

# Using Certified Policies to Regulate E-Commerce Transactions\*

Victoria Ungureanu<sup>†</sup>  
Department of MSIS  
Rutgers University  
ungurean@rbs.rutgers.edu

## Abstract

E-commerce regulations are usually embedded in mutually agreed upon contracts. Generally, these contracts enumerate agents authorized to participate in transactions, and spell out such things like rights and obligations of each partner, and terms and conditions of the trade. An enterprise may be concurrently bound by a set of different contracts that regulate the trading relations with its various clients and suppliers. This set is dynamic because new contracts are constantly being established, and previously established contracts end, are annulled or revised.

We argue that existent access control mechanisms cannot adequately support the large number of regulations embedded in disparate, evolving contracts. To deal with this problem we propose to use certified policies. A certified policy (CP) is obtained by expressing contract terms regarding access and control regulations in a formal, interpretable language, and by having them digitally signed by a proper authority. In this framework, an agent making a request to a server presents to the server such a CP together with other relevant credentials. A valid certified policy can then be used as the authorization policy for the request in question.

It is the thesis of this paper that this approach would make several aspects of contract enforcement more manageable and more efficient: (a) deployment: certified policies may be stored on certificate repositories, from where they can be retrieved as needed, (b) annulment: if a contract is annulled, the corresponding CP should be invalidated; the latter can be conveniently supported by certificate revocation, and (c) revision: revision of contract terms can be done by publishing a new certified policy, and by revoking the old one. The proposed approach is practical in that it does not require any modification of the current certificate infrastructure, and only minor modifications to servers.

In this paper we propose a language for stating contract terms, and present several formal examples of certified policies. We describe the implementation of the enforcement mechanism and present experimental performance results.

---

\*ACM Transactions on Internet Technologies, February 2005

<sup>†</sup>Work supported in part by DIMACS under contract STC-91-19999 ITECC, and Information Technology and Electronic Commerce Clinic, Rutgers University

# 1 Introduction

In order to save money, be competitive and efficient, more and more enterprises are taking steps towards conducting transactions with trading partners on-line [15]. Trading relations are based on mutually agreed upon contracts. Generally, these contracts enumerate agents authorized to participate in transactions, and spell out such things like rights and obligations of each partner, and terms and conditions of the trade. An enterprise may be concurrently bound by a set of different contracts that regulate the trading relations with its various clients and suppliers. For example, Ford has approximately thirty thousand suppliers, each operating under a different contract, and General Motors has about forty thousand [16]. *Control occurs then as an ancillary to such commercial agreements.*

The problem addressed in this paper is how to formulate and enforce the control regulations embedded in contracts. What makes this problem considerably more complex, and more challenging than traditional access control, is the large set of autonomous regulations that need to be carried out. There are currently two main methods to establish a set of policies: (1) to establish a dedicated server for each policy (for e.g. [13, 35]), and (2) to *combine* them into a single super-policy (e.g. [4, 5, 8, 20, 25, 27, 43]). We will argue that both approaches are problematic in e-commerce context.

Having a dedicated server for each contract is an expensive proposition if the number contracts an enterprise is bound by is very large. The large number of contracts also makes combination of policies difficult to perform. Moreover, even if it would be possible to create such a composition, it would still be very problematic to maintain it. New contracts are constantly being established, and previously established contracts end, or are being revised. Each such modification triggers, in turn, the modification of the composition-policy, leading to a maintenance nightmare.

Thus, in e-commerce context, a huge, ever-changing policy, subsuming all contract regulations bounding an enterprise, becomes prohibitively hard and error-prone to maintain. And establishing a dedicated server for each contract is simply unpractical. To deal with this problem, we propose in this paper to use the certificate framework for supporting contract regulations. Certificates, by which we mean digitally signed statements of some sort, are commonly used to establish relations between parties who are physically distant or do not know each other. Generally, a certificate conveys information regarding a subject—a software agent or a human user. Certificate based authorization is carried out as follows: An agent submits a request, together with a certificate (or a list of certificates) to a server. The server verifies the credentials and grants or denies the request, according to its internal, predefined access control policy.

In e-commerce context, the policy that a server has to enforce is denoted by the regulations agreed upon in the contract. We are proposing here that digital signatures should be used not only to certify the credentials a user presents, but also to authenticate contract regulations. Namely, this approach assumes that: (a) the terms of a contract pertaining to access and control regulations are expressed in a formal, interpretable language, and are digitally signed by a proper authority, and (b) an agent making a request presents to a

server such a certified policy (abbreviated here as CP), together with other relevant credentials. A valid certified policy can then be used as the authorization policy for the request in question.

It is the thesis of this paper that this approach would make several aspects of contract enforcement more manageable and more efficient:

- **deployment**: certified policies sanctioned by an enterprise may be stored on repositories, from where they can be retrieved as needed.
- **annulment**: if a contract is annulled, the corresponding CP should be invalidated; nullification of CPs can be conveniently supported by certificate revocation.
- **revision**: if the terms of a contract need to be revised, this can be done simply by publishing a new certified policy, and by revoking the old one.

In this paper, we describe a formalization for certified policies and a framework for their support. The CP formalization supports both stateless and stateful contract regulations. In the proposed framework, certified policies are enforced by generic policy engines, called observers, that can interpret and carry out provisions embedded in CPs. The key difference between an observer and other generic policy enforcers proposed recently is that the latter type generally enforces a monolithic security policy, which is known a-priori (for e.g. [24, 30, 17, 25]). In our approach, there is no longer a monolithic policy; instead, there are several, autonomous control policies, expressed as CPs.

To gauge the efficacy of the framework we run a simulation study driven by empirical data. The study compares the performance of the CP framework with the performance of a conventional enforcing mechanism, employing monolithic policies, obtained by *composing* contract regulations. The study shows that CPs are generally more efficient than conventional composition policies — dramatically more efficient, if contract regulations are stateful. To be more specific, when a quarter of requests are governed by stateful contracts, a CP-based mechanism outperforms a conventional mechanism by a factor of ten.

The rest of the paper is organized as follows: We start, in Section 2, by describing how contract regulations can be expressed as CPs, and by illustrating this concept with an example. We follow, in Sections 3 and 4 by discussing how certified policies can be revoked and updated. Section 5 presents an extension to the concept of certified policy, which allows expressing contract terms as function of dynamically modifiable state. Section 6 shows how internal regulations of an enterprise can be also formalized as CPs and enforced by observers. The system implementation is introduced in Section 7. The efficiency of the mechanism is discussed in Section 8, where we show that a mechanism employing CPs is more efficient and more scalable, than a mechanism employing conventional composition policies. Section 9 discusses related work, and we conclude in Section 10.

## 2 Certified Policies

To illustrate the nature of contract provisions we describe now, in an informal manner, an example of a contract (we will show in Section 2.3 how its terms can be formalized by a certified policy). Consider that a travel agency, called TravelRus, may purchase discounted tickets from an airline company, called FlyAway, under the contract defined below:

*The price of each ticket bought between January 1 and May 1, 2004 is discounted by 10%. The client-enterprise, TravelRus, specifies a bank account from which payments should be made, and the vendor enterprise, FlyAway, specifies a bank account into which payments should be deposited. A business transaction between the two enterprises is initiated by a purchase order (PO) sent by some agent of TravelRus, which specifies traveler information, departure and return dates, flight numbers, and class. The exchange of merchandise and payment between the two parties is subject to the following provisions:*

- *Only agents certified as travel agents by `travelRus_CA`<sup>1</sup>, a designated certification authority of the client enterprise, may issue purchase orders.*
- *If there are seats available for the flights requested, an e-ticket is issued and 90% of the nominal price of the ticket is transferred from the account of TravelRus into the account of FlyAway.*

A certified policy defines in a formal language contract terms that can be verified automatically. We emphasize that contracts often contain provisions that do not fall into the above category. For example, when the purchased merchandise is not digital, terms regarding its quality, freshness, and delivery cannot be checked automatically. Such terms cannot be formalized by CPs and need to be verified using other methods.

