The following paper was originally published in the
Proceedings of the 7th USENIX Security Symposium
San Antonio, Texas, January 26-29, 1998

Unified Support for Heterogeneous Security Policies in Distributed Systems

Naftaly H. Minsky and Victoria Ungureanu
Rutgers University

Unified Support for Heterogeneous Security Policies in Distributed Systems

Naftaly H. Minsky[*] Victoria Ungureanu[*]

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
$\{minsky, ungurean\}@cs.rutgers.edu$

Abstract

Modern distributed systems tend to be conglomerates of heterogeneous subsystems, which have been designed separately, by different people, with little, if any, knowledge of each other — and which may be governed by *different security policies*. A single software agent operating within such a system may find itself interacting with, or even belonging to, several subsystems, and thus be subject to several disparate policies. If every such policy is expressed by means of a different formalism and enforced with a different mechanism, the situation can get easily out of hand.

To deal with this problem we propose in this paper a security mechanism that can support efficiently, and in a *unified* manner, a wide range of security models and policies, including: conventional *discretionary* models that use capabilities or access-control lists, *mandatory* lattice-based access control models, and the more sophisticated models and policies required for commercial applications. Moreover, under the proposed mechanism, a single agent may be involved in several different modes of interactions that are subject to disparate security policies.

1 Introduction

Modern distributed systems tend to be conglomerates of heterogeneous subsystems, which have been designed separately, by different people, with little, if any, knowledge of each other — and which may be *governed by different security policies.* A single software agent operating within such a system may find itself interacting with, or even belonging to, several subsystems, and thus be subject to several disparate policies. For example, an agent may be subject to a multi-level security policy when retrieving military documents; it may carry *capabilities* that provide it with certain access rights to computing resources; and, while accessing certain financial information, it may be subject to the "Chinese Wall" security policy [4]. If every such policy is expressed by means of a different formalism and enforced with a different mechanism, the situation can get easily out of hand.

To deal with this problem we propose in this paper a security scheme under which policies are defined formally and explicitly, and are enforced by a unified mechanism. Each policy under this scheme specifies the type of messages regulated by it and the *law* that governs these messages.

The proposed mechanism is based on the concept of "law-governed architecture" for distributed systems [13], and on the more recent concept of "regulated interaction" (RI) [14]; it is currently implemented by an experimental toolkit called Moses. This toolkit can support a wide range of security models and policies, including: conventional *discretionary* models that use capabilities or access-control lists, *mandatory* lattice-based access control models [18], and the more sophisticated models and policies required for commercial [5] and clinical [1] applications. Moreover, under Moses, a single agent may be involved in several different modes of interactions that are subject to disparate security policies.

[*] Work partially supported by NSF grants No. CCR-96-26577

The paper is organized as follows: Section 2 attempts to motivate the need for a unified mechanism by considering two security policies that are difficult to support by conventional mechanisms, and which may need to coexist in a single system. Section 3 defines the concept of a security policy under RI, and shows how such policies are defined and enforced. Section 4 presents the detailed implementation under RI of the policies of Sections 2. Sections 5 discusses some related work, and Section 6 concludes this paper.

2 Motivating Examples

To motivate the need for a new unified security mechanism we describe here two policies designed for commercial applications which are difficult to implement by traditional means, particularly in distributed systems; another such policy, involving capabilities, is discussed in Section 3.4. Later we will present the implementation of these policies under the security mechanism to be proposed here, making it evident that a single agent can be subject, concurrently, to several policies.

2.1 The Chinese Wall ($\mathcal{CW}$) Policy

Consider a distributed database that contains files belonging to various commercial companies. Let these companies be partitioned into a set of disjoint "competition cliques," where each clique contains companies that compete with each other in the market place. And let the clients of this database be financial analysts, who may have access to any number of competition cliques. According to a common commercial practice, such access is subject to the *Chinese Wall* ($\mathcal{CW}$) policy [4] which can be stated as follows:

> *A priori, an analyst can get information about* any *company of a clique* q *to which he has access. But once the analyst gets information about a given company in* q*, he is not allowed to get information about any other company in this clique.*

Thus, under this policy, what the analyst can get from the database, at a given moment in time, depends on what he got from the database in the past. Recently, it has been shown how this policy can be implemented in MLS systems by casting it as a multilevel lattice based relabel policy [8], or by using reflexive-flow relations [7]. However, MLS does not lend itself to distributed implementation, where the files of companies in a given clique are maintained by several servers belonging to possible different administrative domains. Moreover, it is very difficult in this case (if at all possible) to implement this policy either by means of *access control lists* (ACL), or by the *capability-based* scheme — the two main access control techniques used in distributed systems [20]. The implementation of the $\mathcal{CW}$ policy using ACL would require each server to know what, if anything, his client got from other servers in the past, or even what he is requesting from them concurrently. This cannot be done in a scalable way, since it requires multi-casting of all queries. The implementation of $\mathcal{CW}$-policy with capabilities would mean that whenever a given client reads one of the files of a company, its capabilities for files of other companies in the same clique should be revoked. But revocation is difficult to carry out by traditional means, in particular because it requires a central authority, and is generally not supported.

