The Secondary and Approximate Authorization Model and its Application to Bell-LaPadula Policies

Jason Crampton
Information Security Group
Royal Holloway, University of London
jason.crampton@rhul.ac.uk

Wing Leung
LERSSE*
University of British Columbia
wingl@ece.ubc.ca

Konstantin Beznosov
LERSSE*
University of British Columbia
beznosov@ece.ubc.ca

ABSTRACT

We introduce the concept, model, and policy-specific algorithms for inferring new access control decisions from previous ones. Our *secondary and approximate authorization model* (SAAM) defines the notions of *primary* vs. *secondary* and *precise* vs. *approximate* authorizations. Approximate authorization responses are inferred from cached primary responses, and therefore provide an alternative source of access control decisions in the event that the authorization server is unavailable or slow. The ability to compute approximate authorizations improves the reliability and performance of access control sub-systems and ultimately the application systems themselves.

The operation of a system that employs SAAM depends on the type of access control policy it implements. We propose and analyze algorithms for computing secondary authorizations in the case of policies based on the Bell-LaPadula model. In this context, we define a *dominance graph*, and describe its construction and usage for generating secondary responses to authorization requests. Preliminary results of evaluating SAAM_{BLP} algorithms demonstrate a 30% increase in the number of authorization requests that can be served without consulting access control policies.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; C.2.0 [**Computer Communication Networks**]: Security and Protection

General Terms

Security, Theory

*Laboratory for Education and Research in Secure Systems Engineering (`lersse.ece.ubc.ca`)

SACMAT'06, June 7–9, 2006, Lake Tahoe, California, USA.

Keywords

SAAM, authorization recycling, access control, Bell-LaPadula model

1. INTRODUCTION

Architectures of modern access control solutions—such as [13, 8, 19, 25, 21, 6]—are based on the request-response paradigm. 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 enterprises, PDPs are commonly implemented as dedicated authorization servers [13, 8, 19], providing such important benefits as consistent policy enforcement across multiple PEPs and the reduction of authorization policy administration. The drawbacks are also critical: reduced performance due to communication delays between the PEP and PDP, as well as reduced reliability, since each PEP depends on its PDP and the network connecting them.

The state-of-the-practice approach to improving overall system reliability and decreasing processing delays observed by system users is to cache authorization decisions at each PEP—what we refer to as *authorization recycling*. Enterprise authorization solutions [13, 8, 19] commonly provide PEP-side caching. However, these solutions employ a simple form of authorization recycling: a cached decision is reused only if the authorization request in question exactly matches the original request for which the decision was made. We refer to such reuse as *precise authorization* recycling. This paper explores the possibilities of improving system reliability and performance by inferring *approximate authorizations* based on information in the PEP's cache.

The prospect of recycling approximate authorizations is attractive for two reasons: the lack of fault tolerance (FT) techniques that scale to large populations while remaining cost-effective; and the high cost of requesting, making, and delivering an authorization due to communication delays. A conventional approach to improving reliability and availability of a distributed infrastructure is failure masking through redundancy—either information, time, or physical redundancy [10]. However, redundancy and other general-purpose fault-tolerance techniques for distributed systems scale poorly, and become technically and economically infeasible to employ when the number of entities in the system reaches thousands [12, 27].

Consider, for instance, an enterprise with hundreds of thousands of networked commodity computers. Even if the

mean time to failure (MTTF) of each computer (and all its critical software and hardware components) were one year, in an enterprise with half a million computers,[1] over 1,300 would fail every day. With the average availability of each machine maintained at a level of 99.9% (a somewhat unrealistically high expectation for such a large population of commodity equipment and software), almost 500 computers would be unavailable at any given moment. As generic FT techniques do not scale for enterprises of such size, domain-specific approaches become attractive. SAAM is a representative approach in which the specifics of access control policies are employed to mask PDP-related failures.

Making access control decisions can also be time consuming. In a large enterprise, arriving at an authorization decision could be computationally expensive due to the heterogeneity and large size of the object and subject populations. In addition, evaluating authorization policy could require obtaining just-in-time data from human resources, medical records, and other repositories of business data, which commonly further increases the time required for making an access control decision. The overall delay in requesting and obtaining an authorization for short transactions could make the authorization overhead prohibitively expensive. With business and policy decisions often made to be just good enough, low-cost approximate authorizations could provide a viable alternative.

In this paper, we introduce a general framework for making use of PDP responses that are cached by the PEP. These responses are based directly on information contained in the access control policy—so-called *primary evidence*. We use these cached responses to infer responses to new access requests in the event that the PDP is unavailable. Such responses are based on *secondary evidence* contained in the PEP's cache.

The *secondary and approximate authorization model* (SAAM) is a conceptual framework for authorization requests and responses. It is independent of the specifics of the underlying application and access control policy. For each class of access control policy, specific algorithms for making inferences about the authorization responses generated by a particular access control policy need to be provided.

After introducing SAAM, we extend it to include inference algorithms for access control policies based on the Bell-LaPadula (BLP) model [1, 2]. Our approach to authorization inference for BLP is based on the idea of a *dominance graph*, which provides a partial mapping of subjects and objects to security labels. We illustrate how to use the dominance graph to compute secondary responses, and briefly discuss the computational complexity of our approach.

We are in the process of evaluating our SAAM_{BLP} algorithms. Preliminary results indicate that even with small numbers of objects and subjects per security label, our algorithms yield up to a 30% better "hit rate" for a SAAM-enabled PEP cache than for a conventional one, which recycles precise authorizations only.

The rest of the paper is organized as follows. The next section introduces the fundamental elements of SAAM. Section 3 defines SAAM_{BLP}, the extension of SAAM to the Bell-LaPadula model. We describe the dominance graph and algorithms for constructing and querying the graph. Section 4 discusses related work. In Section 5, we draw conclusions from the obtained results and discuss future work.

[1]Some enterprises, such as Amazon.com, are not far away from having that many computers, which have MTTF as short as two months [27].