When designing CPs, our goals are to mirror as closely as possible the social notion of contract. At the societal level, a contract embodies an agreement between two or more parties involved in a certain (economic) activity, and refer to such things as: the **time frame** in which the activity in question is to be completed, the **agents** authorized to participate, the expected **rules** of conduct of participating agents, and the **penalties** incurred for not complying with the rules.

In practice, contracts may be nullified before the end of their validity period due to changes of legislation, bankruptcy, etc. When a contract is annulled, the corresponding CP should be also invalidated. This can be done by revoking the CP provided that there are trusted **revocation servers**, which maintain and propagate information regarding CP status.

Contract annulment may occur when there is a radical and sudden change in the legal context, or in the business conditions of either party. Often though, such changes are

---

<sup>1</sup>In this example, for the sake of convenience, `travelRus_CA` stands for the public key of the certifying authority trusted to sign travel agent certificates on behalf of the client enterprise, TravelRus.

moderate, and can be handled by contract revisions, without resorting to contract nullification. Examples of situations that call for contract update include correcting omissions and supporting unplanned circumstances (several specific scenarios which call for revisions are described in Section 4). In order to support contract updates, we assume that each CP is identified by a **name** and a **version number**; and that the latest version of a CP is maintained by a **repository**.

Traditional certificates are presented by the bearer to prove identity or group membership; similarly, certified policies are presented to show compliance with an a-priori set-up contract. The underlying assumption, in both cases, is that there is a server which is trusted to verify credentials and to grant access accordingly. In order to ease processing, certified policies contain a special attribute, `type(policy)`, which enables servers to distinguish between CPs and traditional, subject certificates. And to ease the server-location process, a certified policy may have an attribute `servedBy`, whose value denotes the address of a server, trusted by the participating parties with enforcing contract terms.

To summarize, we propose that a certified policy should contain the following mandatory components:

- **type** — denotes the type of a certified statement. It is the presence of the term `type(policy)` which enables a server to distinguish between subject certificates and CPs;
- **name** — denotes the id of the CP;
- **version** — denotes the version of the CP. The version number, as we shall see, is used to support contract revisions;
- **validity period** — denotes the validity period of the CP (coincides with the validity period of the corresponding contract);
- **revocationServer** — specifies the address of the server maintaining/disseminating information regarding CPs status;
- **repository** — specifies the address of the server that maintains the latest version of the CP in question.
- **provisions** — specify, in a formal language, the terms of the contract that can be verified automatically.

Finally, a certified policy is obtained by signing a statement comprised of these, and possibly other attributes, by a trusted principal.

## 2.1 Expressing Contract Terms

Contract terms can be quite naturally expressed by means of any formal language supporting *event-condition-action* (ECA) kind of rules. We are using here an extension of a language devised for support of control policies [29, 30] built on top of Prolog [10]. In this language contract terms are embedded in rules of the form:

```

eval(R,Cert):-
    condition-1,...,condition-k,
    provision-1,...,provision-n,
    accept.

```

This rule states that, if *condition-1* through *condition-k* are satisfied, request *R* is valid. These conditions may refer to request *R* and to certificate(s) *Cert* accompanying *R*. Moreover, this rule calls for the execution of *provision-1* through *provision-k*, which denote additional actions required by the contract.

In order to support contract provisions, the language proposed here defines a set of additional predicates, called primitive operations. Some of the primitive operations currently supported are displayed in Figure 1, and are briefly described below:

- **Operations on certificates:** these operations probe the content of a certificate and determine the issuer, or the values of the attributes attested by it. The operations on certificates include: `issuer(PK,Cert)` which binds variable `PK` to the public key of the issuer of certificate `Cert`, and `bind(Attr,Val,Cert)`, which binds `Val` to the value of attribute `Attr`, in certificate `Cert`. Since some attributes are used frequently, the language provides the following specializations of `bind`: `name(N,Cert)`, which binds variable `N` to the value of attribute `name` in certificate `Cert`, and `role(R,Cert)`, which binds `R` to the value of attribute `role` in certificate `Cert`. (We mention that certificates are translated into an internal format, essentially a Prolog list, before being passed to the Prolog interpreter. The predicates defined above are implemented as operations on lists.)

For example, assume that variable `Cert` is bound to a certificate signed with public key `rsa:3:ab1cd2ef`<sup>2</sup>. Suppose further that this certificate contains the following statement: `name(johnDoe), role(travelAgent), employer(travelRus)`. Then, after carrying out operation `issuer(PK,Cert)`, the value of `PK` is `rsa:3:ab1cd2ef`. Similarly, the effect of operations `name(N,Cert)`, `role(R,Cert)` and `attr(employer,Val,Cert)` is to bind variable `N` to `johnDoe`, variable `R` to `travelAgent`, and variable `V` to `travelRus`.

- **Communication primitives:** provide means for notifying servers whether requests are authorized or not, and for communication with various agents. Operation `accept` denotes that the evaluated request is sanctioned by the `CP` in question. Operation `reject` denotes that the evaluated request does not conform with the `CP`. Finally, operation `deliver(M,D)` sends message `M` to destination `D`.

For brevity, we do not discuss here several important aspects of the language, including *obligations*, and the treatment of *exceptions*. For these issues, and for other applications of the language, the reader is referred to [30].

---

<sup>2</sup>Real keys are, of course, longer.

---



---

Operations on certificates	
<code>issuer(PK,Cert)</code>	binds <code>PK</code> to the public key of the issuer of certificate <code>Cert</code> ;
<code>bind(Attr,Val,Cert)</code>	binds <code>Val</code> to the value of attribute <code>Attr</code> in <code>Cert</code> , if it exists; fails otherwise;
<code>name(N,Cert)</code>	binds <code>N</code> to the value of attribute <code>name</code> in <code>Cert</code> , if it exists; fails otherwise;
<code>role(R,Cert)</code>	binds <code>R</code> to the value of attribute <code>role</code> in <code>Cert</code> , if it exists; fails otherwise;

---

Communication primitives	
<code>accept</code>	denotes that the request is valid;
<code>reject</code>	denotes that the request is invalid;
<code>deliver(M,D)</code>	delivers message <code>M</code> to destination <code>D</code> .

---



---

Figure 1: Some primitive operations.

## 2.2 Deployment of Certified Policies

Several components are necessary for establishing certified policies: repositories—which maintain and disseminate certified policies; revocation servers—which record and distribute CPs status; and servers—which interpret contract terms and bring them to bear. We start by describing their functionality, and follow with a brief discussion regarding how they may be established.

Contracts can be established between two enterprises (business-to-business, or B2B, commerce) or between an enterprise and an individual client (business-to-consumer, or B2C, commerce). The proposed framework assumes that, in either case, enterprises code the control terms from the contracts they are bound by as CPs, and publish them on repositories from where they can be fetched as needed.