2.2 The Sealed-Bid ($\mathcal{SB}$) Auction Policy

A common way for selling artwork or real estate is sealed-bid auction in which secret bids are issued for an advertised item in a predefined time-frame. The security requirements of this process have been studied recently by Franklin and Reiter in [9], and paraphrased here as follows:

1. every auction has a predefined time frame for bidding, no bids can be issued outside of this frame;

2. once a bid is issued it cannot be repudiated; but one can out-bid himself any number of times.

3. the winner is the issuer of the highest bid;

4. the bidders identity are not revealed at any time and to any party, not even to the auctioneer; the same holds true for the sums bided by losing participants.

5. the auctioneer is guaranteed to be able to collect the money from the winning bidder; losing bidders do not forfeit money;

6. an agent can participate concurrently in any number of auctions.

Henceforth we will refer to these collection of requirements for sealed-bid auction as the $\mathcal{SB}$-policy.

Observing that this policy does not lend itself to implementation by any of the traditional security mechanisms, Franklin and Reiter proposed an elegant new technique for it. Their implementation uses a novel cryptographic technique called verifiable signature sharing, which requires replicating the auction servers.

We will show in section 4.2 how this policy can be implemented by the very same mechanism that we use to implement the Chinese Wall policy. Moreover, our implementation does not require the duplication of the auction servers, thus being more efficient.

3 Security Policies Under Regulated Interaction

We start by defining our concept of a security policy. We continue by showing how a policy is defined by what is called a "law", and how it is enforced under Regulated Interaction (RI)—describing as much of RI, and of the Moses[1] toolkit that implements it, as is needed for this purpose. In Section 3.4 we illustrate this mechanism by showing how uncopyable capabilities can be implemented under RI. In Section 3.5 we show how policies are created and maintained; and we conclude with a brief discussion of the fault tolerance and scalability of our mechanism.

3.1 The Concept of a Security Policy

We define a *security policy* $\mathcal{P}$ to be a triple $\langle\mathcal{M},\mathcal{G},\mathcal{L}\rangle$, where

- $\mathcal{M}$ is the set of messages, regulated by $\mathcal{P}$. They are called $\mathcal{P}$-messages.
- $\mathcal{G}$ is a distributed group of *agents*, sometimes called a "policy-group," that are permitted to send and receive $\mathcal{P}$-messages, and thus are the participants in policy $\mathcal{P}$.
- $\mathcal{L}$ is the set of rules regulating the exchange of $\mathcal{P}$-messages between members of group $\mathcal{G}$, called the *law* of this policy. Broadly speaking, the law determines who in group $\mathcal{G}$ can send which $\mathcal{P}$-messages to whom, and what should the effect of such a message be.

For example, the components of the policy for secure-bid auction are as follows: the group $\mathcal{G}_{\mathcal{SB}}$ consists of all agents participating in the auction, including the auctioneers. The set of messages $\mathcal{M}_{\mathcal{SB}}$ consists of all the messages exchanged during the auction including: initiating an auction, issuing a bid, and announcing the results; and the law $\mathcal{L}_{\mathcal{SB}}$ is the set of rules described above for $\mathcal{SB}$, written in a given formal language. We introduce such a language in the following section.

It should be pointed out that we take a policy to have an independent existence, separate from the agents participating in it. We provide means for an agent to *join* a given policy $\mathcal{P}$—subject to the law of this policy—which will enable this agent to send and receive $\mathcal{P}$-messages.

3.2 The Law

A law $\mathcal{L}$ of a policy $\mathcal{P}$ determines the treatment of $\mathcal{P}$-messages is defined by specifying what should be done when such a message is sent, and when it arrives. More specifically, we deal with the following two kinds of events that are regulated under RI[2]:

- `sent(x,m,y)` — occurs when agent `x` sends an $\mathcal{L}$-message `m` addressed to `y`. The receiver `x` is considered the *home* of this event.
- `arrived(x,m,y)` — occurs when an $\mathcal{L}$-message `m` sent by `x` arrives at `y`. The receiver `y` is considered the *home* of this event. If the destination is the keyword `all`, `m` is multicasted to all members of the group. The sender `x` is considered the *home* of this event.

We assume no prior knowledge of, or control over, the occurrence of these *regulated events*. But the

[1] We will use the names RI and Moses somewhat interchangeably.

[2] Note that RI regulates some additional types of events, which are not relevant to security, and are, thus, ignored here.

effect that any given event e would actually have is prescribed by the law $\mathcal{L}$ of the message in question. This prescription, called the *ruling* of the law for this event, is a (possibly empty) sequence of *primitive operations* (discussed later) which are to be carried out at the home of e, as the immediate response to its occurrence.

Structurally, the law $\mathcal{L}$ is a pair $\langle\mathcal{R},\mathcal{CS}\rangle$, where $\mathcal{R}$ is a fixed set of rules defined for the entire group $\mathcal{G}$ of the policy in question, and $\mathcal{CS}$ is a mutable set $\{\mathtt{CS}_x \mid x \in \mathcal{G}\}$ of what we call *control states*, one per member of the group. These two parts of the law are discussed in more detail below.

