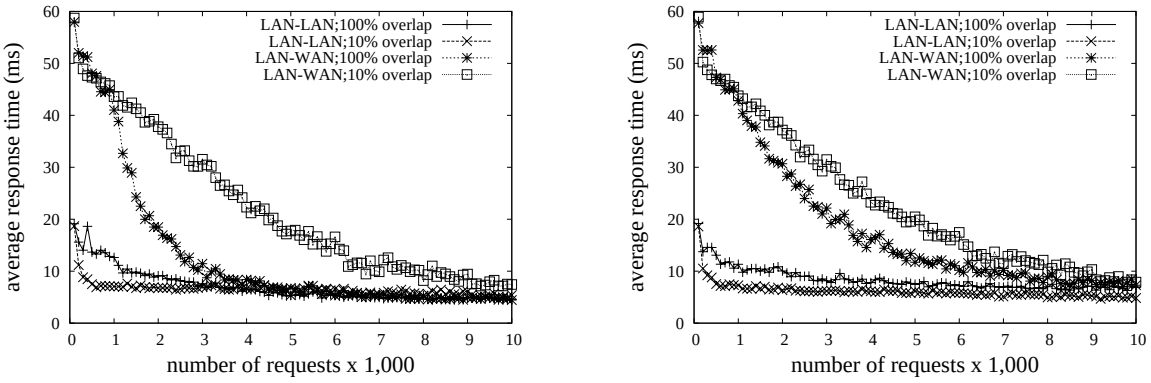
improvement continuously decreases, because more and more requests can be resolved locally. For the WAN-WAN scenario, using CSAR was worse than using just SAAM due to the high cost of cache misses on remote SDPs.

(iv) In **CSAR with the response verification** scheme, each response returned from remote SDPs was verified. Figure 5(c) shows that the impact of response verification on response time was small: response time increased by less than 5ms on average. When the local cache increased, this overhead became smaller since more requests could be resolved by the local SDP; thus, less verification was involved. Note that the time for response verification was independent of the testing scenario, which means that the 5ms verification overhead applied to the other two scenarios. This is why we did not show verification time in the graphs for the other scenarios.

(v) In **CSAR with the eager recycling** scheme, the primary responses from the evidence lists of secondary responses were incorporated into each SDP's local cache. As expected, eager recycling helped to reduce the response time in all three scenarios, and the effect was especially significant when the PDP or SDPs were remote, since more requests can quickly be resolved locally. The maximum observed improvement in response time over SAAM was by a factor of two. The results also demonstrate that the response time was reduced only after some time. This is because the evidence lists became useful for eager recycling only after the remote SDPs have cached a number of requests.

(vi) In **CSAR with the concurrent authorization** scheme, each SDP resolved the requests *concurrently*. Figure 5(c) demonstrates that the response time was significantly reduced in the beginning and decreased consistently. The drawback of concurrent authorization, however, is that it increases the overhead traffic and causes extra load on each SDP and the PDP. It could be a subject of future research to study and try to reduce this overhead.

(vii) In **CSAR with both eager recycling and concurrent authorization** scheme,



(a) CSAR with eager recycling.

(b) CSAR without eager recycling.

Figure 6: Response time comparison between overlap rate of 10% and 100%. The requests follow a uniform distribution.

both eager recycling and concurrent authorization were enabled. Figure 5(c) shows that this method achieved the best performance among those tested.

The above conclusions were drawn for 100% overlap rate. For comparison, we also ran the experiments using 10% overlap rate in LAN-LAN and LAN-WAN scenarios. Figure 6 compares the response times for the CSAR with and without eager recycling. In the LAN-WAN scenario, the small overlap rate led to increased response time for both schemes because more requests had to resort to the PDP, and the eager recycling scheme experienced more increases. On the other hand, in the LAN-LAN scenario, the response time was reduced in the beginning with the small overlap rate due to the reduced number of RMI calls, since the SDP sent most requests to the PDP directly rather than first to other SDPs which could not help.

Summary: The above results demonstrate that although using CSAR with sequential authorization may generate higher response times, adding eager recycling and/or concurrent authorization helps to significantly reduce the response time. Eager recycling is responsible for the effective increase of cache warmness, while concurrent authorization enables SDPs to use the fastest authorization path in the system.