2. SAAM BUILDING BLOCKS

Users of a computer system generate requests to access resources maintained by that system. We assume that the computer system enforces some kind of access control policy, by which different users have different degrees of access to the protected resources of the system. In particular, we are interested in distributed computer systems in which distinct, trusted software components are used to

1. intercept application requests;
2. decide whether each request should be granted by evaluating the request with reference to the system's access control policy.

These components are often known as the *policy enforcement point* (PEP) and the *policy decision point* (PDP), respectively. Figure 1 illustrates a generic access control architecture for distributed computer systems. PEPs vary depending on the technology and the application. They can be security interceptors, as in CORBA [21], ASP.NET [17], and most Web servers, or part of the component container, as in COM+ [7] and EJB [6]. They can also be simply application code, as in the case of current implementations, sometimes based on static or dynamic "weaving" using aspect-oriented software development techniques [14].

We distinguish between the *application request* generated by the user process, which is intercepted by the PEP, and the *authorization request*, which is generated by the PEP and forwarded to the PDP for evaluation. We make this distinction because in distributed computing environments, the format of the authorization request must be compatible with the PDP logic and is commonly quite different from the format and content of the application request. For example, this transformation is performed by the *context handler* in the XACML-compliant PEP. The context handler generates an XACML *request context*, which is sent to the PDP for processing [20]. Another example is the authorization request made by the CORBA Security Interceptor to the Access Decision Object (ADO), which acts as a PDP. The request to the ADO supplies subject's attributes, target object ID, implemented interface, and operation to be invoked, but omits the parameters of the application request. For the sake of brevity, we will write *request* for *authorization request*.'

The PDP returns an *authorization decision* (or simply, *decision*) to the PEP, which enforces that decision. The decision is based on the *access control policy* (a.k.a. *authorization information*) maintained by the PDP.

In what follows, we assume that the PDP is unable to make a timely decision, either because of hardware or software failures, or because of communication problems or high network latency. Instead, the PEP caches authorization requests and responses, and compares new requests with this cached information in order to compute authorization decisions. As these decisions are not obtained from the PDP, they are by necessity *secondary*. The following sub-sections define the building blocks of a general model and specific algorithms for making timely and safe secondary decisions: authorization requests and responses, secondary decision point, and the strategy for computing approximate authorizations.

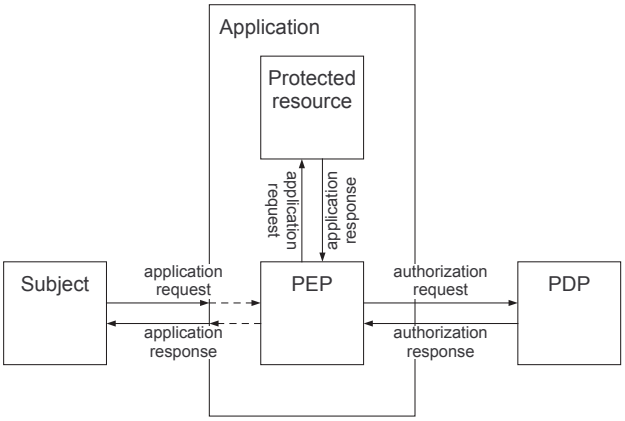


Figure 1: Generic access control architecture

2.1 Authorization Requests

Users are modelled as *subjects*, which are generally understood to be the computer processes or threads associated with the application program(s) run by the user. A subject is associated with security-relevant attributes, which typically include identifiers for the user and groups or roles to which the user belongs. Resources are modelled as *objects*. Generally there may be more than one form of interaction between a subject and an object: typical examples include read, write, and execute. Hence, an access request has traditionally been modelled as a triple (s, o, a), where s is a subject, o is an object, and a is an action or access right. It is common nowadays to include contextual information in an access request, for example, the time of the request or the location of the subject.

Definition 1. *An* authorization request *is a tuple* (s, o, a, c, i)*, where* s *is a subject,* o *is an object,* a *is an access right,* c *is contextual information, and* i *is a request identifier.*

Each request has a unique identifier. It is commonly supported by the underlying technologies for matching authorizations to the corresponding requests. For example, in middleware based on the semantics of remote procedure calls (RPC), such as CORBA [21], EJB [6], DCOM [4], and DCE [15], an object request broker (ORB) uses request IDs to pair the outgoing requests on remote objects with the incoming responses. The format and representation of request identities are technology-specific.

In many cases we are interested in requests that are identical except for their respective identifiers. Hence we use the following notation for requests interchangeably: (q, i) and (s, o, a, c, i). We say that the requests (q, i) and (q, i') are *equivalent*.

2.2 Authorization Responses

An authorization response may be generated by the PDP or it may be generated by the PEP using cached information. In this paper, we will be concerned with the latter case.

Definition 2. *An* authorization response *for request* (q, i) *is a tuple* (r, i, E, d)*, where* r *is a response identifier,* d *is a decision and* E *is a list of response identifiers.*

To maintain the generality of SAAM, the decision element of the response tuple may take a number of different values apart from the standard responses of `allow` and `deny`. In particular, a decision may return an `undecided` response to indicate that a response could not be computed.

Definition 3. *If a response has the form* $(r, i, [\,], d)$*, where* $[\,]$ *denotes the empty list, then we say that it is a* primary response*. Otherwise, we say that* (r, i, E, d) *is a* secondary response*.*

A primary response is made by the PDP, and is based directly on the access control policy, which we call *primary* authorization information (namely, the access control policy). Other responses are made by the PEP, based on earlier requests and decisions. The list of response identifiers E forms the *evidence* that was used to make the decision. Secondary responses are said to be based on *secondary* authorization information (namely, the cached responses and the corresponding requests).

