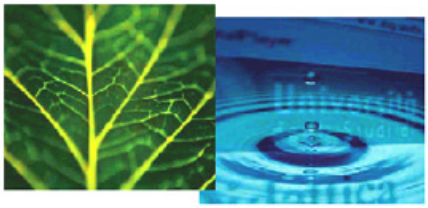
PhD Dissertation



International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Security Policy Enforcement in Service-Oriented Middleware

Gabriela Gheorghe

Advisor: Prof. Bruno Crispo

Committee:

Prof. Ernesto Damiani

Prof. Fabio Massacci

Prof. Mauro Conti

December 2011

Dipartimento di Ingegneria e Scienza dell'Informazione
Università degli Studi di Trento
Via Sommarive 14, I-38050 Povo di Trento, Italy
Email: gabriela.gheorghe@disi.unitn.it
Web: http://disi.unitn.it/~gheorghe

Acknowledgements

To travel hopefully is as important as to arrive. For one's doctorate, the dissertation is as important as the experience whose end it marks. They are both a test of one's motivation and hardest work. They are both – like, perhaps, yoga – a journey of exercise, self discovery, and learning.

The experience wouldn't have been possible without my advisor. I thank him for initiating me in what research means, and for his constant support and feedback throughout all this time. From him I've learnt how to form a perspective over the security field as a whole, and how to structure, position and present my work.

The good results and the fun couldn't have been possible without the friendly support of several colleagues. I have to thank Anja for her energy and good ideas; Stephan for being rigorous and diligent; Yudis for the very challenging discussions on science vs. engineering; Roberto for fun technical discussions in the nick of time.

I was lucky to encounter and work with a number of great people when I was in my internships. I need to thank Theo Dimitrakos for his enthusiasm and advice; Srijith for his helpful comments; Lieven Desmet and Wouter Joosen for encouragements; Frank Piessens for being very frank (no pun intended); Pierre de Leusse and David Brossard who have given me lots of useful hints and comments.

A lot of wonderful friends who have been there for me all this time and whom I thank sincerely are: Roberta, Gianluca, Nata, Alessandro, Csaba, Nauman for being constant adventure colleagues; Mihai and Andreea for being old friends indeed; hectic Stefano and Cristiano, for the greatest discussions over a beer. During my PhD I've made other friends from whom I've learnt lots of cool things that would help me afterwards: I thank Jose for being patient and for teaching me to do climbing; Agnes, for teaching me what chocolate really is, and how to be young no matter the age; Mandy and Sergio for being great house-mates; Fadi and Ada for their useful 2pm Sunday coffee habit; Matthew for introducing me to Skeptoid and a healthy skeptical attitude.

Last but not least, I thank my family and the Grasshopper, who have been there for me and missed me from a distance. They have my full gratitude for having been patient when I was working late, and for having encouraged me to do my best.

I thank you all!

Gabriela Gheorghe,
Trento, November 2011

The main foundations of all States, whether new, old or mixed, are good laws and good arms. But [..] you cannot have the former without the latter, and where you have the latter, are likely to have the former [..].

Niccolò Machiavelli, IL PRINCIPE, Chapter 12.

Abstract

Policy enforcement, or making sure that software behaves in line with a set of rules, is a problem of interest for developers and users alike. In a single machine environment, the reference monitor has been a well-researched model for enforcing policies. However, applying the same reference model in distributed applications is complicated by the presence of multiple users and concerns, and by the dynamism of the system and policies.

This thesis deals with building, assessing and configuring a tool for distributed policy enforcement that acts at application runtime. In a service-oriented architecture setting, the thesis proposes a set of adaptive middleware controls able to enact policies across applications. A core contribution of this thesis is the first message-level enforcing mechanism for access and usage control policies across services. In line with the idea that no security mechanism can be perfect from the beginning, the thesis also proposes a method to assess and amend how correctly a security mechanism acts across a distributed system. Another contribution is the first method to configure an authorisation system to satisfy conflicting security and performance requirements. This approach is based on the observation that policy violations can be caused by inappropriately fitting the enforcing mechanisms onto a target system. Putting these three contributions together gives a set of middleware tools to enforce cross-service policies in a dynamic environment. These tools make the user in control over continuous and improvable security policy enforcement.

Keywords security policy, enforcement, SOA, access control

Contents

List of Figures

Listings

Chapter 1

Introduction

Massively distributed applications, spanning networks and countries, are now the norm rather than the exception in the software industry. Facebook, Twitter, eBay, Amazon are just a few examples of private systems that put in motion tens or hundreds of thousands of servers, and that, in serving millions of users, deal with huge amounts of data and operations. A similar example is the Nationwide Health Information Network (NHIN)[1] – a super-network of applications and services across over 1000 US healthcare organisations. NHIN specifies a set of standards and policies for service-based healthcare systems; NHIN's initial projects are serving millions of patients [72].

A number of forces act simultaneously on these enterprise systems. With more clients, systems need to grow while being (at least) as efficient as before. To grow larger, enterprises have started to assimilate components offered by different, more specialised, providers. This is an economic choice, when it is cheaper to buy or rent a service, than to build it from scratch. To be efficient, performance techniques are now employed to reduce scaling overheads and maximise application performance. Along with the growth, systems need to adapt to changing demands and markets. This means systems should support more features and protocols, should provide security guarantees such as data integrity and confidentiality, and should tolerate faults and security misbehaviours to varying degrees.