The control state $\mathtt{CS}_x$: This is the part of the law $\mathcal{L}$ that is associated with the individual member x of a group. It is a bag of terms, called the *attributes* of this member. The main role of these attributes is to enable $\mathcal{L}$ to distinguish between different kinds of members, so that the ruling of the law for a given event may depend on its home. Some of the attributes of an agent have a predefined semantics, such as the attribute self(n) where n represents the name of the member. However, the semantics of attributes for a given group is defined by the law. For instance, in our implementation of the Chinese Wall policy, Section 4.1, an analyst x might have an attribute companyPermit(c), which means that x is allowed to access company's c data.

The Primitive Operations: The operations that can be included in the ruling of the law for a given regulated event e, to be carried out at the home of this event, are called *primitive operations*. They are "primitive" in the sense that they can be performed *only* if thus authorized by the law. These operations include:

- Operations that change the CS of the home agent. Specifically, we can perform the following operations: (1) +t which adds the term t to the control state; (2) -t which removes the term t; (3) t1←t2 which change term t1 with term t2; and (4) incr(t(v),x) which increments the value v of a term t with some quantity x.
- Operation forward(x,m,y) emits to the network the message m addressed to y. (When a message thus forwarded to y arrives, it would trigger at y the event arrived(x,m,y).) The most common use of this operation is in a ruling for event sent(x,m,y), where operation forward (with no arguments) simply completes the passing of the intended message.
- Operation deliver(m) delivers the message m to the home-agent. The most common use of this operation is in a ruling for event arrived(x,m,y), where operation deliver (with no arguments) simply delivers the arriving message to the home agent.

The global set of rules $\mathcal{R}$: The function of $\mathcal{R}$ is to evaluate a ruling for any possible regulated-event that occurs at an agent with a given control-state. In our current model, $\mathcal{R}$ is represented by a very simple Prolog-like program—or, if you will, a set of situation–action rules. When this "program" $\mathcal{R}$ is presented with a goal e, representing a regulated event, and with the control-state of the home agent, it produces a list of primitive-operations representing the ruling of the law for this event. For the details of this formulation the reader is referred to [15]; here we will illustrate it with a detailed example in Section 3.4.

3.3 The Distributed Enforcement Mechanism

The law for a given policy $\mathcal{P}=\langle\mathcal{L},\mathcal{M},\mathcal{G}\rangle$ is enforced in principle as follows: there is a *controller* associated with each member of group $\mathcal{G}$, logically placed between the agent and the communications medium, as it is illustrated in Figure 1. All controllers have identical copies of the global set of rules $\mathcal{R}$ of $\mathcal{L}$, and each controller maintains the control states of the agents under its jurisdiction.

All controllers have identical copies of the global set of rules $\mathcal{R}$ of $\mathcal{L}$, and each controller maintains the control states of the agents under its jurisdiction. The steps taken when a member x wishes to send a $\mathcal{P}$-message m to a member y are:

1. x sends m to its assigned controller. The controller evaluates the ruling of the law $\mathcal{L}$ for the event send(x,m,y) and it carries out this ruling. If part of the ruling is to forward the message m to y, the controller sends m to the controller assigned to y.

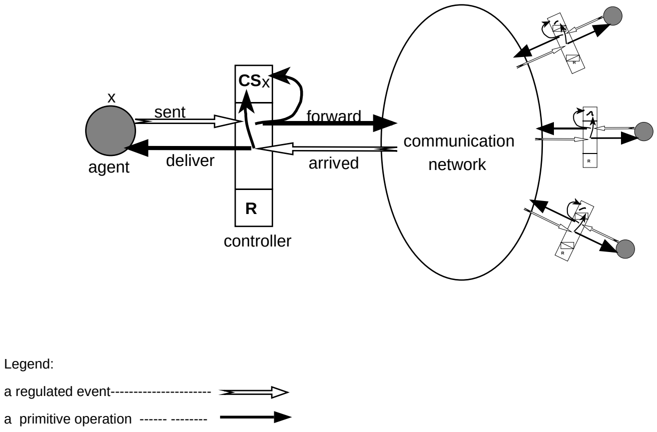
x
agent
sent
deliver
CS_X
R
controller
forward
arrived
communication network
Legend:
a regulated event---------------------
a primitive operation ------ --------

Figure 1: Enforcement of the Law

2. when m arrives to y's controller it generates an `arrived(x,m,y)` event. The ruling for this event is computed and carried out. The message m is delivered to y if so required by the ruling.

The essential aspect of this architecture is that *all controllers have identical copies of the law.* It is in this sense that the law is said to be global to the group.

The correctness of the proposed mechanism is based on the following assumptions: (1) messages are securely transmitted over the network, and (2) $\mathcal{P}$-messages are sent and received only via correctly implemented controllers, interpreting law $\mathcal{L}$. To ensure the first of these conditions, every agent belonging to $\mathcal{G}$, and each controller, have a pair of (RSA) keys: a public key known to a trusted authority and a secret key known only by itself. If the messages sent across the network are digitally signed, their authenticity is guaranteed as long as the private key is not disclosed.