Definition 4. *Let* (q, i) *be a request. Then the response* (r, i, E, d) *is* precise *if either the response is primary, or it is secondary and there exists an equivalent request* (q, i') *with response* $(r', i', [\,], d)$*. In the former case* $E = [\,]$*, and in the latter* $E = [r']$*. A response is* approximate *if it is not precise.*

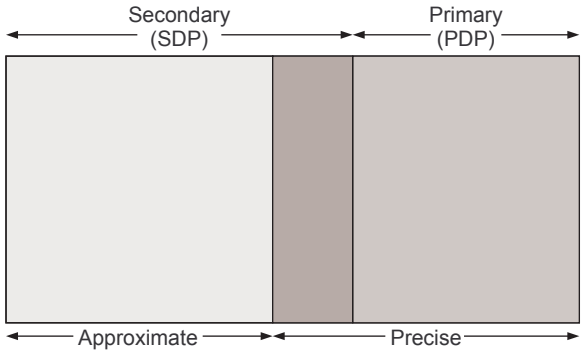
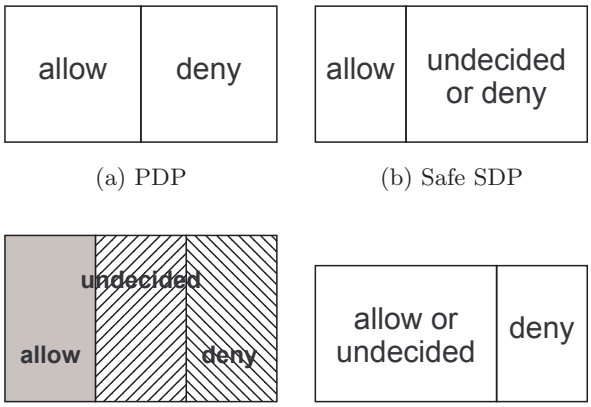


Figure 2: Relationships between different response types

Figure 2 illustrates the relationships between different types of responses. The set of primary responses, for example, is a subset of the set of precise responses. The intersection of precise and secondary responses, the middle section of the diagram, represents the set of responses made by the PEP for which there is a cached response to an equivalent request. These are the responses that the state-of-the-practice PEPs provide. The left-most part of the diagram indicates the set of approximate responses. It is this set of responses that will be the focus of the remainder of the paper.

2.3 Secondary Decision Point

When the subject had not previously made an equivalent request, the PEP is unable to compute a precise response. We must therefore define what information is required to make an approximate response that is consistent with the underlying access control policy. A PEP component that implements SAAM algorithms for computing secondary responses is called a *secondary decision point* (SDP).



(a) PDP (b) Safe SDP

(c) Safe and consistent SDP (d) Consistent SDP

Figure 3: Response sets of a PDP and the corresponding safe and consistent SDPs

In SAAM, we define two basic types of SDPs according to the properties of their response spaces in relation to the PDP responses.

DEFINITION 5. *We say that an SDP is* safe *if any request it allows would also be allowed by the PDP. We say that an SDP is* consistent *if any request it denies would also be denied by the PDP.*

A safe SDP returns either `undecided` or `deny` for any request for which it cannot infer an `allow` response. A safe SDP can be configured or designed to implement a *closed world policy* by simply denying any request that it cannot evaluate. In other words, such as SDP would "fail safe" – one of the important principles identified by Saltzer and Schroeder in their seminal paper [24]. The SAAM_{BLP} SDP described in Section 3 is safe. In general, we would wish to implement a safe and consistent SDP. Such an SDP returns the same response as the PDP would have for any request that it can evaluate. However, the limitations of the underlying access control policy, time or space complexity of the inference algorithms, or business requirements could limit an SDP implementation to being either safe or consistent, but not both.

2.4 On the Strategy for Computing Approximate Authorizations

The most important consideration when constructing an SDP is to decide on the appropriate *response strategy* when faced with incomplete information. In other words: What relationship should an approximate response have to the precise response for the same request? Speaking in general terms, this strategy has to make various assumptions about the structure of the space of either subjects, object, access rights, or any combination of the above. It might also have to make assumptions about the structure of the access constraints defined by the access control policy in question (e.g., simple security property and *-property in BLP). In this section, we discuss the thinking behind our approach to this strategy. In Section 3, we present a specific strategy for BLP.

We assume that the authorization policy is not readily available to the SDP. In other words, the secondary response is based on the set of cached primary responses and any information that can be deduced from an application request and the system environment. In the simplest case, we can consider two requests (q, i) and (q', i') such that if q had previously been allowed, then q' should also be allowed, and if q had previously been denied, then q' should also be denied.

Let us assume, for example, that `r` denotes generic read access and `w` denotes generic write access, where `w` is implemented as allowing read and write access (as in the Bell-LaPadula model). Then the request $(s, o, \mathtt{r}, c, i')$ can be granted if the request $(s, o, \mathtt{w}, c, i)$ has previously been granted. Conversely, the request $(s, o, \mathtt{w}, c, i')$ should be denied if the request $(s, o, \mathtt{r}, c, i)$ has previously been denied. In this case, the ability to make an approximate response is based on the structure of the access-right space, namely the fact that `w` "implies" `r`.

Alternatively, it might be that the SDP is provided with a summary or "digest" of the authorization policy. This digest could simply provide an ordering on the set of subjects derived from the information in the authorization policy.[2] Specifically, we define an ordering on the set of subjects such that $s \geqslant s'$ if the set of requests authorized for s is a superset of the requests authorized for s'.[3] Then the request (s', o, a, c, i') is allowed if the request (s, o, a, c, i) has previously been allowed and $s' \geqslant s$. Conversely, (s', o, a, c, i') is denied if (s, o, a, c, i) has been denied and $s \geqslant s'$.

We can summarize these observations by writing $q \Rightarrow q'$ to denote that a decision to grant q determines whether to grant q' (and hence a decision to deny q' determines whether to deny q). Then the request (q', i') is granted if a previous request (q, i) exists such that $q \Rightarrow q'$ with a cached response $(r, i, E, \mathtt{allow})$, and is denied if a request $(q, i,)$ exists such that $q' \Rightarrow q$ with a cached response $(r, i, E, \mathtt{deny})$.

Clearly, 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 simulator of the PDP. If it is a safe SDP, the number of false negatives will decrease over time; if it is a consistent SDP, the number of false positives will decrease over time.