From the financial to the healthcare domain, systems and data need to be protected against misuse, abuse or accidents. Since events like 9/11, stakeholders have become more aware of the need to protect their systems; equally, users have grown aware of the need to protect their data and activity. The common need for security guarantees has concretised into *security policies* and *policy enforcement*. Security policies specify allowed system or data usage; they usually cover authentication, authorisa-

[1] NHIN `http://healthit.hhs.gov/portal/server.pt?open=512&mode=2&cached=true&objID=1142`

tion, and data flow restrictions. Security policy enforcement means how to make sure that software behaves in line with such policies. Security enforcement controls, or mechanisms, are the software tools that perform security policy enforcement.

1.1 The Problems

Complying with security policies is equally important and difficult for a distributed application. Security compliance makes customers safe and happy and maintains a good reputation for the company. Yet the difficulties come from three main directions, which have not been properly addressed yet:

Building globally-consistent security enforcing controls. When you do not own all services in your system and consequently cannot verify their code, you need to build security policy enforcement controls over these services. But then, how to *react* in the face of a policy violation? How to ensure that a policy is enforced across the enterprise in the same way, to all services onto which it should be enforced? How to deal with policy changes so that the service code is minimally affected? When integrating third-party code, how to easily enforce new security policies onto it at runtime?

Assessing security enforcing controls. Either when you own the security enforcing controls, or when you use third-party ones, you need to know if the mechanism actually works as expected. So how to actually check that an existing control or set of controls *really* enforces a policy? What runtime proofs can you gather in order to assess how compliant is your application with a given set of enforcement controls? When do you need to replace security controls with 'better' ones?

Configuration of security controls. When out-of-the-box security controls need to be plugged into the application, what parameters of these controls need be tuned to adapt to the current context? Do these parameters have any effect on the overall security guarantees of the application? How do you choose among equally safe security enforcing controls? How to connect security controls and how to manage them consistently?

Answering these questions is invaluable to enterprise stakeholders, be them management or administrators. First, they would be able to manage security enforcement controls that evolve independently of the targets they are protecting. Second, administrators would have at all times control over the security mechanisms deployed on the application. This means that the administrators would be aware of security issues

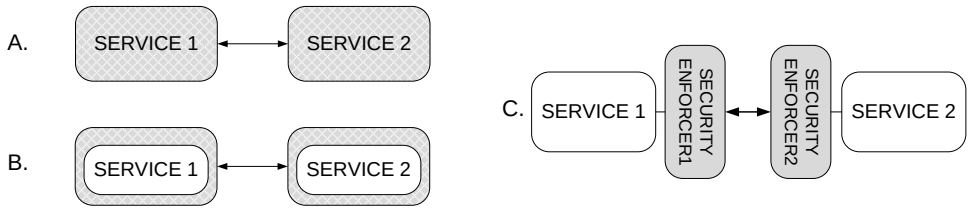


Figure 1.1: Different architectures for security enforcement: A) security code is woven with the business code; B) security wrappers around business code; C) security proxies protecting business code.

in real-time, could apply countermeasures and see if these countermeasures actually work. Third, security administrators would know how to compare out-of-the-box security controls, and chose and adapt those that are better suited for their applications.

Securing service-based enterprises is a problem left to each application owner to solve. This leads to *multiplication of security mechanisms* that realise the same functions, and that are well different within the same composite application. Figure 1.1 shows existing techniques to perform security policy enforcement in practice. The first approach (shown in Figure 1.1 A) is to weave the security code into the business code (or application logic), but this is problematic when the security code needs to change because e.g., the security policy changes, or an error is discovered in the enforcement code. The downtime incurred to the application to debug or change the policy code could be avoided by another technique: wrapping (shown in Figure 1.1 B). Security wrappers make policy changes independent of the service logic, and are associated with the service they protect. Yet wrapping is also problematic because it does not scale with the number of services and policies. Assuming there is one policy to enforce between Service 1 and Service 2 in Figure 1.1 B, then there would be two connections to enforce the policy for two services, six connections to enforce the policy on for three services – in general, $n(n-1)$ communication connections on which to enforce the same policy among n services. Using a mediator (shown in Figure 1.1 C) entity to manage communication links, as an enabler for service integration, would reduce the connections to be managed with an order of magnitude, and this idea has been already suggested in practice [70, 116]. Yet security mediators, or security proxies, are built in-house, isolated, and independent of the integration tools.

Security mechanism assessment is also hard. Proprietary security mechanisms only undergo limited in-house testing against generic security issues. When later the service is outsourced, together with its accompanying security mechanisms, then the in-vitro tests would no longer be relevant for the complex real-world context where the software is used. It comes with no surprise that, in these conditions, stakeholders would only know about the effectiveness of their (own or rented) security mechanisms

during audit reviews. Because the audit process only investigates for unwanted effects, then enterprise stakeholders can react in one of two ways: either re-implement the flawed controls, or ask for legal compensation from the partner supplying the security controls. In all, the management have to wait until the audit discovers if their system really enforces security policies. Then it might be too late to intervene.