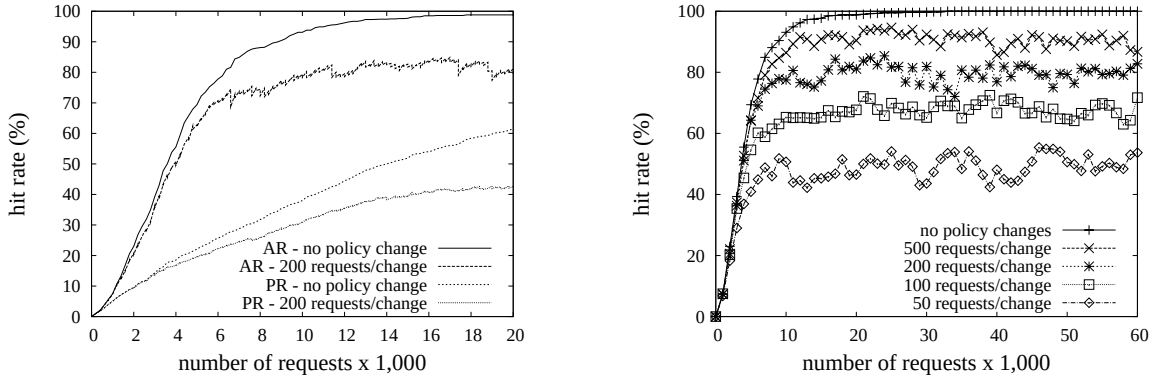
4.2.2 Evaluating the Effects of Policy Changes

We also used the prototype to study CSAR's behavior in the presence of policy changes. Since the hit rate depends on the cache warmness, and a policy change may result in removing one or more responses from SDP 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.

In all our experiments for policy changes, the overlap rate between SDPs was 100% and the requests were randomly generated. The test driver maintained a separate thread responsible for firing a random policy change and sending the policy change message to the PCM at pre-defined intervals, e.g., after every 100 requests. To measure the hit rate at run-time, we employed a method similar to the one used during the simulation experiments. Each request sent by the test driver was associated with one of two modes: *warming* and *testing*, used for warming the SDP caches or testing the cumulative hit rate respectively. Each experiment switched from the warming mode to the testing mode when a policy change message was received. After measuring the hit rate right before and after each policy change, the experiment switched back to the warming mode.

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 so 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.

Figure 7(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.



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

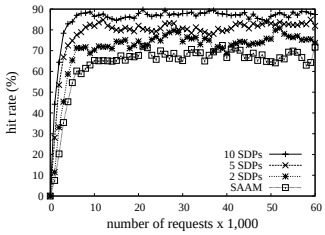
(b) Hit rate as a function of number of requests at various frequencies of policy change.

Figure 7: The impact of policy changes on hit rate with a single SDP. The requests follow a uniform popularity distribution.

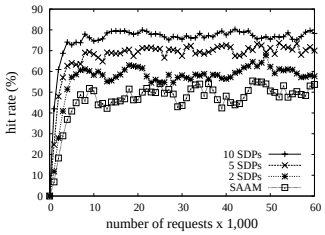
Figure 7(a) indicates that the hit-rate drops are small for both approximate recycling and precise recycling. For approximate recycling, the largest hit-rate drop was 5%, and most of the other drops were around 1%. After each drop, the curve climbed again because the cache size increased with new requests.

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 policy changes. The reason is that approximate recycling employs an inference algorithm based on a directed acyclic graph. A policy change could partition the graph or just increase its diameter, resulting in a greater reduction in the hit rate.

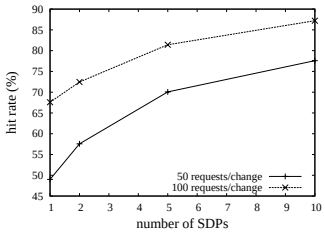
Although the hit-rate drop for each policy change was small, one can see that the cumulative effect of policy changes could be large. As Figure 7(a) shows, the hit rate of approximate recycling decreased about 20% in total when the request number reached 20,000. This result led us to another interesting question: Would the hit rate finally



(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 8: The impact of SDP cooperation on hit rate when policy changes. The requests follow a uniform popularity distribution. The overlap rate between SDPs is 100%.

stabilize at some point?

To answer this question, we ran another experiment to study how the hit rate varied 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 7(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, one cannot 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 were, the lower the stabilized hit rates were, since the responses were removed from the SDP caches more frequently.