In summary, in order to compute an approximate authorization, we need to cache primary authorizations and we need some method for inferring whether and how each cached response is applicable to the current request. The notion of "applicability" depends on the implementation, the underlying access control policy, and the additional information that is available to the SDP. An important aspect of future research will be to determine "what works" in terms of the information that can be usefully and efficiently stored by the SDP (in addition to the cached responses). In the next section, we examine how approximate authorization works when the underlying access control policy is based on an information flow policy for confidentiality, as implemented in the Bell-LaPadula model.

[2]This would be particularly easy for RBAC systems.

[3]Note that this does not define a partial ordering on the set of subjects as it is not anti-symmetric: that is, $s \leqslant s'$ and $s' \leqslant s$ does not imply that $s = s'$.

3. SAAM$_{\text{BLP}}$

This section describes the construction of an SDP for an information flow policy for confidentiality, which is used in the Bell-LaPadula (BLP) model [1, 2]. The information flow policy forms the mandatory part of the model and defines the following sets and functions:

- a set of subjects S and a set of objects O;
- a lattice of security labels L;[4]
- a security function λ: $S \cup O \rightarrow L$.

The *simple security property* permits a subject s to read an object o if $\lambda(s) \geqslant \lambda(o)$; the **-property* permits a subject s to write to an object o if $\lambda(s) \leqslant \lambda(o)$. The BLP model identifies three generic access rights to which these security properties apply: `read`, which is a read-only action; `append`, which is a write-only action; and `write`, which is a read-write action. Hence, the request $(s, o, \texttt{write}, c, i)$ is only granted if $\lambda(s) = \lambda(o)$.

In the full BLP model, a request must be authorized by an access matrix (in addition to satisfying the simple security and *-properties). In this paper, we focus on the mandatory part of the model.

We now describe how we can derive approximate authorization responses for the BLP model. We distinguish between three different scenarios:

1. Each SDP contains a copy of the security lattice. The $\lambda(o)$ is locally available to the SDP. The $\lambda(s)$ is either part of the security context of the application request (i.e., it is "pushed" from the client to the server) or is also locally available to the SDP.
2. Each SDP contains a copy of the security lattice. Primary responses (from the PDP) for a request of the form (s, o, a, c, i) contain $\lambda(s)$ and $\lambda(o)$.
3. Primary responses contain no information about the security level of the entities in the request.

To decide requests in the first scenario, the SDP looks up the security labels for both subject and object and compares them using the lattice. Clearly, no involvement of the PDP is required for the SDP to compute a precise response. This case is common among systems in which (1) $\lambda(o)$ is part of the object meta-data, and (2) subject credentials are "pushed" from the client's security subsystem to that of the server. An example of systems that support "pushing" subject credentials is CORBA [21]. However, not all systems use the "push" approach for credentials, due to the limitations of the underlying security protocols, the communication technologies, or the administrative constraints.

In the second scenario, the SDP can decide a new request, provided both subject and object have been involved in earlier requests. In particular, the SDP searches its cache for requests involving the two entities and obtains their respective security labels. The SDP can then use the security lattice to determine whether one label dominates the other and hence generate an appropriate response to the request. Clearly, the first two cases are simple; we therefore focus on the more challenging third case.

[4]Strictly speaking, the BLP model requires L to have the form $C \times 2^K$, where C is a linearly ordered set of security classifications and K is a set of needs-to-know categories.

A naïve solution to the third scenario would be to provide each SDP with the security lattice and a complete set of (o, $\lambda(o)$) (s, $\lambda(s)$) tuples, essentially combining the PEP and PDP. Although such an approach is straightforward, it does not scale with large populations of subjects and objects, and voids the benefits of decoupling PDPs and PEPs, namely consistent policy enforcement across multiple PEPs and reduced administration of authorization policy. In this section, we describe an approach that does not have these drawbacks.

Intuitively, responses to requests enable us to infer information about the relative ordering on security labels associated with subjects and objects. If, for example, the requests $(s, o, \texttt{read})$ and $(s', o, \texttt{append})$ are allowed by the PDP, then we can infer that $\lambda(s) \geqslant \lambda(o) \geqslant \lambda(s')$. If, in addition, the request $(s, o, \texttt{append})$ is denied, then we can infer that $\lambda(s) > \lambda(o)$.

To record the relative ordering on subject and object security labels, we build a data structure, called a *dominance graph*, from authorization responses made by the PDP. Each node in the dominance graph represents a set of entities with the same security label, while each edge records information about the ordering of entities. The dominance graph represents partial knowledge about the security lattice and the mapping of entities to labels in that lattice. We now describe the dominance graph more formally, its construction, and how it can be used to generate approximate responses.

3.1 The Dominance Graph

The dominance graph is constructed from the set of responses made by the PDP. It is used to encode information that can be inferred from those responses. Let Q denote the set of requests for which primary responses exist, and let R denote the set of corresponding responses. Let $S(R)$ denote the set of subjects that appear in requests in Q and let $O(R)$ be defined analogously for objects.

Definition 6. *Given a set of responses R, the* dominance graph $G(R) = (V(R), E(R))$ *is a directed acyclic graph with the following properties:*

- *For each vertex $v \in V(R)$, $v \subseteq S(R) \cup O(R)$ and for all $x, y \in v$, $\lambda(x) = \lambda(y)$;*
- *For each edge $(x, y) \in E(R)$, a response exists in R that implies $\lambda(x) \geqslant \lambda(y)$.*

When R is obvious from the context, we simply write $G = (V, E)$ for the dominance graph, S for $S(R)$, and O for $O(R)$. We write E^* to denote the transitive closure of the binary relation E. In other words, $(v, v') \in E^*$ iff there is a (directed) path from v to v' in G. Examples of dominance graphs are shown in Figure 6.

There is no ambiguity in referring to the security label of a node in the graph. By definition, each entity associated with a particular node in the dominance graph has the same security label. Henceforth, we will write $\lambda(v)$ to signify the security label of all the entities contained in node v of the dominance graph.