It is also problematic to configure out-of-the-box security tools for enforcement, to replace previous controls that were found inappropriate during security assessment. This configuration means updating any available parameters of the links among distributed controls e.g., how the controls are distributed, how they communicate, and how they react to a policy violation. First, existing policy servers and security enforcement tools seldom provide the same parameters to set - each tool and suite provides its own set of customisable features, thus its own security guarantees. Second, security controls (e.g., wrappers) around application services use up the application's communication channels for transmitting security information. This is a problem because the communication links are vulnerable to attacks, consequently an attack to the link would impact both application and security mechanisms. In short, the security mechanism overlay is not always 'aware' of the application that it is protecting.

Assessing and configuring security controls have been treated separately because building security mechanisms is a task performed in isolation by each company. The main problem with this approach is global policy compliance. Take for instance Separation of Duties (SoD). SoD is a policy that is commonly required in service-oriented enterprises, requiring business tasks to be performed by more than one actor. For a hospital, SoD might impose that the doctor roles to request and approve a blood transfusion for a patient, are different. For the Red Cross, SoD might impose that the doctor roles that approve and distribute a blood donation to a hospital, are different. When the hospital integrates with Red Cross services, both institutions would continue to use their own SoD controls. But even if SoD is enforced *locally* by each of the two organisations, *global* SoD is overlooked: when doctors work for both hospital and the Red Cross, the same doctor can request blood transfusion and distribute the donated blood to the same hospital; or approve the transfusion and distribute the blood to their hospital, etc.

SoD is just one of the many policies that need to be enforced across service-oriented systems. The existence of such security policies stems from the existence of security regulations of legal frameworks in various industries. For instance, in healthcare, there are HIPAA [233] and HL7 [99]; in telecommunications, there are European Directives on data retention [61], protection [186] and privacy [230]; in banking and accounting, there are Sarbanes-Oxley [232] or Basel II [21]. Such wide-spanning reg-

ulations concretise into security policies whose enforcement cannot be left to each organisation to handle separately. This is because each organisation will reuse its existing controls, and these were built for 'selfish' protection: they do not care about the compliance of other players (they do not know about other systems at design time), but only protect their own. Hence, that security constraints are global, and that they cannot be enforced locally, these are two other reasons for unifying the security mechanisms to enact them.

1.2 Thesis Contributions

This thesis designs and implements a common framework to address the three problems (building, assessing and configuring mechanisms for security enforcement) together. We place our solution at the middleware layer, so that it can be independent of specific platforms while at the same time flexible and reusable from a service-oriented perspective. With the goal to realise unified, flexible and assessable enforcement of security policies, our concrete contributions are:

Building security controls. We define and implement an enforcement mechanism that enacts low-level security policies across several service endpoints in a service-oriented application. This approach has, by construction, the following advantages:

- it keeps security mechanisms independent of application logic, for which reason security management is separated from application evolution;
- it makes policy enforcement globally consistent: whenever policies change, or mechanisms change, the new updates will be seamlessly enacted;
- it can scale with an increase in policies and services because policies are enforced across multiple service endpoints, from a central point;
- it is quite powerful because it acts at the level of communication links. The enforcement actions span a broad range: unlike other approaches, our framework can not only block, but also modify, duplicate or delay messages on the infrastructure. Also, it can trigger complex enforcement actions when the situation and context require to do so.

These ideas are at the basis of xESB, an instrumented message bus for policy enforcement, that has been published in [83] and re-discussed in [82]. Another novelty that our mechanism introduces is enforcement across abstraction layers. We combine message-level and business-process level mechanisms to obtain a

wider range of reactions to security violations; to our knowledge, this has not been done in enforcement, until now of an all-or-nothing nature. This contribution has been published in [80].

Assessing security enforcement controls. We realistically consider that violations cannot always be prevented. Consequently, we propose and implement a method to assess how well a security enforcement mechanism achieves its goal, at runtime. This aspect is quantified on a scale whose thresholds have been set by the security policy specification, and is called a *security enforcement indicator*. We design and implement a runtime feedback loop so that whenever the indicator surpasses critical values in the policy, a set of compensations is triggered. In this way, continuous system (re)assessment can reveal if the compensations actually worked, and if the implementation of the security policy enforcer is actually correct. This design has been patented as patent application EP 250 684.5, filed to the European Patent Office.

Configuration of security controls. We propose and implement the first middleware tool to manage security controls deployed on a service-oriented application. This management deals with setting the connection parameters of the security components whose task is to enforce a policy. Until now these connections have been considered to be fixed, but with our approach, it is now possible to reconfigure or replace existing controls with 'better' ones. This can be a consequence of the runtime assessment process and can also happen at runtime. The motivation for runtime reconfiguration is that the controls should evolve (in terms of a set of parameters) along with the application. These discussion and results have been published in [79].

In all, this thesis designs, implements and analyses the first middleware framework for enforcing security policies across applications endpoints, at runtime. This framework allows for customised reactions to security events that spans several abstraction levels, can assess its assurance levels on the fly, and can adapt to an evolving application context. This can be an invaluable tool in the context of today's distributed applications, for both security management, and security administrators.

The work reported in this thesis has been made possible within the FP7 project MASTER (Managing Assurance, Security and Trust for wEb Services). Within the same European project, we have published another paper on a capability and maturity model [81]; on the more practical side, we have conducted the integration of xESB within the MASTER prototype and the result was demonstrated in an international conference [42].