Repositories and revocation servers are an integral part of public key infrastructure (PKI), and can be used without any modification. However, servers need to be modified in order to deal with certified policies. The proposed approach assumes that an application server (e.g. Web server, database, or e-mail server) has an associated *observer*, to which it passes received requests for evaluation. Observers are generic policy-engines that can verify certificates, maintain relevant information regarding CPs they serve, and interpret and carry out contract terms. Servers are trusted to service only requests sanctioned by observers. (Details regarding CP enforcement and server implementation are presented in Section 7.)

Repositories, revocation servers, application servers and observers need to be maintained by an organization trusted by all the parties involved in the contract. In the case of B2B commerce, the two enterprises usually trust each other. In this case, the organization

establishing PKI and servers/observes can be one (or both) of the enterprises bound by the contract. We point out that even though the enterprises trust each other, they are still interested in ensuring compliance with contract terms. Short of that, agents (insiders or outsiders) may abuse the terms of the contract, either maliciously or mistakenly. For example, FlyAway may insist on enforcement because it ensures that discounted tickets are sold only to travel agents from TravelRus, and not to arbitrary buyers. In this particular scenario, the terms of the contract between TravelRus and FlyAway should be enforced by servers established by FlyAway.

In the case of B2C e-commerce, participants do not trust each other, especially when the enterprise is small, or not well-known to customers. We assume that in this case, the organization establishing the framework components can be a service provider, like Ebay and AOL, or software provider, such as Microsoft, or Oracle, say. (We mention the Microsoft already provides a service, called bCentral<sup>3</sup>, that allows small and medium businesses to maintain their e-commerce sites on Microsoft servers.)

### 2.3 A Certified Policy Example

We demonstrate here how contract terms can be expressed as certified policies by showing how the contract between TravelRus and FlyAway presented informally above, can be represented as a CP. In this particular example, all contract terms can be formalized by the correspondent CP. We emphasize again that this is usually not the case; contracts often have provisions that cannot be automatically verified, especially if the merchandise bought is not in digital form.

The implementation assumes that requests issued under this contract are processed by a Web server established by FlyAway. Specifically, it is assumed that the processing is done as follows: (1) a PO is issued by completing a purchase form maintained by the server, (2) if the PO is valid, the form information is processed by a CGI-program, which as a result emits an e-ticket, and (3) the e-ticket is embedded into an HTML page, which is sent to the agent who submitted the form. Furthermore, the implementation assumes the existence of a database (which maintains the seats available for FlyAway flights and their price) and of a payment mechanism (which handles payment between various clients and FlyAway). The interface to these tools is provided by two predicates called, respectively, **check** and **transfer**. These predicates are outside the scope of the enforcement mechanism and their implementation is not presented.

The contract is formalized by the certified policy displayed in Figure 2. The CP consists of two parts: preamble and rules. The preamble section specifies the following attributes. First, the name of the certified policy, the version number and the validity period. In particular, this CP is named `travelRus-flyAway`, its version number is one, and it is valid between 1/1/2004 and 5/1/2004. Second, the preamble defines the addresses of the revocation servers and of the repositories, which are trusted to maintain information regarding this CP. In this example, the functionality is provided by a single server, whose address is

---

<sup>3</sup><http://www.bcentral.com>

trust.flyAway.com.

The CP rules formalize contract provisions; to make rules easier to understand, each is followed by an explanatory comment, in italics. In this example, there are two rules that mandate in what conditions a purchase order is valid. Specifically, by Rule  $\mathcal{R}1$ , a purchase order is accepted (and a ticket is emitted), if the following conditions are satisfied: (1) the requester presents a certificate, `ClientCert`, issued by `travelRus_CA`, which certifies the bearer to be a `travelAgent`, (2) the amount the agent is willing to pay equals the discounted price of the ticket, and (3) there are seats available for the flight. Additionally, if all conditions are met, an amount that equals 90% of the nominal price of the ticket is transferred from `acc1`, the account of TravelRus, into `acc2`, the account of FlyAway. By Rule  $\mathcal{R}2$ , if any of the conditions mentioned above does not hold, then the request is rejected as being invalid under the contract.

*Preamble:*

```
type(policy).
name(travelRus-flyAway).
version(1).
validity([1,1,2004],[5,1,2004]).
repository("trust.flyAway.com").
revocationServer("trust.flyAway.com").
```

```
 $\mathcal{R}1$ . eval(request(Data,cgiProgram,method(post)),ClientCert) :-
    issuer(travelRus_CA,ClientCert),
    role(travelAgent,ClientCert),
    Data=[Amount,TravelerInfo,Class,FlightNo1,Date1,FlightNo2,Date2],
    check(SeatsAvailable,Price,Class,FlightNo1,Date1,FlightNo2,Date2),
    if (Amount= Price*0.90 and SeatsAvailable > 0) then
        (transfer(Price*0.90,from(acc1),to(acc2)),
         accept).
```

*A purchase order is accepted if the following conditions are met: (1) the requester presents a certificate, `ClientCert`, issued by `travelRus_CA`, which certifies the bearer to be a `travelAgent`, (2) the amount the agent is willing to pay equals the discounted price of the ticket, and (3) there are seats available for the flight. Additionally, an amount that equals 90% of the nominal price of the ticket is transferred from `acc1`, the account of TravelRus, into `acc2`, the account of FlyAway.*

```
 $\mathcal{R}2$ . eval(M,Cert) :- reject.
```

*All other requests are rejected.*

Figure 2: A CP example.

### 3 Contract Annulment

Contracts may be annulled when, for example, one of the partners is bankrupt, or has merged with another firm, or has been split into several firms. Usually, when a contract is annulled, the rights established by the corresponding CP need to be nullified as well. For example, if TravelRus goes out of business, travel agents formerly employed by it should not be able to buy discounted tickets from FlyAway (except if special provisions are made). Nullifying contract regulations is difficult to carry out if these regulations are expressed as components of a monolithic access control policy. This is because it requires changing the composition-policy—a laborious and error-prone process. Moreover, if the composition policy is enforced by disparate servers then the update has to take effect simultaneously on all servers. This in turn, requires means for synchronization, which are generally not supported. However, if contract regulations are expressed as CPs then invalidation of CPs can be conveniently done by certificate revocation.

Certificates might become invalid for various reasons and should be revoked. (For example, the secret key authenticated by a certificate might be lost or compromised, or the owner information, like role or address, might change.) Most revocation mechanisms rely on the existence of *certificate revocation lists* (CRLs) maintained by trusted revocation servers ([3]). To revoke a certificate, the owner of the certificate, or another responsible authority, sends the revocation server a signed message identifying the certificate to be revoked. Upon receipt of the message, the revocation server updates its CRL and disseminate the information.

Depending on the method used for disseminating revocation information one distinguishes between pull and push-based systems. In pull based systems, applications, which need to check the validity of a certificate, query the revocation server and, in response, receive the latest, signed CRL. In push based systems, interested parties subscribe to a CRL service. Afterwards, revocation servers generate and deliver CRLs in accordance with the rate stated in the subscription request [28, 44].

Similarly, certified policies may be revoked before their expiration date, by sending appropriate notices to revocation servers. However, unlike traditional certificates, CPs do not have an owner, and consequently revocation notices have to be issued by some designated authority. For now, we assume that the issuer of a CP is allowed to send revocation notices regarding the CPs signed by him/her. Given that the issuer is a trusted principal, it can be assumed that a CP is not annulled arbitrarily, without the knowledge or consent of all parties.

Like for traditional certificates, dissemination of revocation notices regarding CPs may be done using either push or pull based mechanisms. However, push-ing is far more attractive than pull-ing because, in the former case, request authorization incurs no additional overhead due to contract validation. This is so, because a server always has a suitably recent CRL that can be used to verify the validity of the CP governing a request. In contrast, under pull-based mechanisms, a server needs to contact the appropriate revocation server every time a request is received.