Condition (2) above is more problematic, and can be handled at two levels of security. First, if one is willing to trust the OS-kernel of the hosts of all members of the policy-group $\mathcal{G}$ — which may be the case within the intranet of a given enterprise – then this condition can be satisfied by placing the controllers in the OS kernels. Each controller would acquire the law that needs to be interpreted from some trusted authority.

One way to handle the case of untrusted OS-kernels is to ensure the integrity of the controllers by building them into *physically secure coprocessors* [22, 23], or into *smart cards* [11]. Such a secure device consists of a CPU, non-volatile memory, encryption hardware and special sensing circuitry to detect intrusion. The sensing circuitry erases non-volatile memory before attackers can penetrate far enough to disable the sensors or read memory contents. If Moses is implemented on such physically secure hardware devices, the receiver of a $\mathcal{P}$-regulated message has a high degree of trust that this message is authentic, in the sense that the message has been sent by a genuine controller interpreting law $\mathcal{L}$ of the policy $\mathcal{P}$.

Controllers can also be trusted if they are maintained as a public utility by a large, trustworthy, financial institution—like Visa or Master Card—by an Internet provider, or by the post offices of various countries. More about such controller utility can be found in a technical report obtainable from the authors.

3.4 Example: A Capability Based Regime

In centralized systems protection has been traditionally realized by means of capabilities and access-control lists, each of these models offering its advantages. In distributed systems however, the full power of capabilities has not been realized so far. In particular, in timesharing operating systems like Hydra [6] and StarOs [10] it was possible to control dissemination of capabilities by specifying whether a given capability can be moved or delegated to others. This feature is no longer supported in capability-based distributed systems [16], because nothing prevents a user from duplicating the capabilities he holds. We do not have this problem because capabilities are kept by controllers, which are *trusted* to execute only prescribed operations. This is demonstrated by policy $\mathcal{CR}$ which imposes a capability based access control regime in which *capabilities can be moved* from one agent to another but *cannot be copied*[3].

The components of policy $\mathcal{CR}$ are as follows: The group $\mathcal{G}_{\mathcal{CR}}$ consists of the servers and all their clients. The set $\mathcal{M}_{\mathcal{CR}}$ consists of all the messages exchanged between the server and its clients; and the law $\mathcal{L}_{\mathcal{CR}}$ mirroring the rules described above, is

[3] This is only a finger exercise, meant to illustrate the mechanism; a full implementation of capabilities should consider copyable capabilities, revocation, etc.

presented in Figure 2.

The set $\mathcal{R}_{\mathcal{CR}}$ of this law consists of four rules. Each is followed with a comment (in italic), which together with the following discussion, should be understandable even for a reader not familiar with Prolog. Consider a set of clients that perform operations by sending messages of the form `execute(o,op,p)` to servers, where `op` is an operation to be executed on object `o`, and `p` are parameters for this operation. We assume that initially clients have in their control state arbitrary sets of capabilities represented by terms of the form `capability(o,ar)`, where `o` is the name of an object, `ar` is the set of access rights the agent has for `o`. What gives these terms their meaning as access privileges is Rule $\mathcal{R}1$ of our law, is that a request to `execute` an operation `op` on some object `o` is forwarded only if the sender has an attribute `capability(o,ar)` *and* if `op` belongs to `ar`. When such a message arrives at a server, it is delivered by Rule $\mathcal{R}2$. Note that the servers need not know about our access control scheme, they just respond to every request they receive.

The rest of this law defines the manner in which capabilities are to be moved from one agent to another. This is done as follows: by rule $\mathcal{R}3$, if the owner of a capability for object `o`, sends a `move(capability(o,ar))` message, his capability for `o` is removed from his control state. By rule $\mathcal{R}4$, the arrival of a `move(capability(..))` message causes the addition of the corresponding capability in the control state of the receiver.

Initially: Every client has in its control state various attributes of the form `capability(O,AR)`, where `O` is an object on which the client has the set `AR` of access rights.

$\mathcal{R}1$.

```
sent(C,execute(Op,O,P),S) :-
     capability(O,AR)@CS,
     member(Op,AR),
     do(forward).
```

A request to `execute` *an operation* `Op` *on some object* `O` *is forwarded only if the sender has a capability for* `O` *and if* `Op` *belongs to* `AR`*, the set of access rights.*

$\mathcal{R}2$.

```
arrived(C,execute(Op,O,P),S) :-
     do(deliver).
```

Arrived `execute` *messages are delivered without further ado.*

$\mathcal{R}3$.

```
sent(C,move(capability(O,AR)),D) :-
     capability(O,AR)@CS,
     do(-capability(O,AR)),
     do(forward).
```

A `move(capability(..))` *will be forwarded only if the sender has a capability for the object* `O`*. The sender's capability is removed from his control state.*

$\mathcal{R}4$.

```
arrived(C,move(capability(O,AR)),D) :-
     do(+capability(O,AR)).
```

The arrival of a `move(capability(..))` *message causes the addition of the corresponding capability in the control state of the receiver .*

Figure 2: Law $\mathcal{L}_{\mathcal{CR}}$ - Establishing a capability based regime

3.5 The Creation and the Maintenance of a Policy

Under our current implementation of the Moses toolkit, a new policy $\mathcal{P}$ is established by creating a number of controllers that interpret law $\mathcal{L}_{\mathcal{P}}$ and a server that provides persistent storage for the law $\mathcal{L}$ of this policy– including the control-states of all members of the policy-group $\mathcal{G}$. This server is called the *secretary* of $\mathcal{P}$, to be denoted by $\mathcal{S}_{\mathcal{P}}$. The following are some of the services provided by such a secretary.