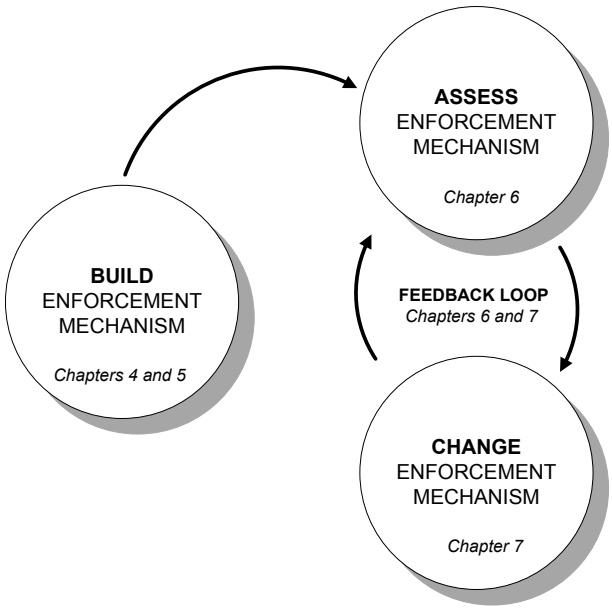


Figure 1.2: The flow of the chapters in this thesis, and where their individual contributions stand.

1.3 Overview

This dissertation is organised as follows. The scene is set in Chapters 2, and 3. Chapter 2 describes the background concepts and technologies that are used throughout the thesis: Web services and service-oriented applications, security regulations, security policies that stem from regulations, and security policy enforcement as the main focus of our work. Chapter 3 presents the scenario used in this thesis: the healthcare domain, governed by a number of security regulations whose implementation is challenging to tackle.

The main chapters from now on are organised as visualised in Figure 1.2. Chapter 4 presents xESB, the first concrete message-level policy enforcer mechanism for service-oriented systems. xESB can enforce access control policies on communication across several Web service endpoints, and has a central point of control over the whole system. Using xESB, instead of existing separated controls, has the advantage that is easier to manage, that it scales with the number of policies, and is completely independent of the application logic behind each service endpoint that it protects. We discuss the usage of xESB with two policy languages, and we extend its enforcement capabilities with external tools. In this way, we show that xESB provides great flexibility and customisability to the security mechanism developer.

Chapter 5 presents an approach of performing policy enforcement across abstraction layers. We start off from the limitations of xESB to process (and hence react to)

the full content of communication flows, and the limitation of existing business process enforcement to control message flows at all. We propose a way to compensate the limitations of each enforcement approach by means of a central enforcer mechanism that would combine the strengths of the two approaches together. The result is a more flexible enforcing mechanism, whose enforcement capabilities span both levels.

Chapter 6 proposes and implements a method to quantify how well security mechanism enforces a policy, by analyzing message-level logs. The measure used for this quantification comes with what we call an enforcement indicator. The indicator is useful for administrators and management to verify at runtime how well the system performs in terms of security enforcement. If the value of the indicator surpasses the critical limits set within the security policy, then a feedback loop is triggered by which the system reacts to the misbehaviours that the indicator had detected in the first place. A possible reaction is to replace the enforcer mechanism with another one, or, as the next chapter describes, a reconfiguration of the existing controls.

Chapter 7 analyses and implements a middleware tool that manages the connections among the security infrastructure elements. We show how inappropriately connecting security mechanisms together can result in wrong security decisions. We implement a configuration manager that can compute 'better' ways to connect enforcement components (i.e., configurations); the manager can, with the help of a middleware such as xESB, enact the found configurations onto a running system.

Chapter 8 presents the techniques of enforcing a policy in a single-machine system, and also several distributed enforcement approaches. Parts of this chapter have been published in a technical report [78]. Chapter 9 concludes by stating that the features presented in Chapters 6 and 7 can make xESB the first adaptable enforcement mechanism that is aware of the application that it protects and that evolves along with it, by reassessment and reconfiguration.

Chapter 2

Background

Everything should be made as simple as possible, but no simpler.
Albert Einstein, *On the Method of Theoretical Physics*, Herbert Spencer lecture, Oxford, June 10, 1933

In order to better understand our approach, this chapter provides an overview of main technologies and concepts we use. We start from surveying service-oriented architectures and some of the most prominent SOA middleware technologies. On this setting, we look at some of the European and US regulations on securing data and software behaviour in service-oriented applications spanning multiple network and ownership domains. Since such regulations need to be respected, they are translated into security policies that need be implemented in software. After summarizing the essential traits of access and usage control policies, we define policy enforcement along with some of the mechanisms by which it can be achieved. We then describe some policy management aspects, and lastly some of the methods to assess if the deployed security mechanisms are functioning properly.