Furthermore, push-ing CRLs does not pose a substantial burden on revocation servers. To see why, recall that both revocation servers and their subscribers, which in this case are the application servers enforcing CPs — belong to the *same administrative domain*. It follows, that the number of subscribers is a-priori known, and thus it is possible to plan ahead and to choose the number of revocation servers to be commensurate with the number of subscribers.

**Discussion** We end this presentation by noting that certificate revocation lists have long been a point of contention in the PKI community. Opponents of this technology attribute to CRLs a number of semantical and technical limitations [18, 34, 45]. Chief among them is the high cost incurred by CRL management and distribution. However, CRL advocates have argued that there are environments where this cost is not prohibitively expensive, and where CRLs are the most efficient means for distributing revocation information. Among them, we mention business-to-consumer e-commerce [28], and single administrative domains [28, 31].

## 4 Contract Update

Generally contracts are not immutable, and in practice, revisions of a contract may be called for various reasons, like, for example, to correct omissions, to better accommodate the needs of either party, or to reflect unplanned circumstances. To be more specific, consider again our contract-example. Here are some possible scenarios that would require its modification:

1. *FlyAway also accepts payment with credit card. When payment is made with credit card, the discount is only 7%.*
2. *Certificates for TravelRus employees are now issued by another certifying authority, whose public key is denoted by the short-hand notation `travelRus_CA'`.*

In traditional control mechanisms, such changes take effect by implementing a revised access control policy into *each* server that uses the policy in question—a difficult and time-consuming undertaking. If, however, contracts are implemented as CPs, contract revision can be handled simply by: (a) issuing a new version of the CP, whose rule reflect the updates deemed necessary, and (b) revoking the previous, obsolete version. Moreover, contract update can be supported in an *efficient manner* provided that revocation information is disseminated using push based mechanisms. This is because a server can retrieve the latest version of a certified policy as soon as it was informed about the revocation of the previous version. It follows that a server does not need to check if a certified policy has been updated at the time a request is evaluated, and as such no additional overhead is incurred.

To show how contract-update can be supported, we present in Figure 3 a new version of our CP example that incorporates the last revision mentioned above. The preamble of this CP specifies that the second version of `travelRus-flyAway` CP is valid from 03/15/2004 until 05/01/2004. The revision is materialized into Rule  $\mathcal{R}1$  which states that an agent certified as a `travelAgent` by `travelRus_CA'` may issue purchase orders for tickets at discounted prices.



Figure 3: Revised version of the CP-example.

## 5 Stateful Contract Regulations

So far, contract terms have been formulated by taking into account the request content and any credentials the requester might present. While this is sufficient for many types of commercial agreements, there are cases when the terms of a contract take also into account additional information—referred here as control state. The control state, which is likely to *change dynamically* during the lifetime of a contract, might consist, for example, of: the different phases of a contract, the state of transactions regulated by the contract, or the past actions of various agents.

To illustrate this type of terms, we introduce the following amendments to our contract-example:

- *Only a limited number of tickets, say 100, may be purchased at the discounted price.*
- *FlyAway accepts reservations for tickets. A purchase order for a reserved ticket is honored only if made within 24 hours from the reservation—after this time limit, the reservation is considered void.*
- *The travel agency may cancel a ticket up to ten days after the purchase. In this case, the amount paid for the ticket minus a penalty fee of 15% of the*

---



---

<code>T@CS</code>	returns true if term <code>T</code> is present in the control state, and fails otherwise
<code>addCS(T)</code>	adds term <code>T</code> to the control state;
<code>delCS(T)</code>	removes term <code>T</code> from the control state;
<code>incr(T(V),D)</code>	increases the value of parameter <code>V</code> of term <code>T</code> with quantity <code>D</code> ;
<code>dcr(T(V),D)</code>	decreases the value of parameter <code>V</code> of term <code>T</code> by quantity <code>D</code> .

---



---

Figure 4: Operations on control state.

*amount paid is transferred from the account of FlyAway into the account of TravelRus.*

Thus amended the contract is stateful since it is sensitive to the number of discounted tickets, and to the times reservations, purchase orders and cancellations are made.

In order to support this type of contracts, certified policies have an additional component, called control state. Structurally, the control state of a CP is a list of Prolog terms, whose meaning is defined by the rules of the CP. For example, the control state of a CP implementing the first amendment presented above, will contain a term `discountedAvailable(D)`, where `D` would represent the number of discounted tickets that TravelRus can buy. The value of the control state can be tested and modified using the primitive operations presented in Figure 4. The use of these primitives will be illustrated in Section 5.1, which presents the implementation of a stateful CP.

The preamble of a stateful CP contains two additional clauses. First, an `initialCS` clause which defines the initial value of the control state. For example, the formalization of the amended contract above, contains the clause `initialCS([discountedAvailable(100)])`. Second, a stateful CP contains a `stateServer` clause that contains the address of a server, which is trusted to maintain correctly the control state during the lifetime of the CP. We mention that such servers are implemented as specialized observers.

When an observer validates a request governed by a stateful contract, it needs to retrieve the control state from the server mentioned by the `stateServer` clause. After the observer verifies whether the request complies with CP-rules, it returns the updated state to the `stateServer`. To maintain state consistency, the `stateServer` serves a request for the state of a CP, only if the state is not being used by another observer. And to ensure that the state is not modified by arbitrary agents, a `stateServer` communicates only with trusted observers (i.e. established by the same organization.)



Figure 5: An example of a stateful CP.

## 5.1 The Formalization of a Stateful Contract

Figure 5 present a fragment of a CP implementing the first amendment presented above (for brevity, we do not implement the last two amendments). The preamble of this CP prescribes that the `stateServer` of this contract is `state.flyAway.com`, and that the initial value of the control state consists of a term `discountedAvailable(100)`.

Rule  $\mathcal{R}1$  deals with purchase orders for discounted tickets. Like in the previous versions of the contract, this rule checks the certificate presented by the requester and verifies that the amount offered equals the discounted price. Additionally, in this case, the rule checks if the travel agency did not used its quota of discounted tickets. If all conditions are satisfied, the PO is accepted, and the value held in the term `discountedAvailable` is decreased by one.

## 5.2 Control State Update

Like any other contract term, the state may need to be revised for various reasons. For example, consider the previous contract, and suppose that FlyAway decides to increase TravelRus' quota of discounted tickets. Implementing such a revision requires updating the value of the control state.

To handle state updates effectively we introduce into the language a new type of rule, called `updateCS`, which mandates how the control state should be modified. An `updateCS` rule is triggered if the `stateServer`, which is maintaining the state of a given certified policy `C`, detects that `C` has been revised (i.e. that the version number has changed). This is done as follows. If both versions are stateful, the `stateServer` evaluates the `updateCS` rules, if any, found in the contract revision. If, however, only the new version is stateful (the previous version is stateless) it initializes the control state with the value found in the `initialCS` term.

To illustrate the use of the `updateCS` rule, we present in Figure 6 a fragment of a revision of the previous contract which calls for the increase in the number of discounted tickets by 50 tickets.

```
Preamble:
  type(policy).
  name(travelRus-flyAway).
  version(4).
  validity([4,15,2004],[5,1,2004]).
  repository("trust.flyAway.com").
  revocationServer("trust.flyAway.com").
  stateServer("state.flyAway.com").
  initialCS([discountedAvailable(50)].

R1. updateCS :-
    discountedAvailable(D)@CS,
    incr(discountedAvailable(D),50).

When a change in version is detected, the control state is updated before any request is evaluated. In this example, the value of the discountedAvailable term, established by the previous version of the contract, is increased with 50.
```

Figure 6: CP-fragment dealing with state update.