For a process `x` to be able to exchange $\mathcal{P}$-messages under a policy $\mathcal{P}$, it needs to send a `connect(a)` message to $\mathcal{S}_{\mathcal{P}}$, asking to be associated with some agent `a` that is a member of the group $\mathcal{G}$ of $\mathcal{P}$. $\mathcal{S}_{\mathcal{P}}$ is likely to require authentication, which can be in form of a password, an X.509 certificate [12] or the recently developed SDSI certificate [17]. If the secretary agrees to make the connection, it would assign `x` to some controller interpreting law $\mathcal{L}_{\mathcal{P}}$, after providing this controller with the current control-state and the public key of the agent.

Once this connection is made, the interaction of `x` with the various members of policy $\mathcal{P}$ does not directly involve the secretary $\mathcal{S}_{\mathcal{P}}$. However, if some event at `x` ends up changing the control-state of the member `a` it is associated with, this change would be automatically communicated to $\mathcal{S}_{\mathcal{P}}$.

The secretary of a policy also acts as a name server for the members of its group $\mathcal{G}$, and it provides means for admitting new members into $\mathcal{G}$, and for removing existing members from it. These operations, which are subject to $\mathcal{L}$, are not discussed here in detail.

Finally, we note that a policy does not have to be supported by a single secretary. It is possible, in principle, for a policy to have several secretaries, each maintaining a subgroup of $\mathcal{G}$.

3.6 Fault Tolerance and Scalability

Regulated interaction lends itself to fault tolerant and scalable implementations, as we argue briefly below.

Fault Tolerance: Since RI assumes nothing about the interacting agents, it is tolerant to all their failures, even of a Byzantine kind. But RI is sensitive to two kinds of failures: (a) the failure of the `secretary`, which may have a devastating effect on the long term existence of the policy-group, even if it has no effect on the immediate interaction between its members; and (b) the failure of a controller. Fail-stop failures of these two kinds can be handled by well known methods. Failures of the secretary can be addressed by means of the state-machine approach [19], using a toolkit such as Isis [2] for the active replication of the secretary. Failures by controllers can be analogously handled by replication of each controller. Alternatively, given a reliable secretary, it may be sufficient for the controllers to notify the secretary of all state changes.

Scalability: Since the law is enforced strictly locally, by the controller of each agent, the size of the policy-group has no effect on the interaction between its members. Therefore, RI is *naturally scalable*, particularly in the case of an *open group*. However, when a group is supported by a single secretary, as in our current implementation, then the size of the group does affect operations such as finding the name of a fellow member of a group, or reporting to the secretary a change of the CS of a given member. But this has a second order effect on the efficiency of interaction under RI.

3.7 Implementation Status

An experimental prototype of the Moses toolkit has been implemented. Our controllers are written in Java, so that Moses toolkit is portable to different platforms. Because our rules do not require the full power of Prolog language, we have built an interpreter for the needed subset of Prolog. This implementation distinguishes between two types of agents:

(i) A *bounded agents*, driven by a specific program (which is what "binds" it). This programs, which can be written in C, C++, Java, or Prolog, uses a set of preprogrammed primitives for communication with Moses' controllers.

(ii) An *unbounded agents*, which represents a human, not bound by any program. Such an agent communicates with its assigned controller via Netscape, using application specific interfaces consisting of HTML documents with embedded applets. Our choice was motivated by (1) the almost universal deployment of WWW browsers; and (2) the ease of learning to use this interface.

The implementation has been tested on UNIX platforms including SunOS and Solaris. The controllers have not yet been deployed on physically secure devices.

4 Implementation of Our Example Policies

To illustrate the expressive power of the proposed mechanism, we present here the implementation in Moses of two disparate policies mentioned previously: the Chinese Wall policy, and the sealed-bid auction policy. Recall that although these policies, and the one discussed in Section 3.4, are defined by separate laws, unrelated to each other, a given agent may be subject to several of these policies, with respect to different modes of interaction it is involved in.

It should be pointed out that our implementation of the sealed-bid policy assumes no loss of messages and no failure of controllers. On the other hand the implementation of the Chinese Wall policy is robust with respect to loss of messages and fail-stop failures of controllers.

4.1 An Implementation of the Chinese Wall Policy

This policy is established by law $\mathcal{L}_{\mathcal{CW}}$, displayed in Figure 3. We assume that the servers of the distributed financial database are trusted to respond to $\mathcal{L}_{\mathcal{CW}}$ messages of the form `request(c)`, where `c` identifies a company, by means of `respond(c,data)`, where `data` represents information about `c`. Note that under this law no explicit access control is required on the part of the server.