Note, however, that two different nodes may have the same security label. In the simplest case, where $Q = \{(s, o, \texttt{read}, c, i)\}$ and $R = \{(r, i, [\,], \texttt{allow})\}$, we can only infer that $\lambda(s) \geqslant \lambda(o)$. Hence, the dominance graph would contain two nodes, one containing s and one containing o,

and a single edge from the node containing s to the node containing o. It may be that $\lambda(s) = \lambda(o)$, but we would require additional requests and responses to infer this fact. In particular, if the PDP granted the request $(s, o, \texttt{append}, c, i')$, we could infer that $\lambda(s) \leqslant \lambda(o)$ and hence create a loop in the dominance graph, which can be collapsed to a single node containing s and o.

Moreover, a path in the dominance graph, and the transitivity of the partial order, means that we can use the dominance graph to make approximate responses. In particular, if $s \in v$ and $o \in v'$ and there is a path from v to v' in the dominance graph (that is, $(v, v') \in E^*$), then we can deduce that $\lambda(s) \geqslant \lambda(o)$ and that a request from s to read o should be granted.

Note that the dominance graph is a faithful (though incomplete) representation of the security lattice L in the sense that if $l \leqslant l'$ in L, $v, v' \in V$, $\lambda(v) = l$ and $\lambda(v') = l'$, then $(v', v) \in E^*$. As more primary responses are added to R, more edges can be added to the graph, and more cycles will be created in the dominance graph; this lead, after cycle collapsing, to fewer nodes, each containing more entities. Over time, therefore, the dominance graph will tend to resemble the security lattice.

3.2 Constructing the Dominance Graph

In this section, we present two algorithms: the first is for adding an edge to the dominance graph given a response from the PDP that allows a new request (s, o, a, c, i); the second is for coalescing multiple nodes in the graph into a single node. It is used when a cycle is created within the dominance graph and two or more nodes are collapsed into a single node in which all the entities have the same security level.

Note that a `deny` response from the PDP does not allow enough information to be inferred about the ordering between two entities to add an edge to the dominance graph. Hence, in the first algorithm we only consider `allow` responses from the PDP. Making use of `deny` responses is the subject of ongoing work.

3.2.1 Adding New Responses

When the PDP allows a request (s, o, a, c, i), we can add further information to the dominance graph. In particular, we may add a new node for s or o if they do not already exist in the graph, and we may add an edge from s to o if a involves reading o (a is `read` or `write`), or from o to s if a involves writing to o (a is `append` or `write`). Figure 4 shows pseudo-code for the *AddAllowResponse* algorithm, which takes the current dominance graph and the request (s, o, a, c, i) as parameters, and constructs a new dominance graph.

The algorithm first adds s and o to the graph if they are not already associated with one of the nodes (lines 02–07). It then adds a new edge to the graph if there is no path between the entities (lines 10–11 and 17–18). The edge is directed from s to o if the requested right includes reading o (line 12), and is directed from o to s if the requested right includes appending to o (line 19).

Notice that the algorithm *CollapseCycles* is used whenever a cycle would be created in the dominance graph by the addition of an edge (lines 13–14 and 20–21). We describe this algorithm in the next section.

```
 1: AddAllowResponse(G, (s, o, a, c, i))
 2: if s is not in any node in G then
 3:    add s to G
 4: end if
 5: if o is not in any node in G then
 6:    add o to G
 7: end if
 8: let v_s denote node containing s
 9: let v_o denote node containing o
10: if a ∈ {read, write} and (v_s, v_o) ∉ E* then
11:    if (v_o, v_s) ∉ E* then
12:       add an edge between v_s and v_o
13:    else
14:       G = CollapseCycles(G, v_o, v_s)
15:    end if
16: end if
17: if a ∈ {append, write} and (v_o, v_s) ∉ E* then
18:    if (v_s, v_o) ∉ E* then
19:       add an edge between v_o and v_s
20:    else
21:       G = CollapseCycles(G, v_s, v_o)
22:    end if
23: end if
```

Figure 4: Adding a response to the dominance graph

3.2.2 Removing Cycles

The addition of an edge to the dominance graph may introduce a cycle in the graph. For any cycle comprising the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)$ we have, by definition of the dominance graph,

$$\lambda(v_1) \leqslant \lambda(v_2) \leqslant \cdots \leqslant \lambda(v_n) \leqslant \lambda(v_1).$$

By transitivity, we deduce that

$$\lambda(v_1) = \lambda(v_2) = \cdots = \lambda(v_n).$$

Hence, our algorithm removes cycles from the dominance graph by reducing the set of nodes in the cycle to a single node.

A pseudo-code implementation of this algorithm is shown in Figure 5. The algorithm takes the dominance graph G, a start node v_{start} and an end node v_{end} as parameters. The first stage of the algorithm (line 02) is to compute the subgraph G' comprising all paths from v_{start} to v_{end}. The intuition is that *CollapseCyles* is called if a response means that we could infer that the edge (v_{end}, v_{start}) should be added to the graph. Hence, the existence of a path from v_{start} to v_{end} implies the existence of a cycle in the dominance graph. As a result of the algorithm execution, G' is replaced by the single node v_{start}. We then compute all nodes outside G' that are neighbors of some node in G' (lines 03–04). These nodes are re-connected to v_{start}, the node that will replace G' (lines 10–15). We also re-assign all subjects and objects associated with each node in G' to v_{start} (lines 05–09). Finally, we prune all the redundant edges and nodes from the dominance graph (lines 19–20).

When the set of cached responses is small, the dominance graph will contain many nodes, as there will be insufficient information to infer that several entities have the same security level. Over time, the number of cached responses will grow and cycles will emerge and be collapsed into single nodes, reducing the number of nodes in the dominance

```
1: CollapseCycles(G, v_start, v_end)
2: let G' = (V', E') be subgraph comprising all paths from
   v_start to v_end
3: let InEdges = {(v, v') ∈ E : v ∈ V \ V', v' ∈ G'}
4: let OutEdges = {(v', v) ∈ E : v ∈ V \ V', v' ∈ G'}
5: for all v' ∈ V' do
6:    for all x ∈ v' do
7:       add x to v_start
8:    end for
9: end for
10: for all (v, v') ∈ InEdges do
11:    add (v, v_start) to E
12: end for
13: for all (v', v) ∈ OutEdges do
14:    add (v_start, v) to E
15: end for
16: delete all nodes in V' except for v_start
17: delete all edges in InEdges and OutEdges
```