## 6 Support for Internal Regulations

A contract embeds regulations pertaining to the trade regulated by the contract at hand. Enterprises often define general internal policies that refer to *all* its e-commerce transactions. In this case, an e-commerce transaction is governed by the appropriate contract *and* the internal regulations of the enterprise(s) involved in the contract.

As an example of internal regulations consider again the TraveRus-FlyAway scenario. Suppose further, that FlyAway has the following policy regarding on-line purchase orders:

*For auditing reasons, all valid purchase orders should be sent to a designated agent.*

Such internal regulations can also be expressed as CPs and enforced by observers. For example, Figure 7 shows the implementation of the internal policy of FlyAway. Like contract terms, formalized internal regulations are signed by trusted parties and are stored by repositories. And updates of internal policies are handled similar to contract terms — namely, by issuing a new version of the CP, and by revoking the old one. However, unlike CPs denoting contract terms, which are retrieved (when needed) from repositories, internal regulations are deployed on observers *before* observers become operational.

If internal regulations are expressed as CPs, an observer validates a request in two steps. First, an observer verifies that the request complies with contract terms. Second, an observer verifies that the request complies with the internal policy of the enterprise. For example, when an agent from TravelRus sends a purchase order to a server, the observer associated with the server evaluates the PO with respect to two certified policies: `travelRus-flyAway` and `flyAway` (displayed in Figures 5 and 7).

```

Preamble:
  type(policy).
  name(flyAway).
  version(1).
  validity([1,1,2004],[1,1,2005]).
  repository("trust.flyAway.com").
  revocationServer("trust.flyAway.com").

R1. eval(request(Data,cgiProgram,method(post)),ClientCert) :-
      Data=[Amount,TravelerInfo,Class,FlightNo1,Date1,FlightNo2,Date2],
      deliver(Data,auditor),
      accept.

  A purchase order is delivered to the auditor.

```

Figure 7: The CP formalization of FlyAway internal regulations.

We point out that if an enterprise has already in place a mechanism that enforces internal regulations, then it might be preferable to continue using it. If this solution is adopted, observers should be extended to communicate with the mechanism enforcing internal regulations. (So far, this type of communication has not yet been implemented.)

## 7 System Architecture

The system architecture, illustrated in Figure 8, relies on the existence of the following trusted entities: revocation servers, repositories, application servers, observers and state servers. Under this scheme, an application server (e.g. Web server, database, or e-mail server) has an associated observer, to which it passes received requests for evaluation.

Observers are generic policy engines that verify certificates, maintain CP-information, and interpret and carry out CP-rules. The proposed enforcement mechanism assumes that the following data is available to observers at set-up time. First, the *list of trusted issuers of CPs*, called TI. For each such issuer, an observer records its public key, which is used to verify signatures. As we shall see, an observer authorizes a request only if accompanied by a CP signed by a trusted issuer. Second, the *list of revocation servers* distributing revocation information regarding CPs. By design, an observer subscribes to these revocation servers as soon as it starts to operate. Specifically, an observer subscribes to receive information regarding certificates issued by principals mentioned by its TI list. Finally, if an enterprise formalizes its internal regulations as certified policies, an observer is primed with the CP embedding these regulations. (If an enterprise uses a different mechanism for enforcing internal regulations an observer should pass the request and the credentials to it.)

### 7.1 Certified Policy Enforcement

We explain now how CP enforcement is carried out. Consider that an agent **A** makes a request **R**, to a server **S**, which has an associated observer **O**. Assume further that **R** is accompanied by a certified policy **C**, and by a list (possibly empty) of subject-certificates. Then the following steps are taken:

1. Server **S** passes the request and the certificates to observer **O**.
2. **O** verifies that **S** is authorized to serve requests issued under the contract designated by **C**. This implies checking if **C** has been signed by a principal belonging to the list of trusted issuers (**TI**).
3. If this is the first request governed by **C** that is received by observer **O**, then **O** sets up an entry for **C**, in which it records the name, the status of the CP (valid or revoked), the current version, and the rules.
4. If the CP is valid, **O** checks if request **R** has arrived during its validity period.
5. If the request is accompanied by subject certificates then, for each such certificate **SC**, an attempt is made to confirm that **SC** is valid. That implies verifying that: (1) the signature is correct, (2) the certificate belongs to **A**, and (3) the certificate has not been revoked.
6. If **C** is stateful, **O** retrieves the control state from the `stateServer` mentioned by **C**.

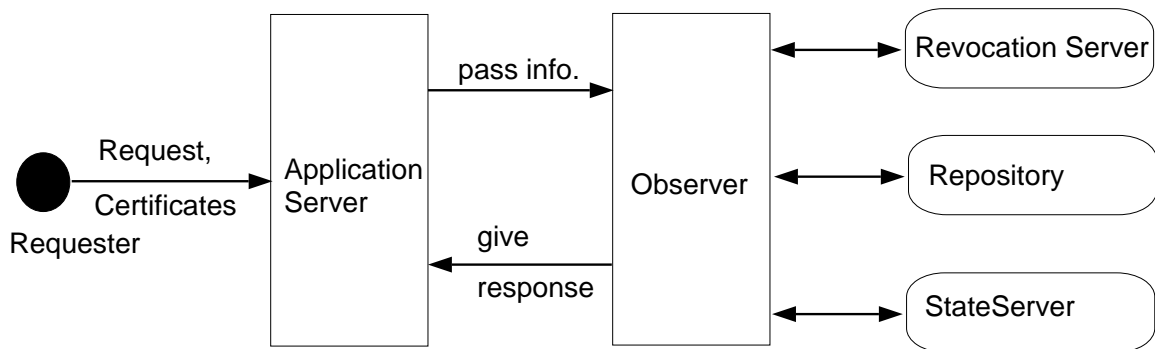


Figure 8: System architecture.

7. Observer  $O$  checks whether there is a rule in the CP authorizing the request and carries out the ruling. Additionally, if  $C$  is stateful,  $O$  returns the updated state to the `stateServer`.
8. If  $O$  has a certified policy embedding internal regulations, then  $O$  verifies that the request complies with this CP.
9. Finally, the server is informed of the outcome: if the request was authorized then  $S$  processes the request; otherwise, the request is discarded.

Note that if, at any time, the revocation server mentioned by  $C$  notifies  $O$  that  $C$  has been revoked, then  $O$  updates the entry of  $C$  accordingly. Moreover,  $O$  searches for an update of this certified policy. For this, it asks the repository mentioned by  $C$  for the latest version. Assuming that the repository has a newer version of  $C$ , then  $O$  updates the version number and the rules.

## 7.2 Implementation

In the current implementation we are using the Jigsaw [11] Web server, developed by W3 Consortium, which has been modified to communicate with observers. Observers are implemented mostly in Java, and operate as independent processes. Observers are *multi-threaded*, where each thread performs all steps associated with a request before accepting a new one. When a thread blocks (because, for example, the state is used by another thread/observer), another thread is chosen to run.

As a benchmark for observer performance we measured the average time taken by an observer to compute and carry out the ruling for a given request. Specifically, we have used purchase orders sent under the contract between TravelRus and FlyAway. The measured processing time, averaged over 1000 runs, is 3 ms. In the experiment the observer was running on a SUNW, Ultra-2 machine operating at 296 MHz, using Solaris 2.6 operating system and Java 1.2.