Under law $\mathcal{L}_{\mathcal{CW}}$ a client is authorized to access data belonging to a company `c` if either of the following conditions is satisfied:

1. the client has a clique permit for clique `q` to which `c` belongs—such a permit is represented by a term `cliquePermit(q)` in the control state of the client;

2. the client has a company permit for company `c`—represented by the term `companyPermit(c)` in the control state of the client.

If the access is granted based on a `cliquePermit`, then this permit is automatically removed to prevent the client from accessing data regarding a competitor company, and replaced with a `companyPermit` for company `c`.

This implementation deals with the exceptional situation when a first time request for a company is not honored for whatever reason. In this case, a client should be able to access information about another company belonging to the same competition clique. That is why we chose to replace a `cliquePermit` by a `companyPermit` at the time the client *receives* the information and not when he makes the *request*. In this respect, the protocol presented is tolerant to server faults of type fail/stop and to lost messages. Note also that Rule $\mathcal{R}1$ checks for the presence of a company permit or a clique permit at the time a request is sent. This check is not needed for the correctness of the protocol, it is performed to ensure that only potentially valid requests are sent to servers and thus diminish the possibility of server congestion, and thus of denial of service.

$\mathcal{I}$nitially: Every client has in its control state some attributes of the form `cliquePermit(Q)`

$\mathcal{R}1$.
```
sent(U,request(C),S) :-
     (companyPermit(C)@CS |
     (belongsTo(C,Q),cliquePermit(Q)@CS)),
     do(forward).
```

If a user `U` has a permit for company `C` or he has a permit for clique `Q` to which `C` belongs, then the request is forwarded.

$\mathcal{R}2$.
```
arrived(U,request(C),S) :-
   do(deliver),
   do(+requested(C,U)).
```

If a `request` message arrives at a server, the message is delivered. Also, a term `requested(C,U)` is added to the control state to record the fact that user `U` is requesting data about company `C`.

$\mathcal{R}3$.
```
sent(S,response(C,Data),U) :-
     requested(C,U)@CS,
     do(-requested(C,U)), do(forward).
```

In response to a `request(C)` message, a server can send a `response(C,Data)`. Note that this message may be sent only by the server to whom the user addressed the request, i.e. the server which has the term `requested(C,U)`.

$\mathcal{R}4$.
```
arrived(S,response(C,Data),U) :-
     companyPermit(C)@CS,do(deliver).
```

If user `U` receives information about a company `C`, for which he has a permit, the data is delivered.

$\mathcal{R}5$.
```
arrived(S,response(C,Data),U) :-
     belongsTo(C,Q),cliquePermit(Q)@CS,
     do(-cliquePermit(Q)),
     do(+companyPermit(C)),
     do(deliver).
```

If user `U` receives information about a company `C`, belonging to a clique `Q` he loses the permit for clique `Q` and gets a permit for the company `C`.

$\mathcal{R}6$. `belongsTo(att,communication).`

$\mathcal{R}7$. `belongsTo(ibm,communication).`

⋮

This rules state that att and ibm belongs to communication competition clique. There will be one such rule for every company whose financial information is available.

Figure 3: Law $\mathcal{L}_{\mathcal{CW}}$ for Chinese Wall Policy

4.2 An Implementation of the Sealed-Bid Auction Policy

We introduce $\mathcal{L}_{\mathcal{SB}}$, displayed in Figure 4, which implements the law of the sealed-bid auction policy $\mathcal{SB}$ introduced in Section 2.2. This law regulates two different types of messages: the messages that can be used by the agents involved in this policy to withdraw and deposit money, and the messages related to the auction per se.

The deposit and withdrawal of money. We assume that an agent called `bank` is a financial institution to which both auctioneers and bidding agents have accounts. The bank is trusted to respond only to $\mathcal{P}_{\mathcal{SB}}$ messages of the form `transaction(..)` and we assume it performs the financial operations correctly. Any agent has in its control state a term `cash(amount)` where `amount` is the sum available for bidding. Under this law, agents are allowed to transfer money from their account to their bidding fund and vice versa. An agent, wishing to withdraw sum `s` from its bank account sends to `bank` the message `transaction(type(withdrawal),sum(s))`. The `bank` responds with an `addCash(s)` message if the transaction is valid. When the agent receives such a message the amount `s` is automatically added to its cash (Rule $\mathcal{R}4$). Similarly, an agent can make a deposit in amount of `s` into his bank account, by sending the message `transaction(type(deposit),sum(s))` (Rule $\mathcal{R}1$). This message will be forwarded to the `bank`, only if the agent has enough cash, after his `cash` term is decreased accordingly (Rule $\mathcal{R}1$).