Figure 5: Removing cycles from the dominance graph

graph. The dominance graph will eventually become identical to the security lattice and each node will be labelled with precisely the set of entities that have the corresponding security label.[5]

3.2.3 Example

We now illustrate how a simple dominance graph develops as responses are generated by the PDP. We will assume that the security lattice L is simply a linear ordering with three elements (say, Low < Medium < High) and that the set of needs-to-know categories is empty. For the sake of readability, we will omit the context variable in these requests and put the request identifier in the right column. Let us assume that the following requests are all allowed by the PDP:

$(s_1, o_1, \texttt{read})$,	(1)	$(s_4, o_2, \texttt{append})$,	(6)
$(s_2, o_1, \texttt{append})$,	(2)	$(s_4, o_3, \texttt{read})$,	(7)
$(s_3, o_2, \texttt{read})$,	(3)	$(s_4, o_4, \texttt{read})$,	(8)
$(s_3, o_1, \texttt{write})$,	(4)	$(s_3, o_3, \texttt{write})$,	(9)
$(s_1, o_2, \texttt{read})$,	(5)	$(s_2, o_4, \texttt{write})$.	(10)

The evolution of the dominance graph is shown in Figure 6. We note the following features of this diagram.

- Request (1) causes the first two nodes to be created in the dominance graph; the arrow is directed from the subject to the object because it is a `read` request.
- Request (2) causes one new node to be added to the graph, because the requested object was also used in the previous request.
- Request (3) causes two further nodes to be created in the graph, because neither the requesting subject nor the requested object are stored in the cache.
- Request (4) causes two edges to be added to the graph, creating a cycle (because `write` in BLP is `append` and `read`). As a result, s_3 and o_1 are associated with the same node.
- Request (5) has no effect on the dominance graph, because a path already exists between s_1 and o_2 in the graph.
- Requests (6), (7), and (8) simply increase the height of the dominance graph.
- Request (9) creates a cycle involving four nodes in the graph, which are collapsed onto a single node containing o_1, s_3, o_2, s_4, and o_3.
- Finally, request (10), for performing generic `write`, creates a cycle between s_2, and o_4, causing them to be collapsed into a single node.

At this point, the dominance graph is identical to the security lattice.

3.3 Generating Approximate Responses

Given a request (s, o, a, c, i), we can use the dominance graph to compute an approximate response. If a is `read`, then we allow the request if a path exists from the node containing s to the node containing o in the dominance graph. If a is `append`, then we allow the request if there is a path from the node containing o to the node containing s in the dominance graph. Finally, if a is `write`, then we allow the request if s and o belong to the same node in the dominance graph.

```
1: EvaluateRequest(G, (s, o, a, c, i), l)
2: let v_s be the node containing s
3: let v_o be the node containing o
4: if a = write then
5:    if v_s = v_o then
6:       return allow
7:    else
8:       return undecided
9:    end if
10: end if
11: if a = read then
12:    if there is a path of length ⩽ l from v_s to v_o then
13:       return allow
14:    else
15:       return undecided
16:    end if
17: end if
18: if a = append then
19:    if there is a path of length ⩽ l from v_o to v_s then
20:       return allow
21:    else
22:       return undecided
23:    end if
24: end if
```

Figure 7: Computing an approximate response

Note that this decision process may yield "false negatives", in the sense that our algorithm may return an `undecided` response for those requests that the PDP would

[5]Formally speaking, we can define a binary relation $\sim$ on $S \cup O$, where $x \sim y$ iff $\lambda(x) = \lambda(y)$. It is easy to prove that $\sim$ is an equivalence relation, and hence that $S \cup O$ is partitioned into equivalence classes. Each equivalence class is a node in the dominance graph.

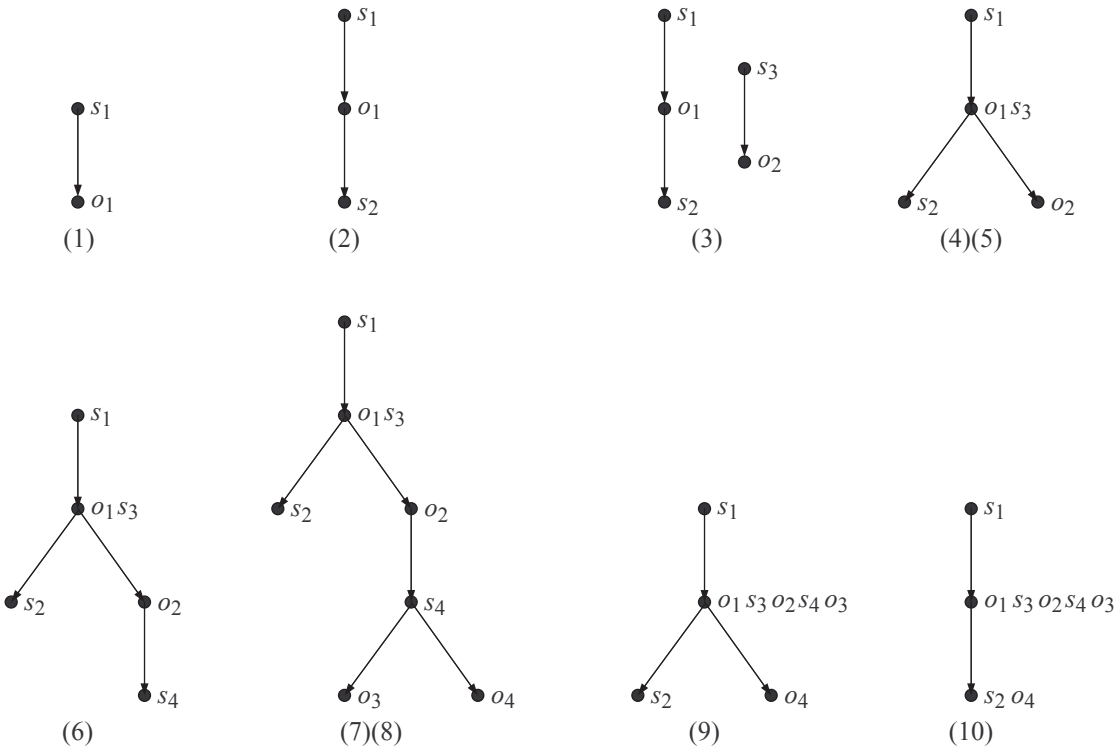


Figure 6: Evolution of the dominance graph

allow. In other words, the *EvaluateRequest* algorithm implements a safe SDP.[6]

We can limit the scope of the search for paths in the dominance graph by specifying a maximal length l for the paths that can be traversed in trying to decide the request. This has the effect of reducing the search space within the dominance graph, thereby reducing the time taken by the SDP to compute an approximate response. Clearly, this also reduces the possibility of finding a suitable path in the dominance graph, thereby increasing the number of false negatives.