2.1 Web Services

A Web service is a program on the Web that can interoperate with other Web services irrespective of the platform or language in which they were built. W3C defines it as "*a software system designed to support interoperable machine-to-machine interaction over a network*". A Web service accomplishes an established functionality and has an interface made public through a description language. Based on two alternative technologies (on whose advantages there is an ongoing debate), there are two big types of services:

- *WS-* services* also known as *WS-heavy* or *Big* services: is the approach that combines Simple Object Access Protocol (SOAP) with Web Service Description Lan-

```
POST /sampleService1 HTTP/1.1
Accept-Encoding: gzip,deflate
Host: localhost:8080
SOAPAction: "http://www.vordel.com/sampleService/Add"
User-Agent: Jakarta Commons-HttpClient/3.1
Content-Type: text/xml;charset=UTF-8
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:sam="http://www.vordel.com/sampleService">
   <soapenv:Header/>
   <soapenv:Body>
      <sam:Add>
       <sam:bbb>34</sam:bbb>
       <sam:b>34</sam:b>
      </sam:Add>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 2.1: A SOAP message with the HTTP binding.

guage (WSDL), usually over HTTP (but UDP and SMTP also possible).

- *RESTful services* use Representational State Transfer (REST) instead of SOAP, with Web Application Description Language (WADL), over HTTP.

The first approach uses WSDL – a description language based on XML, now in version 2.0 [247]. WSDL describes a service container and endpoint addresses, ports, operations along with the data structure needed to invoke an operation of the service. A Web service communicates with consumers via platform-independent XML messages, in a SOAP format [250]. A SOAP message transmitted via HTTP is shown in Listing 2.1; it consists of HTTP binding metadata and the SOAP envelope with a SOAP header and a SOAP body. Error or status information within a SOAP message needs to be wrapped in a SOAP Fault element. Despite having become a standard since 2000, and integrate-able with other Web service standards like WS-Policy, WS-Addressing, WS-Federation, etc, SOAP has received massive criticism in terms of complexity and performance [71].

The architectural alternative has been Representational State Transfer (REST), described by Roy Fielding in his dissertation [67]. REST also uses HTTP as the transport protocol, but unlike SOAP, has a uniform interface to access services; this interface employs the actions of HTTP 1.1, primarily GET, PUT, POST, DELETE. A RESTful Web service is easier to build and lighter in terms of XML content, only requires XML for its invocation, along with a unique URI. In a standardisation attempt, W3C has proposed WADL as an alternative syntax to WSDL, providing a “machine process-able description of HTTP-based Web applications" [248]. With an ever-growing interest in

REST, it has been stated that the two protocols can coexist: while REST seems more appropriate for simple integration scenarios, SOAP and WS-* standards consider more advanced quality of service aspects for enterprise applications [187].

Irrespective of the technologies used, nowadays IT businesses make their services commercial in that service usage costs, and is regulated by contracts that usually take the form of *Service Level Agreements* or SLAs. A SLA is a contract established between a service provider and a service consumer, with the purpose of clarifying what is meant by performance for delivering the service, and respectively, what thresholds are required by the consumer and what are those acceptable for the provider.

2.2 Service-Oriented Architectures

In order to connect and use a Web service, a program must know the Web service's URL and its WSDL description. A simple way to manage Web service addresses and interfaces, was to create Web service registries, much like telephone books. Just like a caller gets to know the telephone number of somebody they need to speak to by looking it up in a registry, the same worked for using a Web service. Automating the way in which Web services find out about other services and communicate with them, gave rise to concepts such as service-oriented applications and architectures.

Service-Oriented Architectures (SOA) are a set of concepts and principles that describe the design, integration and use of Web services that are distributed across applications and networks. As a mature IT paradigm with over ten years of existence, SOA makes the central target of investments in current application infrastructure worldwide, and these investments are still growing compared to previous years, according to Gartner [76]. Much highlighted SOA applications include the Cloud, defined by NIST as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services)" [161], and Web 2.0, as a set of principles and practices whereby the Web is a platform of scalable services where users are co-developers [178].

OASIS has issued a reference model for SOA in 2006 [169], and a reference architecture in 2009 [173]. The reference model is described in terms of *capabilities*, defined as the set of tangible effects triggered by a consumer on a producer's side; such effects can be either a direct exchange of information, or a change in a shared state known across the system. Two other notions in the model are *services* as enablers of capabilities, and *interactions* between service producers and service consumers by which consumers' needs are matched with the capabilities of the providers. The refer-

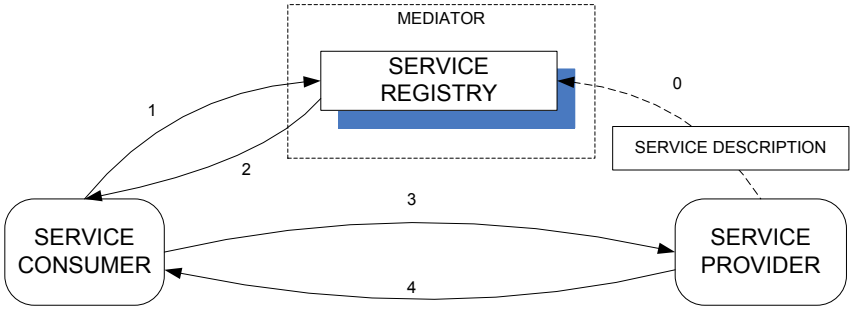


Figure 2.1: Basic SOA interactions: (0) a Service Provider publishes its service description to a mediator – a Service Registry. As a result of querying the mediator for a particular service (1), the Consumer will get the data from the description (2) allowing it to bind to the Provider directly (3). The Service Provider will then reply to the Consumer (4).