The auction process. Intuitively, a sealed-bid auction proceeds as follows. First, an auctioneer `x` can start a sale by multicasting the message `startAuction(item(i),end(et))`, where `i` is a unique description of the item to be sold, and `et` is the time when the auction ends (Rule $\mathcal{R}5$). At the same time, a term `auction` is added to the control state of the auctioneer which records the highest bid made so far (initially 0) and the name of its issuer (initially null). When such a message arrives at an agent `y`, a term `bided(x,i,et,0)` is added to `y`'s control state (Rule $\mathcal{R}6$). This term serves two purposes: (i) to enable `y` to bid for the item as many times as he wants to, but only in the allotted time; and (ii) to record the maximal bid made by `y` for item `i` (initially 0). An agent `y` makes a bid of `val` dollars for item `i` by sending a message `bid(i,val)` to the auctioneer. Such a message is forwarded to the destination only if the following conditions are met: (i) the deadline `et` has not yet passed, (ii) `val` is the highest bid `y` made so far, and (iii) `y` has enough cash (Rule $\mathcal{R}7$). Also, the cash amount `y` possesses is decreased, and the `bided` term is modified to reflect that `val` is his highest bid for `i`.

All such `bid` messages arrive at the controller of the auctioneer `x`, which maintains a term `auction` recording the largest bid so far and the name of the winner (Rule $\mathcal{R}8$). Note that the auctioneer himself never gets the bids, the winner is determined by the controller automatically, thus ensuring bidders privacy and correctness of the computation.

The auction of item `i` is finalized by a message `endAuction(i)` from the auctioneer, which he is allowed to send only after the deadline has passed ($\mathcal{R}9$). The effects of this message are described briefly below. First, the auctioneer's amount of cash is increased by the value of the highest bid, which he can later deposit in the bank, by Rule $\mathcal{R}1$. Second, the controller of the auctioneer will multicast the message `endAuction(i,w)` containing the identifier of the winner to all group members. Note that the auctioneer himself does not know the identity of the winner—only his controller has this information. When the message `endAuction(i,w)` arrives at a losing bidder, his cash amount is increased by the highest value he bided on `i`, so he does not loose any money (Rule $\mathcal{R}10$). When this message arrives at the winner `w`, then, by the same rule, the message is delivered, thus notifying `w` that he won.

Note that for the sake of simplicity, we do not address here the situation of an auctioneer denying the winner his prize. This can be prevented by having the controller of the auctioneer in question send the winner an appropriate certificate.

Initially: Every agent has in his control state an attribute `cash(Amount)` where `Amount` is the sum the agent can use for bidding (initially 0).

$\mathcal{R}1$.
```
sent(X,transaction(type(T), sum(S)),bank) :- (T = withdrawal |
          (T=deposit, cash(Amount)@CS, S<Amount, do(dcr(cash(Amount),S)))), do(forward).
```

If the transaction is a `deposit`*, the message is forwarded only if the client has enough cash. If this is the case, the amount of cash is decreased by* `S`.

$\mathcal{R}2$.
```
arrived(_,transaction(type(T), sum(S)),bank) :- do(deliver).
```

$\mathcal{R}3$.
```
sent(bank, addCash(S), _) :- do(forward).
```

Messages sent to the bank are delivered, and messages sent by the bank are forwarded without further ado.

$\mathcal{R}4$.
```
arrived(bank, addCash(S), X) :- cash(Amount)@CS, do(incr(cash(Amount),S)).
```

The cash amount is increased by `S`*, when a successful bank transfer is performed.*

$\mathcal{R}5$.
```
sent(X,startAuction(item(I),end(ET)),all) :- auctioneer@CS,
          not (auction(I,_,_,_)@CS), do(+auction(I,ET,null,0)),
          do(forward).
```

An auctioneer can start an electronic auction by sending a `startAuction` *message to* `all` *members. The message contains an identifier* `I` *of the item to be auctioned and the time* `ET` *when the auction ends.*

$\mathcal{R}6$.
```
arrived(X,startAuction(item(I),end(ET)),Y) :-
          do(+bided(X, I, ET, 0)),do(deliver).
```

A `startAuction` *message is delivered to the destination. A term* `bided(X, I, ET, 0)` *is added to the control state of the receiver.*

$\mathcal{R}7$.
```
sent(Y,bid(I, Val),X) :- not(auctioneer@CS),clock(T), T < ET,
          bided(X, I, ET, V)@CS, V<= Val, cash(Amount)@CS,
          V + Amount >= Val, do(bided(X, I, ET, V)←bided(X, I, ET, Val)),
          do(dcr(cash(Amount)), Val-V), do(forward).
```

If the time to bid has not expired and the bidder has enough cash, then a `bid` *message containing* `Val` *the value of the bid, is forwarded to the initiator.*

$\mathcal{R}8$.
```
arrived(Y,bid( I, Val),X) :- auction(I, ET, W, Max)@CS, Val > Max,
          do(auction(I,ET,W,Max)←auction(I,ET,Y,Val)).
```

If `Val` *is the biggest amount bided so far for* `I`*, the sender's identifier is recorded in the* `auction` *term along with* `Val`.