One obvious choice of l can be derived from the security lattice L. In particular, note that L is finite, hence the Hasse diagram[7] of L contains a path of maximal length l_{max}. In other words, one strategy would be to terminate the search for an approximate response if we do not find a path of length less than or equal to l_{max} between the nodes containing the relevant entities in the dominance graph.

The algorithm used to compute an approximate response is shown in Figure 7.

[6]It is less easy to develop an algorithm that implements a consistent SDP. One might imagine that such an algorithm would deny a read request, for example, if there is a path from the node containing the object to the node containing the subject. However, this path includes the possibility that the security label of the object and subject are equal, in which case the PDP would allow the request and the SDP would deny the request. In other words, the SDP is not consistent, as it has denied a request that the PDP would allow.

[7]The *Hasse diagram* of a lattice is the graph of the reflexive, transitive reduction of the partial order relation [5].

3.4 Complexity Analysis

In this section, we provide further analysis of SAAM_{BLP}. Full details are available in the associated technical report [16].

We will write n to denote the number of responses from the PDP in the cache (and we will assume that this is equal to the number of requests). The number of edges in the dominance graph is bounded by n (since a response can add at most one edge to the graph) and the number of nodes is bounded by $2n$ (since a single response can add at most two entities to the dominance graph). We will assume that we cache responses in such a way that we can find a subject or object in time $\mathcal{O}(\log n)$ (using hash tables [11], for example). We will also assume the use of a breadth-first search algorithm (BFS) that can traverse the dominance graph in time proportional to the sum of the numbers of edges and nodes in the graph; that is, in time $\mathcal{O}(n)$.

3.4.1 Collapsing Cycles in the Dominance Graph

The algorithm that processes new responses from the PDP calls the *CollapseCycles* algorithm, so we consider the time complexity of this first. This algorithm contains the following steps:

- compute all the paths from v_{start} to v_{end} (line 02), which has complexity $\mathcal{O}(n)$ (using BFS);
- move subjects and objects to v_{start} (lines 05–09), which has complexity $\mathcal{O}(n)$ (since there may be $\mathcal{O}(n)$ entities in the cycle);
- create new edges connected to v_{start} (lines 10–15),

which has complexity $\mathcal{O}(n)$ (since there may be $\mathcal{O}(n)$ edges to be added).

Hence, the overall time taken to collapse cycles in the dominance graph is $\mathcal{O}(n)$.

3.4.2 Processing a PDP Response

The main steps in adding a response to the request (s, o, a, c, i) (as shown in Figure 4) are to:

- test whether s is already associated with a node in the graph (line 02), which has complexity $\mathcal{O}(\log n)$;
- test whether o is already associated with a node in the graph (line 05), which has complexity $\mathcal{O}(\log n)$;
- test whether a path between the nodes containing s and o exists (lines 10 and 14), which has complexity $\mathcal{O}(n)$;
- collapse any cycles that are created in the graph (lines 14 and 21), which has complexity $\mathcal{O}(n)$.

Hence the total time to add new information from a PDP response to the dominance graph is $\mathcal{O}(n)$.

3.4.3 Generating an SDP Response

The main steps required to generate an SDP response are to:

- compute the node(s) containing the subject and object (lines 02–03), which has complexity $\mathcal{O}(\log n)$;
- compute whether there is a (possibly trivial) path between the two nodes (lines 04–24), which has complexity $\mathcal{O}(n)$.

The overall complexity of computing an SDP response is therefore $\mathcal{O}(n)$. (As we discuss in Section 3.3, one way of controlling the time taken to evaluate a request is to limit the length of paths that are traversed in the dominance graph when executing *EvaluateRequest*.)

4. RELATED WORK

SAAM is a conceptual framework that is implemented by an SDP and used to compute approximate responses to authorization requests. These heuristics provide a replacement for conventional authorization responses from the PDP, when the PDP is unavailable. SAAM assumes that PDP responses will be cached and used to infer approximate responses.

Caching authorization responses is not a new idea in the domain of access control: it has been used to improve system efficiency and reliability [3, 8, 13, 19, 28]. However, these and other approaches only compute precise authorizations and are therefore only effective for resolving repeated requests. SAAM can resolve new requests by extending the space of supported responses to approximate ones. In other words, SAAM provides a richer alternative source for authorization responses than do existing approaches. It is unique in offering a systematic approach to authorization recycling by suggesting a generic model of authorization requests, and responses, as well as providing response classification and strategies.

Other work [18, 22, 23] exploits the relationships between (database) objects to improve system scalability, while SAAM_{BLP} uses the relationships between subjects and objects to improve system reliability and reduce latency. This work also infers authorizations from existing ones, but ours is the first to re-use previous responses to infer information about the underlying access control policy, and to make approximate responses that are consistent with that policy.

SAAM is a domain-specific approach to improving performance and fault tolerance of those access control mechanisms that employ remote authorization servers. Three general classes of fault tolerance solutions are failure masking through information redundancy (e.g., error correction checksums), time redundancy (e.g., repetitive invocations), or physical redundancy (e.g., data replication). SAAM employs physical redundancy [10]: when the PDP is unavailable, the SDP would be able to mask the fault by providing the requested access control decision. However, general-purpose physical redundancy techniques for distributed systems scale poorly beyond a small number of systems [9], and become technically and economically infeasible when the scale reaches the thousands [27]. Unlike such techniques, our approach requires no specialized software, simply modifications to the logic of the PEP cache. No distributed state, election, or synchronization algorithms are necessary either. With SAAM, only authorization responses are cached, and no dynamic authorization data are replicated, enabling linear scalability on the number of PEPs and PDPs.