Figure 7(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 hit rate under continuous policy changes could benefit from the cooperation. In these experiments, we varied the number of SDPs from 1 to 10. Figure 8(a) and Figure 8(b) show hit rates versus the number of requests observed when the policy changed every 50 and 100 requests. Figure 8(c) compares the eventual stabilized hit rate for the two frequencies of policy changes. As we expected, cooperation between SDPs improved the hit rate.

Note that when the number of SDPs increased, the curves after the knee became smoother. This trend was a direct result of the impact of cooperation on the hit rate: cooperation between SDPs compensates for the hit-rate drops caused by the policy changes at each SDP.

Summary: Our results show that the impact of a single policy change on the hit rate is small, while the cumulative impact can be large. Constant policy changes finally lead to a stabilized hit rate, which depends on the frequency of the policy change. In any case, cooperation helps to reduce this impact.

4.2.3 Integration with TPC-W

In this section, we describe the work on integrating CSAR with TPC-W [28], an industry-standard e-commerce benchmark application that models an online bookstore such as Amazon.com. Our primary goal was to understand the complexity of integrating CSAR with real applications, and the secondary goal was to study the impact of policy enforcement on application performance.

A TPC-W deployment consisted of a front-end application server and a database server. We used Apache Jakarta Tomcat 5.5.4 as the application server and MySQL 5.0 as the database server. The Java code run by the application server to generate the web pages and interface with the database was derived from the code freely available from the University of Wisconsin PHARM project [23], whose code implements both the servlets for the business logic and a Java remote browser emulator (RBE) for driving the experiment.

Figure 9 shows the system architecture that integrates CSAR with the TPC-W on a

single application server. To enforce the access control policy, we added the PEP, the SDP, the PDP and a policy file to the original TPC-W architecture. The PEP was implemented as a servlet filter that contained about 100 lines of code. The PEP dynamically intercepted application requests and used them to generate authorization requests which only included the information about the subject, object and access right. We assured that every application request was intercepted by the PEP, then the authorization request was sent to its SDP for decision. The PDP and the SDP were reused from our prototype implementation with only minor changes to the system configuration file.

We modeled three user roles (or security labels) in the access control policy: visitor, customer and administrator. Each role could access a number of application objects. For example, the visitor could only browse books; the customer could not only browse books but also buy books; the administrator could only access the administration pages for managing books.

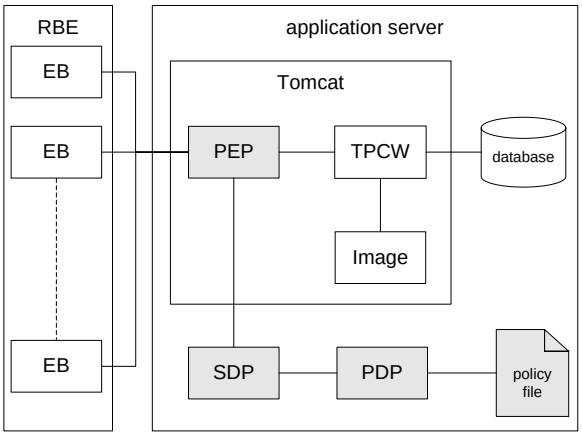
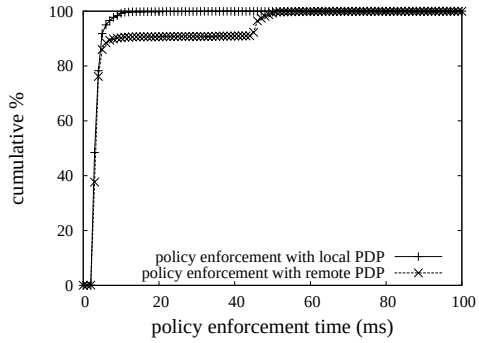


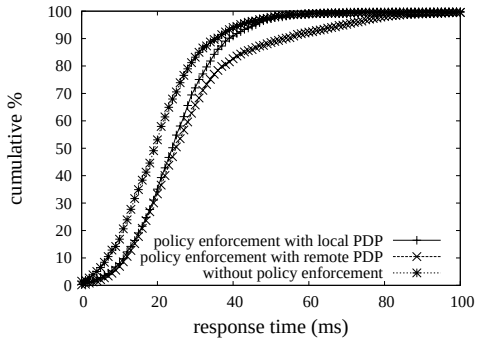
Figure 9: Adding CSAR-based policy enforcement to TPC-W.