ence model defines a service as "the mechanism by which needs and capabilities are brought together". A service has a *service description* to give the service semantics (i.e. what effects it produces, and on what conditions). The set of possible interactions by which a service provides its capabilities is given in the *service interface*, and the basic interactions are shown in Figure 2.1. In order to be used, services need to be visible and reachable to their target consumers. Which are their target consumers and how they are allowed to use the service are described in service policies and contracts, mentioned in [173]. With these definitions, a SOA application is a loosely-coupled system whose atomic entity is the service. World Wide Web Consortium (W3C) defines SOA as a distributed architecture whose salient features are [249]:

- **logical view** refers to the abstract nature of services, whereby the task that a service performs is separated from the way it is performed; the focus in SOA is on business tasks, made up of one or more services;

- **message orientation** refers to the message-based communication among service consumers and providers in a SOA system;

- **description orientation** means that capabilities are described by data that can be processed by a program; this data must fully document the service exposed;

- **platform independence** stresses the interoperability of SOA systems: service consumers and providers can communicate independently of any particular platform or implementation. This is achieved by using XML and service interfaces.

2.2.1 SOA Layers

Based on different categories of architectural concerns, the literature separates among several SOA layers [116, p.34], [124, p. 82],[14, 182, 173]; even though there are some differences in how different authors present them, the common layers are as follows (in a top-bottom order, where top is the closest to business targets):

(L4) The Management or Presentation Layer. This layer includes tools to visualise the activity of the enterprise application.

(L3) The Business Process Layer. This layer is in charge of coordinating services and monitoring their activity from the point of view of business semantics. Service composition and choreography are defined at this layer.

(L2) The Services Layer. This layer exposes the services used by the business. Mechanisms ensuring service discovery, selection, binding, publication reside here.

(L1) The Technical Infrastructure Layer. This layer covers the enterprise components that host the application environment, ensure the quality of service, describe the network topology.

The existence of these layers is reflected by current service-related *middleware* tools. *Middleware* has been defined by Bernstein as all distributed system utilities that "sit in the middle" between the operating system and industry applications [27]. Some examples of middleware in SOA scenarios are: Globus [3] and Shibboleth [105] for Grid (L1-L3), IBM Tivoli [102] for managing enterprise applications (L1-L4), Layer7 Policy Manager [134] and Axiomatics Policy Server [17] for messaging at the L2 level.

Relation to this thesis The work reported in this thesis is not limited to a single layer from those presented above. For any enterprise application, security concerns pervade all of the SOA layers, and this has been stated before [14]. One of the points we make is that good security practices should include security mechanisms spanning all these SOA levels.

2.2.2 Enterprise Service Bus

The Enterprise Service Bus (ESB) is a middleware for mediation between service consumers and producers. As an integration architecture among individual services as well as composite applications, the ESB was designed to be the communication backbone among disparate enterprise components, that no longer have to care about technological disparities. That is, the primary goal was that the ESB ensures that services build on different communication protocols can communicate seamlessly. Originally,

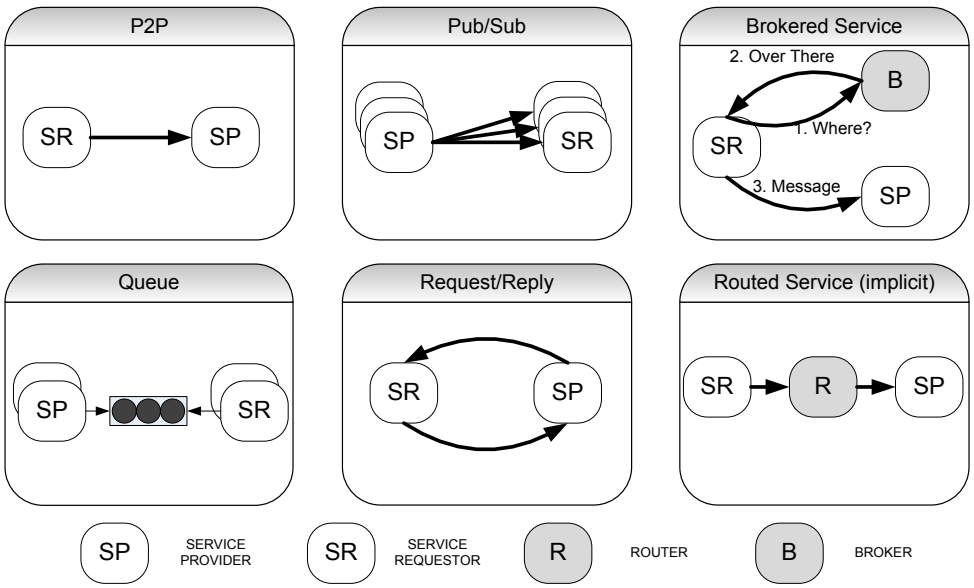


Figure 2.2: SOA classic messaging patterns, taken from Juneja et al. [116].

the ESB would cover layers L1 and L2 of the above; recently, however, current ESB technologies have gained L3 features as well.

It is difficult to accurately define the ESB. Juneja et al. separate among three different definitions for the ESB [116]: the first, saying that the ESB is a set of software patterns for service integration; several messaging patterns (publish/subscribe, queuing, routing, brokering, and request-reply) are shown in Figure 2.2. Another definition sees the ESB as a messaging bus environment whose task is to deliver messages from end to end; a third definition sees the ESB is a software container of services. In this thesis, we use the second definition whereby the ESB is an L1/L2 routing medium.

Most of the current open source ESB tools adhere to a standard called Java Business Integration (JBI), for the deployment and management of services on the ESB [115]. This is a specification of how service components should be packaged into containers, how such containers can be assembled into more complex containers, how such containers can be managed so that it becomes easy to port them and to substitute components with others, and how to manage the lifecycle of such containers. Figure 2.3 shows the major JBI elements: *service engines* (the services hosted within the current environment), *binding components* (connectors to external services), and the *normalised message router*, a centralised mechanism whose task is to route messages among services, as long as these messages adhere to the same format (i.e. they are 'normalised'). The patterns of message exchange between a service consumer and a service provider as inspired from WSDL 2.0:

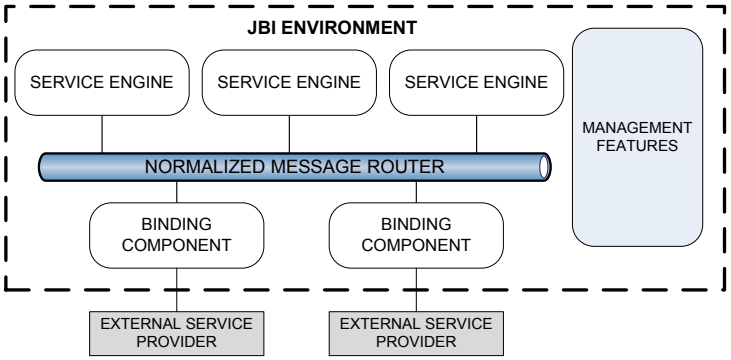


Figure 2.3: The Java Business Integration standard architecture.