5. CONCLUSIONS

We have developed a simulation testbed to evaluate the utility of SAAM_{BLP} and used it to generate a random test set of authorization requests. We then measured the proportions of requests resolved by a conventional PEP that recycles only precise responses and the SDP described in Section 3. Preliminary results demonstrate that when 10% of authorizations are cached, our SDP can evaluate over 30% more authorization requests than a conventional PEP. Interestingly, these results hold even with small numbers of 1000 objects and 100 subjects in a system with a BLP policy defined with a 14-node lattice. We are currently conducting further experiments and will report on our results in detail in the near future. Results of our evaluation indicate that active recycling of precise and approximate authorizations could be a viable alternative to general purpose techniques for improving fault tolerance and reducing delays associated with access control.

To the best of our knowledge, the work described in this paper is the first attempt to formulate a general application-independent framework for computing new authorization responses using previous ones. While our approach is not necessarily appropriate for small-scale access control solutions, it is expected to improve reliability and performance of applications in those enterprises that have a high cumulative rate of partial fail-stop failures, or high communication overhead due to widespread distribution of systems.

The rate of observed failures can be decreased significantly even with a small (10%) proportion of authorizations cached. On the other hand, the reduction of the observed performance overhead is the subject of the trade-off between the delay of PEP-PDP communication and the cost of computing approximate authorizations. However, both the failure rate and the performance overhead reductions are limited by the availability and the performance, respectively, of the

application as well as the network connecting the client to the application. For example, if the application itself is very slow (or the client-server connection is very unreliable), then the overall gain in performance (or availability), as observed by the client, will be small.

We have developed a safe SDP for handling policies based on the Bell-LaPadula model. However, other access control models will likely require different SDPs. Support for role-based access control, time-based policies, history-sensitive policies (e.g., Chinese Wall), dynamic separation of duties, or other types of policies that require subject rights to be consumable (e.g., task-based authorizations [26]) has yet to be developed. Future work will investigate support for the above models as well as the use of subject, object, and access-right space structures.

Acknowledgements

The authors would like to thank Liang Chen for his careful reading of an earlier draft of the paper, Kyle Zeeuwen for doing an evaluation study of the SDP described in Section 3, and Craig Wilson for improving the readability of the paper.

6. REFERENCES

[1] Bell, D., and LaPadula, L. Secure computer systems: Mathematical foundations. Tech. Rep. MTR-2547, Volume I, Mitre Corporation, Bedford, Massachusetts, 1973.

[2] Bell, D., and LaPadula, L. Secure computer systems: A mathematical model. Tech. Rep. MTR-2547, Volume II, Mitre Corporation, Bedford, Massachusetts, 1973.

[3] Borders, K., Zhao, X., and Prakash, A. CPOL: High-performance policy evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), pp. 147–157.

[4] Brown, N., and Kindel, C. Distributed component object model protocol (DCOM/1.0), January 1998.

[5] Davey, B., and Priestley, H. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, United Kingdom, 1990.

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

[7] Eddon, G. The COM+ security model gets you out of the security programming business. *Microsoft Systems Journal 1999*, 11 (1999).

[8] Entrust. *GetAccess Design and Administration Guide*, September 20 1999.

[9] Jimenez-Peris, R., Patino-Martinez, M., Kemme, B., and Alonso, G. Improving the scalability of fault-tolerance database cluster. In *Proceedings of the 22nd International Conference on Distributed Computing Systems* (2002), pp. 477–486.

[10] Johnson, B. W. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996, ch. An introduction to the design and analysis of fault-tolerant systems, pp. 1–87.

[11] Johnsonbaugh, R., and Schaefer, M. *Algorithms*. Pearson Education, Inc., 2001.

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

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

[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *Proceedings of the 1997 11th European Conference on Object-Oriented Programming, ECOOP, Jun 9-13 1997* (Jyvaskyla, Finl, 1997), vol. 1241 of *Lecture Notes in Computer Science*, pp. 220–242.

[15] Kong, M. M. DCE: An environment for secure client/server computing. *Hewlett-Packard Journal 46*, 6 (1995), 6–15.

[16] Leung, W., Crampton, J., and Beznosov, K. Authorization recycling in BLP systems. Tech. Rep. 2005-11-11, University of British Columbia, Vancouver, Canada, 2005.

[17] Meier, J., Mackman, A., Dunner, M., and Vasireddy, S. *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication*. Microsoft Press, 2002.

[18] Motro, R. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *Proceedings of ICDE* (1989), pp. 339–347.

[19] Netegrity. *SiteMinder Concepts Guide*, 2000.

[20] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*, 2005. OASIS Committee Specification (T. Moses, editor).

[21] OMG. *CORBA Services: Common Object Services Specification, Security Service Specification v1.8*, 2002.

[22] Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P. Extending query rewriting techniques for fine-grained access control. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (2004), pp. 551–562.

[23] Rosenthal, A., and Sciore, E. Administering permissions for distributed data: Factoring and automated inference. In *Proceedings of the 15th Annual Working Conference on Database and Application Security* (2001), pp. 91–104.

[24] Saltzer, J., and Schroeder, M. The protection of information in computer systems. *Proceedings of the IEEE 36*, 9 (1975), 1278–1308.

[25] Securant. *Unified Access Management: A Model For Integrated Web Security*, 25 June 1999.

[26] Thomas, R., and Sandhu, R. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management. In *Database Securty XI: Status and Prospects. Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Security* (1997), pp. 166–181.

[27] Vogels, W. How wrong can you be? Getting lost on the road to massive scalability. In *Middleware 2004* (2004). Keynote address.

[28] Wimmer, M., and Kemper, A. An authorization framework for sharing data in web service federations. In *Proceedings of 2nd VLDB Workshop on Secure Data Management* (2005), pp. 47–62.