The experiment used 10 emulated browsers (EBs) to simulate 10 concurrent users and each EB followed a browsing mix behavior defined by the TPC-W specification. We used two PCs in the experiments: one was used for running EBs while the other was used as both application server and database server. Each experiment lasted 30 minutes. We simulated both remote and local PDPs as defined in previous sections, and studied the increase in response times introduced by policy enforcement.

First, we were interested to understand how much time was used for policy enforcement in our setup. We measure the policy enforcement time as the time between the event that the PEP receives a requests from the client and the event that the PEP receives a response from the SDP. Figure 10(a) shows the cumulative distribution of policy enforcement time.



(a) Cumulative distribution of response time



(b) Cumulative distribution of policy enforcement time

Figure 10: The impact of policy enforcement on response time.

For the local PDP, unsurprisingly, almost all policy enforcement times were small, i.e., less than 10ms. For the remote PDP, 10% of the response times were between 40ms and 50ms, which means that these requests were resolved by remote PDPs. The fact that the 90% of enforcement times were less than 10ms even when the PDP was remote, implies a high hit rate on the SDP, which we believe was due to the access pattern and object space of the TPC-W application. In this experiment, each emulated session lasted 15 minutes and client thinking time was 7 seconds, as specified by the TPC-W standard. This means that in each session each user accessed about 128 pages (a.k.a. objects). On the other hand, the TPC-W application consists of only 14 unique pages. Combining these two facts, each page would have been accessed around 9 times in an average session. This behavior unavoidably led to a high cache hit rate.

Second, we were interested to understand how the policy enforcement affected the overall TPC-W response time. Figure 10(b) plots the cumulative distribution of overall TPC-W response times in three scenarios: without policy enforcement, and with policy enforcement by either a local PDP or remote PDP. Compared with the scenario without policy enforcement, the following observations can be made: (1) when the PDP was remote, the average response time increased by 20% since 10% policy enforcement time was between

0 and 10ms, and another 10% was between 40ms and 50ms; (2) when the PDP was local, its curve before 20ms matched the curve of remote SDPs, while its curve after 50ms matched the curve of no policy enforcement. The reason was that most policy enforcement times were less than 10ms.

Summary: Our experience shows that the prototype can be easily integrated with the Java-based TPC-W application. The PEP can be simply implemented as a Java servlet filter and the other CSAR components require few changes. The results show that the overhead of policy enforcement is highly dependent on the application's requests pattern and object space. In the case of TPC-W, a single SDP achieves a 90% hit rate, thus the impact on the overall application performance is small. Based on this result, we decided not to run the experiment with cooperation, as the improvement space for cooperation was only 10%.

5 Related Work

CSAR is related to several research areas, including authorization caching, collaborative security, and cooperative caching. This section briefly reviews the work in these fields.

To improve the performance and availability of access control systems, caching has been employed in a number of commercial systems [17, 11, 20], as well as several academic distributed access control systems [1, 5]. None of these systems involves cooperation among caches, and most of them adopt a TTL-based solution to manage 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. [9] develop SAAM by introducing the design of SDP and concrete inference algorithms for BLP model. CSAR builds on SAAM and extends it by enabling applications to share authorizations. 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. Beznosov et al. [4]

present a resource access decision (RAD) service for CORBA-based distributed systems. The RAD service allows dynamically adding or removing PDPs that represent different sets of polices. In Stowe's scheme [25], 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 schemes all assume that a request needs to be authorized by multiple PDPs and each PDP maintains different policies. CSAR, on the other hand, focuses on the collaboration of PEPs and assumes that they enforce the same policy. This is why we consider these schemes to be orthogonal to ours.

Our research can be considered a particular case of a more general direction, known as *collaborative security*. This direction aims at improving security of a large distributed system through the collaboration of its components. A representative example of collaborative security is collaborative application communities [18], in which applications collaborate on identifying previously unknown flaws and attacks and notifying each other. Another example is Vigilante [8], which enables collaborative worm detection at end hosts, but does not require hosts to trust each other. CSAR can be viewed as a collaborative security since different SDPs collaborate on resolving authorization requests to mask PDP failures or to improve performance.