1. **In-Only** means that the consumer sends a message to a provider who only replies with a status;
2. **In-Out** means that the consumer sends a message to a provider, and the provider replies with a message or fault, and the consumer replies with a status;
3. **Robust In-Only** means that the consumer sends a message to a provider, the provider replies with a status, or a fault; if the reply was a fault, the consumer replies with a status;
4. **In Optional-Out** is similar with In-Out, but the provider need not reply [115].

Relation to this thesis The work reported in this thesis employs two ESB-related tools: ServiceMix ESB, an Apache open source platform that adheres to the JBI paradigms [11]; and Vordel, a proprietary policy-based message gateway also referred to as a 'lightweight ESB' because of its message mediation capabilities [239]. Vordel is actively used in industry.

2.2.3 Business Process Management

The SOA reference architecture [173] defines a *business process* as a sequence of activities that when performed, can have a result that is important for the business. These activities can be arranged in a certain order or flow, combined or split; the result of these operations can in itself be a service, or an aggregation of services. The major benefit of SOA is that the business units of work, abstracted into either individual services or compositions of services, can be (re)used across systems and organisations [116]. Some examples of overlapping components and functions in healthcare systems range from patient registering, to documenting problems and diagnosis, ordering medication, to creating clinical notes, claims and bills [116, p. 208].

Separating the business-relevant services from their technology details allows organisations to concentrate on the functionality (*what* the services can do) rather than the means to achieve it (*how* to deliver the functionality, e.g. what protocols to use, what data formats to chose for inputs and outputs, etc.). Reusing services to deliver meaningful business outcomes relies on two main aspects: (1) their visibility, or discoverability, and (2) on how they can be combined to make up business flows. While discoverability is handled by means of service registries and standard protocols like Universal Description Discovery and Integration (UDDI) [168], the second aspect has been associated with two concepts: *orchestration* and *choreography*. *Orchestration* has been defined by the SOA Reference Architecture as a technique of service composition whereby a single entity – the orchestrator or the conductor – executes and coordinates the business process flow [173]. Unlike orchestration, *choreography* is neither hierarchical nor centralised: it is a service or business process composition technique where the participants are all equal and exchange messages in order to fullfill the same business goal [173]. OASIS standards are also present in this area: the Web Services Business Process Execution Language (WS-BPEL, hereby BPEL) is a Web service orchestration language to specify and execute business processes with Web services [170].

Relation to this thesis While focusing more on security mechanisms at the message-level (L1 and L2), this thesis also examines security mechanisms at the business process layer. Chapter 5 will present a comparison between what can be done at business process level and what can be done at message-level to enforce a security policy.

2.3 Security Regulations

The SOA reference architecture [173] defines the term *SOA governance* as a type of governance for IT systems, that is, the set of practices, regulations and processes aiming to guide, apply and monitor the behaviour of the informational system so that it achieves its IT objectives. One of these objectives is for the organisation to be compliant with security requirements, either its own, or regulations common to certain countries or industries. Such regulations aim to protect against misuse of user data or of information systems that process user data and that respond to user actions. Issuing these regulations is complicated by the complexity of current technologies: the existence of business domains under different legal administrations, different or even conflicting objectives and different methods to achieve them. For instance in Europe, the European Union has issued a number of security and privacy regulations:

- Directives 95/46/EC [62] and 97/66/EC [59] aim to protect personal data without the subject's consent, either in healthcare or in the telecommunications sector;

- Directive 2002/58/EC concerns the way personal data is processed by services in the electronic communications sector so that users' data receives the same protection level irrespective of the technologies used [60]. This directive refers to authorization to access data, protecting data against loss and tampering, and the enforcement of a security policy to process sensitive data;

- Directive 2006/24/EC, also known as the "Data Retention Directive", stipulates a maximum retention period of between 6 and 24 months for users' telecommunication data [61]. The communications data that needs to be retained is not the content of the communication but rather the duration, and identification information of the parties. This regulation has been subject to controversy.

In the US, security and privacy regulations have been issued in the US Constitution, in federal laws, and in the commercial sector [232]. Also, Directive 95/46/EC [62] has been reflected into further regulations like the UK's Data Protection Act [186] on how individuals can control sensitive information about themselves. All these examples show that data protection regulations in the economic sector are based on fundamental legal rights; therefore, a system that complies with such regulations can guarantee protection for its IT assets (e.g., data, infrastructure) and clients as well. Other sets of regulations that deal with security aspects is the Health Insurance Portability and Accountability Act (HIPAA) of 1996 [233] and Health Level 7 (HL7) [97] in healthcare; they will be presented later in Chapter 3.