## 8 On the Efficiency of the Mechanism

To gauge the efficacy of CPs we run a simulation study driven by empirical data. As performance metric, the study uses response time, which is defined as the time interval from the moment a request arrives at the server and up until it ends processing. We evaluate the average response time incurred when using a CP-based mechanism, and we compare these results with the corresponding values resulting when using a conventional enforcing mechanism, employing monolithic policies (MP), obtained by *composing* contract regulations.

The general picture that emerges from this section is as follows: CPs are generally more efficient than conventional composition policies — markedly more efficient, in most cases. To be more specific: if 25% of requests are governed by stateful contracts, the average response time incurred when using CPs is 10 times smaller than when using composition policies. Moreover, in the MP-case response times could grow much higher when the percentage of requests governed by stateful contracts increases.

The rest of the section is organized as follows: We start in Section 8.1 by describing the server model used by the simulation. We follow in Section 8.2 by presenting the performance study.

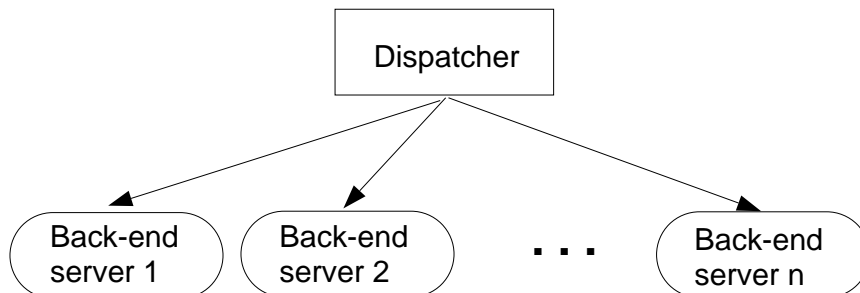


Figure 9: A cluster-based server.

### 8.1 The Server Model

The performance study considers that the server processing e-commerce requests is clustered. Cluster-based servers are commonly used for applications that handle heavy loads because they combine good performance and low cost. A cluster-based server consists of a front-end dispatcher and several back-end servers (see Figure 9). The dispatcher receives incoming requests and then assigns them to back-end servers, which in turn serve the requests according to some discipline. The simulation assumes that each back-end server has an associated observer, and that control states are maintained by a state server.

The study considers that two composition policies are constructed: (1) a stateless policy, which combines all contract regulations that do not take state into consideration, and (2) a stateful policy, which combines all contract regulations that use state information. There-

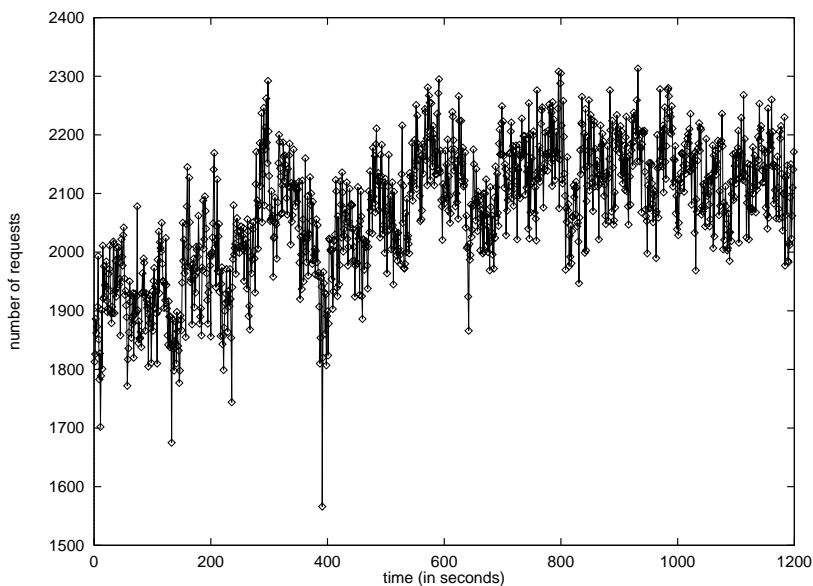


Figure 10: Number of request arrivals per second.

fore, in the MP case, there is a *single* control state, incorporating the dynamic information required by all stateful contracts. The processing of a request is done as follows: a back-end server/observer tries to authorize the request using the stateless policy; if the request is not authorized by the stateless policy, the back-end server employs the stateful policy, and consequently needs to retrieve the common, global state.

We point out that using two composition policies instead of a single one, improves the efficacy of the MP-model. If only a single composition policy is used, the processing of every request requires retrieving the state of this policy. This in turn, means that requests are processed sequentially, and that the parallelism introduced by using a cluster is entirely lost. In contrast, if two composition policies are used, a back-end server needs to retrieve the state only when the request is governed by a stateful contract.

The following assumptions have been made by the simulation:

1. The dispatcher assigns requests in Round-Robin manner.
2. The back-end servers are multi-threaded.
3. The overhead incurred by the dispatcher to select (job, back-end server) pairs is negligible.
4. The communication times between the dispatcher and back-end servers are negligible;
5. The communication times between a back-end server and the state server are negligible.

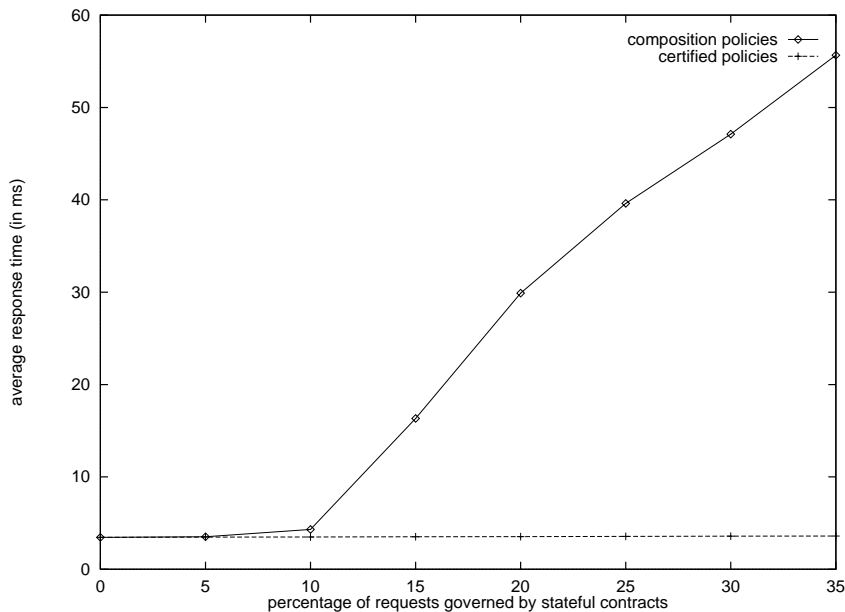


Figure 11: Average response time as function of percentage of stateful requests.

## 8.2 Experimental Performance Study

The simulation model considers a cluster of 16 back-end servers and 1000 contracts. The study uses one of the data traces of request arrivals at a Web cluster, maintained by the Internet Traffic Archive<sup>4</sup>. Figure 10 depicts this trace, which contains approximately 2.5 million requests arriving over 1200 seconds. The contract governing each of the requests was randomly chosen from the aforementioned set of 1000 contracts.