Outside of the security domain, cooperative web caching is another related area. Web caching is a widely used technique for reducing the latency observed by web browsers, decreasing the aggregate bandwidth consumption of an organization's network, and reducing the load on web servers. Several projects have investigated decentralized, cooperative web caching (please refer to [30] for a survey). Our approach differs from them in the following three aspects. First, CSAR supports approximate authorizations that are not pre-cached and must be computed dynamically. Second, the authorizations from other SDPs need to be verified to ensure authenticity, integrity and correctness. Third, CSAR supports various consistency requirements.

6 Conclusion

As distributed systems scale up and become increasingly complex, their access control infrastructures face 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 approach exploits the increased hit rate offered by a larger, distributed cooperative cache of access control decisions. We believe that this solution is especially practical in distributed systems involving cooperating parties or replicated services, due to the high overlap in their user/resource spaces and the need for consistent policy enforcement.

This paper defines CSAR system requirements, and presents a detailed design that meets these requirements. We have introduced a response verification mechanism that does not require cooperating SDPs to trust each other. Cache consistency is managed by dividing all of the policy changes into three categories and employing efficient consistency techniques for each category.

A decision on whether to deploy CSAR depends on a full cost-benefit analysis informed by application- and business-specific factors, for example, the precise characteristics of the application workload and deployment environment, an evaluation of the impact of system failures on business continuity, and an evaluation of the complexity associated costs of the access control system. To inform this analysis, we have evaluated CSAR's application-independent benefits: higher system availability by masking network and PDP failures through caching, lower response time for the access control subsystem, and increased scalability by reducing the PDP load; and costs: computational and generated traffic overheads.

The results of our CSAR evaluation 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 reduced by up to a factor of two, compared with solutions that do not cooperate.

References

[1] Lujo Bauer, Scott Garriss, and Michael 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] Konstantin 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] Konstantin Beznosov, Yi Deng, Bob Blakley, Carol Burt, and John Barkley. A resource access decision service for CORBA-based distributed systems. In *Annual Computer Security Applications Conference*, pages 310–319, Phoenix, Arizona, USA, 1999.

[5] Kevin Borders, Xin Zhao, and Atul Prakash. CPOL: high-performance policy evaluation. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, pages 147–157, New York, NY, USA, 2005. ACM Press.

[6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, pages 126–134, 1999.

[7] Eric A. Brewer. Towards robust distributed systems. In *(Invited Talk) PODC*, Portland, Oregon, 2000.

[8] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the 2005 SOSP*, Brighton, UK, 2005.

[9] Jason Crampton, Wing Leung, and Konstantin Beznosov. Secondary and approximate authorizations model and its application to Bell-LaPadula policies. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 111–120, Lake Tahoe, CA, USA, June 7–9 2006.

[10] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, 2001.

[11] Entrust. GetAccess design and administration guide. Technical report, Entrust, September 20 1999.

[12] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[13] Syam Gadde, Jeff Chase, and Michael Rabinovich. A taste of crispy Squid. In *Proceedings of the 1998 Workshop on Internet Server Performance*, pages 129–136, June 1998.

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

[15] 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.

[16] Zbigniew Kalbarczyk, Ravishankar K. Lyer, and Long Wang. Application fault tolerance with Armor middleware. *IEEE Internet Computing*, 9(2):28–38, 2005.

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

[18] Michael Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software self-healing using collaborative application communities. In *Proceedings of the 2006 NDSS*, pages 95–106, San Diego, CA, 2006.

[19] Paul 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] Netegrity. Siteminder concepts guide. Technical report, Netegrity, 2000.

[21] 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.

[22] OMG. Common object services specification, security service specification v1.8, 2002.

[23] Pharm. Java TPC-W implementation distribution. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2003.

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

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

[26] Paul Strong. How Ebay scales with networks and the challenges. In *the 16th IEEE International Symposium on High-Performance Distributed Computing*, Monterey, CA, USA, 2007. Invited talk.

[27] Renu Tewari, Michael Dahlin, and Harrick Vin. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 273, Washington, DC, USA, 1999.

[28] TPC. TPC-W: Transactional web benchmark version 1.8. http://www.tpc.org/tpcw/, 2002.

[29] Werner Vogels. How wrong can you be? Getting lost on the road to massive scalability. In *the 5th International Middleware Conference*, Toronto, Canada, October 20 2004. Keynote address.

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

[31] XACML-TC. OASIS eXtensible Access Control Markup Language (XACML) version 2.0. OASIS Standard, 1 February 2005.