$\mathcal{R}9$.
```
sent(X,endAuction(I),all) :- clock(T),T > ET +100,
          cash(Amount)@CS,auction(I,ET,W,Max)@CS,
          do(incr(cash(Amount),Max)), do(forward(X,endAuction(I, W),all)).
```

Only the auctioneer `X` *who organized the sale for item* `I` *can send a* `endAuction` *message. The identifier of the winner* `W` *are sent to all group members. Also, the money collected from the winner are added to* `X`*'s cash.*

$\mathcal{R}10$.
```
arrived(X,endAuction(I,W),Y) :- cash(Amount)@CS,
          bided(X, I, ET, Val)@CS, do(-bided(X,I, ET, Val)),
          W=Y→do(deliver) | do(incr(cash(Amount),Val)).
```

When a message `endAuction` *arrives at a bidder* `Y`*, if he is a loser he gets his money back.*

Figure 4: Law $\mathcal{L}_{\mathcal{SB}}$ for Sealed-Bid Auction Policy

5 Related Work

The need for a mechanism for specifying security policies as an alternative to hard coding them into an application occurred to several researchers. Theimer, Nichols and Terry [21] introduced a concept of *generalized capabilities*. Such capabilities contain access control programs (ACP) encoding the security policy to be enforced with respect to this capability. When a server receives a request accompanied by such a generalized capability, it executes the ACP to determine whether the request is valid or not.

Finally, Blaze, Feigenbaum and Lacy [3] built a toolkit called PolicyMaker which can interpret security policies. An agent receiving a request gives it for evaluation to PolicyMaker together with its specific policy, and the requester's credentials. On this basis the request can be found to be valid, invalid or trust can be deferred to third parties. One of the main differences between this work and ours is that PolicyMaker provides no enforcement. In particular, after asking PolicyMaker for its ruling one can proceed by ignoring it.

Also, in both these approaches the rights a user has are *static*: they cannot be modified in accordance with its actions. Thus, a large range of security policies, like separation of duties [5], Chinese Wall, and the movable but uncopyable capabilities, where the *state* of a user determines his rights, cannot be enforced.

6 Conclusion

The essence of the security mechanism proposed here is the existence of a law that is guaranteed to be observed by *all* members of a given policy-group. It is this uniform[4] law that allows the members of the group governed by it to trust each other. This *distributed trust* has several beneficial consequences: (a) it simplifies the formulation of a wide range of policies, some of which cannot be supported by traditional means; (b) it allows a single agent to operate under several distinct policies; (c) it makes the enforcement of policies more efficient; and (d) it makes the mechanism itself scalable. This trust relies on the integrity of the controllers, and on the ability to correctly identify $\mathcal{P}$-messages. These conditions can be met, with a very high level of confidence, by implementing controllers on physically secure devices, and by appropriate authentication protocols. In some cases, however, it should be sufficient to build the controllers into the kernel of the operating systems involved.

[4] The law is uniform with respect to the members of each group.

References

[1] J. R. Anderson. A security policy model for clinical information systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[2] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust managemnt. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[4] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium in Security and Privacy*. IEEE Computer Society, 1989.

[5] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium in Security and Privacy*, pages 184–194. IEEE Computer Society, 1987.

[6] E. Cohen and D. Jefferson. Protection in the HYDRA operating system. In *Operating Systems Principles*, pages 141–160. ACM, Nov. 1975.

[7] S. Foley. The specification and implementation of 'commercial' security requirements including dynamic segregation of duties. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, April 1997. (to appear).

[8] S Foley, L. Gong, and X. Qian. A security model of dynamic labelling providing a tiered approach to verification. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[9] M. Franklin and M. Reiter. The design and implementation of a secure auction service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–14, May 1995.

[10] A. Jones, R. Chansler Jr., I. Durham, K. Schwans, and S. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 117–127, Dec 1979.

[11] M. Jones and B. Schneier. Securing the World Wide Web: Smart Tokens and their implementation. In *Proceedings of the Fourth International World Wide Web Conference*, pages 397–409, December 1995.

[12] S. Kent. Internet privacy enhanced mail. *Communications of the ACM*, August 1993.

[13] N.H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, February 1991.

[14] N.H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 81–98, September 1997.

[15] N.H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building reconfiguration primitives into the law of a system. In *Proc. of the Third International Conference on Configurable Distributed Systems (ICCDS'96)*, March 1996. (available through `http://www.cs.rutgers.edu/~minsky/`).

[16] S. Mullender, G. Rossum, A. Tanembaum, R. Van Renesse, and H. Staveren. Amoeba:a distributed operating system for the 1990s. *IEEE Computer*, May 1990.

[17] R. Rivest and B. Lampson. SDSI-a simple distributed security infrastructure. Technical report, 1996. http://theory.lcs.mitedu/~rivest/sdsi.ps.

[18] Ravi Sandhu. Lattice-based access control models. *IEEE Computer*, November 1993.

[19] F.B. Schneider. Implementing fault tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):300–319, 1990.

[20] A. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[21] M. Theimer, D. Nichols, and Douglas Terry. Delegation through access control programs. In *Proceedings of Distributed Computing System*, pages 529–536, 1992.

[22] J.D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU-CS-91-140R, CMU, 1991.

[23] B. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, May 1994.