The service time of a request is estimated as the sum of the time to verify compliance with contract terms, the time to service the request, and the time to establish and close a connection. The parameters used by the simulation have the following values: (1) in both CP and MP cases, the time to verify compliance with contract terms is taken to be 3 ms (see Section 7.2); (2) the time to establish and close a connection is taken to 0.15 ms (this value was empirically measured in [32]); (3) the time to serve a request by a CGI-program is taken to be 0.6 KB/ms (this value was empirically measured in [38]).

First, the simulation study examines how performance depends on the percentage of *stateful requests*, i.e. requests governed by stateful contracts. Figure 11 displays average response times over the entire time horizon of the trace, when the percentage of stateful requests varies between 0% and 35%. The figure shows that CPs perform far better than MPs when the percentage of stateful requests exceeds 15%. For example, if 25% of requests are stateful than the average response times incurred by CPs and MPs are approximately 3.5 ms and 40 ms, respectively. In fact, when enforcement mechanism employs MPs the average response time increases (and the performance decreases) almost linearly with the

<sup>4</sup><http://ita.ee.lbl.gov/html/traces.html>

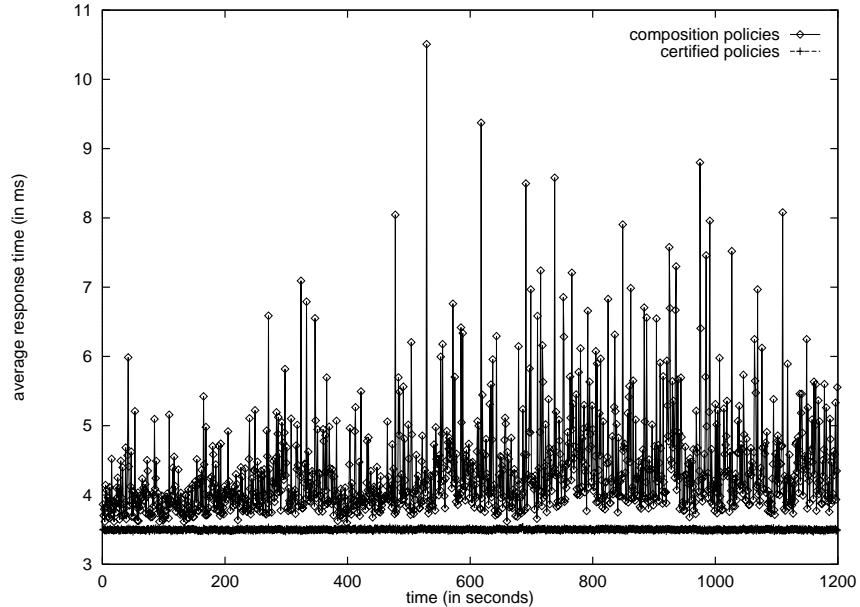


Figure 12: Average response time as function of time for a cluster-based server when 10% of requests require state for authorization.

percentage of stateful requests. This result is explained by observing that when the number of stateful requests grows, more observers request quasi-simultaneously the state, and thus they wait longer and longer for it.

In contrast, the performance of a mechanism employing CPs is barely affected by the percentage of stateful requests. This is because when the number of contracts, and consequently of CPs, is large then, with a high probability, at any given time the requests to be processed are governed by *different* CPs. Therefore, the state of a CP can be retrieved without delay, and only rarely request processing is slowed down because the state is used by another observer/thread.

Next, the simulation study examines how the response times performance varies with time, for a fixed value of the percentage of stateful requests. Figure 12 displays average response times in successive 1-second intervals when 10% of requests require state for authorization. This particular value has been chosen because, in this case, CPs and MPs perform similarly over the entire time horizon (as depicted in Figure 11 the average response times in the case considered are, respectively, 3.5 ms and 4.3 ms).

Figure 12 shows that average response time in MP-case varies considerably with time, while CPs performance is relatively stable. For certain time intervals, CPs outperform MPs by as much as a factor of 3 (for example, in the 529th interval, the average response time obtained when using MPs is 10.5 ms, while the average response time obtained when using CPs is only 3.5 ms).

The variation in performance in the MP case can be explained by bursts of stateful requests. Whenever such bursts occur, back-end servers wait longer to get the state, and

consequently, response time increase. It can be thus seen that the performance of MPs can degrade locally, even for small percentages of stateful requests.

In conclusion, the experiment shows that employing composition policies, and consequently having a single, composite state, considerably hurts the performance of a cluster-based server, because it forces sequential processing of stateful requests. In contrast, a mechanism using CPs takes full advantage by the parallelism introduced by the cluster. Moreover, we expect that the actual performance of CPs relative to MPs to be even better. This is because in practice a composition policy would take longer to evaluate than a certified policy due to its increased complexity. (The simulation study does not consider this factor, and it uses in both cases the same value.)

## 9 Related Work

Current e-commerce research encompasses several areas, including contract modeling, payment mechanisms, and contract enforcement. Modeling work (see, for example, [2, 21, 23]) aims to develop formal languages for expressing contracts between enterprises. Such languages provide agents participating in e-commerce with a common understanding of contract rules, a necessary step in negotiation stage. However, these formalisms do not attempt to enforce a contract, once it is formalized.

Several mechanisms have been designed that provide for secure and efficient transfer of funds (e.g. [19, 22, 33, 36]). Such means are indeed necessary for e-commerce, but they are not sufficient. Commercial activities are not limited to simple exchange of funds and merchandise between a client and a vendor—they often consist of transactions that need to be carried out in accordance to a certain policy.

There has been a growing interest in enforcing regulations embedded in e-commerce contracts, and a number of different, and quite powerful, enforcement mechanisms have been devised, like for example [1, 12, 13, 29, 35, 40, 42]. However, to the best of our knowledge, none of the frameworks proposed so far, embed contract regulations in certified statements, nor do they deal with contract annulment or revision. We will briefly review general access control frameworks designed to support independently stated, evolving policies.

We are aware of only two other works that explicitly consider the issue of policy update in a distributed setting. The first approach, described in [37], calls for a synchronization protocol, essentially a two-phase commit, to be performed by the servers employing the policy to be updated. Since this method requires to abort the synchronization protocol if any of the servers is faulty, it cannot guarantee when a revision becomes effective. In fact, the time to carry on a revision can be arbitrarily large. The second work, presented in [9], assumes that actions performed by a task are authorized by a security server assigned to that task. To establish a new policy, or to update an old one, a new security server is added to the system. This implies that only tasks created after a revision has been established operate under updated policy rules, while others continue to be regulated by the older version(s). But such a solution might is not usually acceptable in e-commerce applications.

There are a number of systems that support disparate policies, and which use certificate

framework for their dissemination. Among them, Access Control Programs (ACPs) [39], PolicyMaker [7] and KeyNote [6], assume that a certificate contains *both the identity of a subject* (given as his public key), and the *permissions delegated to him by the issuer* of the certificate. In these frameworks, establishing a policy requires issuing certificates for all agents having privileges under the policy in question. For example, establishing the example introduced in Section 2.3 requires creating certificates for all travel agents working for TravelRus.

We argue that this approach makes policy/contract update and annulment difficult to carry out, especially if the number of agents, which have rights under a given policy, is large. Annuling a policy established by these means requires revocation of the certificates of *all* agents having privileges under the policy in question. In our framework, annuling a contract requires the revocation of *only one* CP. Similarly, if a policy needs to be revised, new certificates for *all* agents affected by the change have to be issued. As an example, consider the revision introduced by point 2 in Section 4, which stated that TravelRus established a new certifying authority. In the frameworks mentioned above, this minute revision is implemented by issuing new certificates for *all* travel agents. In the framework proposed here, only one CP has to be revoked, and only one CP needs to be issued to revise a contract.

We will further describe two other systems—ABAC [26] and Binder [14]—which also use certified statements for distributing control information. Unlike the body of research mentioned above, these systems extend considerably the scope of certificates. In ABAC (attribute-based access control) certified statements are also used to assert and to delegate attributes (roles). In Binder, arbitrary regulations (expressed as Datalog rules) may be certified and used in access control decisions. These features greatly enhance the expressive power of the models mentioned above, which can support a wide set of policies. However, neither of these approaches supports stateful policies. As such, a number of contracts, including the contract introduced in Section 5, cannot be stated in ABAC or Binder.

## 10 Conclusion

We have argued that existent access control mechanisms cannot adequately support large sets of contract regulations. To deal with this problem we proposed in this paper to certify the contract terms that can be verified automatically, and to use the certificate infrastructure for their support. This approach has several important benefits in e-commerce context. First, one does not have to maintain a dedicated server for each contract (or set of contacts). As such, the number of contracts, in which an enterprise is involved in, is no longer an issue. Second, it is easy to establish new contract provisions: all that is required, is to formalize them as CPs, and deploy them on repositories. Third, contract revision and annulment can be supported with great ease. Finally, a CP-based mechanism yields good performance. We believe that the proposed mechanism should be relatively easy and inexpensive to apply since it uses an infrastructure which is already in place, it is well known and well understood.

## References

- [1] S. Abiteboul, V. Vianu, B. Forham, and Y. Yesha. Relational transducers for electronic commerce. In *Symposium on Principles of Database Systems*, pages 179–187, June 1998.
- [2] A.S. Abrahams, D.M. Eyers, and J.M. Bacon. Mechanical consistency analysis for business contracts and policies. In *Proc of the 5th International Conference on Electronic Commerce Research (ICECR5)*, Montreal, Canada, October 2002.
- [3] C. Adams and R. Zuccherato. Internet x.509 public key infrastructure data certification server protocols. Technical report, Internet Draft, PKIX Working Group, 1998.
- [4] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo. An authorization model and its formal semantics. In *Proceedings of 5th European Symposium on Research in Computer Security*, pages 127–143, September 1998.
- [5] C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *Proceedings of 5th European Symposium on Research in Computer Security*, pages 51–66, September 1998.
- [6] M. Blaze, J. Feigenbaum, and A. D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). In *Security Protocols Workshop*, pages 59–63, 1998.
- [7] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.
- [8] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. of the Seventh ACM Conference on Computer and Communications Security*, pages 164 – 173, Athens, Greece, 2000.
- [9] M. Carney and B. Loe. A comparison of methods for implementing adaptive security policies. In *7th USENIX Security Symposium*, pages 1–14, 1998.
- [10] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.
- [11] World Wide Web Consortium. Jigsaw - the W3C's web server. website:<http://www.w3.org/Jigsaw/>.
- [12] B. Cox, J. D. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *First USENIX Workshop on Electronic Commerce*, July 1995.
- [13] A. Dan, D.M. Dias, R. Kearny, T.C Lau, T. N. Nguyen, F. N. Parr, M. W. Sachs, and H. H. Shaikh. Business-to-business integration with tpaML asnd a business-to-business protocol framework. *IBM Systems Journal*, 40(1):68–90, 2001.

- [14] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium in Security and Privacy*. IEEE Computer Society, 2002.
- [15] Economist. E-commerce (a survey). pages 6–54. (The February 26th 2000 issue).
- [16] Economist. Riding the storm. pages 63–64. (November 6th 1999 issue).
- [17] D. Ferraiolo, J. Barkley, and R. Kuhn. A role based access control model and reference implementation within a corporate internet. *ACM Transactions on Information and System Security*, 2(1), February 1999.
- [18] B.S. Fox and A. LaMacchia. Online certificate status checking in financial transactions: The case for re-issuance. In *Financial Cryptography*, 1999.
- [19] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The Millicent protocol for inexpensive electronic commerce. In *Fourth International World Wide Web Conference Proceedings*, pages 603–618, December 1995.
- [20] L. Gong and X. Qian. Computational issues in secure interoperation. *IEEE Transactions on Software Engineering*, pages 43–52, January 1996.
- [21] M. Greunz, B. Schopp, and K. Stanoevska-Slabeva. Supporting market transactions through XML contracting containers. In *Proceedings of the Sixth Americas Conference on Information Systems (AMCISS 2000)*, August 2000.
- [22] I. Grigg. Financial cryptography in seven layers. In *Proceedings of Financial Cryptography Fourth International Conference, Springer-Verlag LNCS 1962*, February 2000.
- [23] B. N. Grosf, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: Courteous logic programs in XML. In *Proceedings of the first ACM Conference on Electronic Commerce (EC99)*, November 1999.
- [24] J. Jajodia, P. Samarati, V. S. Subramanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings ACM SIGMOD Conference on Management of Data*, May 1997.
- [25] G. Karjoth. The authorization service of Tivoli policy director. In *Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001.
- [26] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the IEEE Symposium in Security and Privacy*, pages 114–130, 2002.
- [27] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 73–87, Oakland, California, May 2002.

- [28] P. McDaniel and A. Rubin. A Response to ‘Can We Eliminate Certificate Revocation Lists?’. In *Proceedings of Financial Cryptography 2000*. International Financial Cryptography Association (IFCA), February 2000.
- [29] N.H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 322–331, Amsterdam, The Netherlands, May 1998.
- [30] N.H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.
- [31] M. Myers. Revocation: options and challenges. In *Financial Cryptography, LNCS 1465*, pages 165–171, 1999.
- [32] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.
- [33] P. Panurach. Money in electronic commerce: Digital cash, electronic fund transfer and ecash. *Communications of the ACM*, 39(6), June 1996.
- [34] R.L. Rivest. Can we eliminate revocation lists? In *Proceedings of Financial Cryptography*, 1998.
- [35] M. Roscheisen and T. Winograd. A communication agreement framework for access/action control. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.
- [36] M. Sirbu and J.D. Tygar. Netbill: An Internet commerce system optimized for network delivered services. In *IEEE COMPCON*, March 1995.
- [37] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium*, pages 123–139, 1999.
- [38] Y.M. Teo and R. Ayani. Comparison of load balancing strategies on cluster-based web servers. *Simulation, The Journal of the Society for Modeling and Simulation International*, 77(5-6):185–195, November-December 2001.
- [39] M. Theimer, D. Nichols, and D. Terry. Delegation through access control programs. In *Proceedings of Distributed Computing System*, pages 529–536, 1992.
- [40] J. D. Tygar. Atomicity in electronic commerce. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC’96)*, pages 8–26, 1996.

- [41] V. Ungureanu. Regulating e-commerce through certified contracts. In *Proc. of the 18th Annual Computer Security Applications Conference (ACSAC 2002)*, pages 35–42, December 2002.
- [42] V. Ungureanu and N.H. Minsky. Establishing business rules for inter-enterprise electronic commerce. In *Proc. of the 14th International Symposium on DIStributed Computing (DISC 2000); Toledo, Spain; LNCS 1914*, pages 179–193, October 2000.
- [43] D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Transactions on Information and System Security*, 6(2):286–325, 2003.
- [44] R. Wilhelm. Publish and subscribe with user specified action. In *Patterns Workshop, OOPSLA*, 1993.
- [45] R. Wright, P. Lincoln, and J. Millen. Efficient fault-tolerant certificate revocation. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, November 2000.