Relation to this thesis Such legal constraints – from regulatory frameworks that apply to individual economic sectors, to security and privacy protection – impact the internal structure of the enterprise and the way it runs its business. Security policies stem from such regulations, and target inner mechanisms of the organisation and system (from IT system architecture to staff responsibilities, and work ethics) but also the interactions between different entities and systems. This thesis deals with how to ensure than an IT system is compliant with such policies, and how to raise awareness and control if it is not.

2.4 Security Policies

For a software application, security and privacy regulations, along with any internal security requirements, are translated into security policies. Regulations and laws are usually generic, since they are formulated for a set of applications in a particular IT sector. They can only go as far as to ask, for instance, for "*protection of electronic means and data against unlawful data processing operations, unauthorised access and specific soft-*

ware" [16, Section 34,e]. This phrase cannot define what "unlawful operations" are, what is "unauthorised access", and what is "specific software". Hence these concepts need to be customised for every enterprise application that seeks to follow such regulations. This translation, from a regulation to one or more implementable policies, can be done with the help of security experts: is not completely automatable since it needs to understand natural language and the semantics of the application under discussion. This problem has been approached before as early as 1993: Maullo and Calo refine an organizational policy that covers organization goals, to various intermediate steps that reach procedural policies that can be executed by the system [148]. In this thesis we assume that security policies have already been inferred from regulations.

The term security policy (hereafter simply 'policy'), bears several definitions. Bishop sees a security policy as "a specific statement of what is and is not allowed. [..] If the system always stays in states that are allowed, and users can only perform actions that are allowed, the system is secure. If the system can enter a disallowed state, or if user can successfully execute a disallowed action, the system is nonsecure." [30]. Based on the notion of a *security policy objective* defined as a statement that protects a concrete asset from unauthorised use, Sterne separates between organisational policies and automated policies [220]. In his view, an *organisational policy* is a set of rules and practices that stipulate the ways in which "an organisation manages, protects and distributes resources to achieve specified security policy objectives"; an *automated security policy* is an implementation of the constraints of the organisational security policy, in a computer system. We take on the viewpoint of Sterne, in that regulations and laws directly impact the security objective of an enterprise application, whilst (automated) security policies are in their turn related to the security objective.

A more formal definition was set by Schneider: a security policy is a program execution that is disallowed, and this is specified by means of boolean predicates on sets of executions [207]. Schneider and others separate among four kinds of policies:

- *access control* policies focus on restricting the operations that subjects perform on objects, for instance "Alice can only access the files in the folder /users/alice";
- *information flow* policies focus on the flow of data among system resources, in that information should not be leaked to parts of the system that are not controlled or trusted; for instance "no data from the passwords file should reach the network";
- *availability* policies ensure that a program that has control over a resource at runtime, should release it at some point in the execution;
- *bounded availability* is a variation of the availability policy that puts a bound on the amount of time or usage of a resource that has been acquired by a program

and needs to be released at some point [22]; for instance “the CEO should have access to any available printer within maximum 3 minutes from their request".

Later, Schneider, Morrisett and Harper show that there are several classes of policies, out of which some are properties, i.e. a criterion over an individual program execution, that can be elicited by monitoring the program at runtime (with what is called an *execution monitor*) [208]. While a taxonomy of policies is given in [208], a taxonomy of properties is given in [142].

Relation to this thesis In this thesis we are concerned with mechanisms that can enact security policies at program runtime. Since information flow policies and availability policies have been deemed not to be properties [207] and hence not completely enforceable at runtime, we continue with a summary of access control and usage control policies that make the target of the enforcement mechanisms of this thesis.

2.4.1 Access Control

Access control is the process of mediating all requests to a resource and determine whether they should be allowed to proceed [202]; or, slightly differently, access control includes the models and mechanisms that restrict subjects from using system resources. A *subject* is a process or service that runs on behalf of a human user. *System resources* can be Web services, system devices, network connections, programs, but can also be lower-level resources like disk, or CPU. An *access control policy* is a security policy that specifies which entities can access certain system resources [30]. Since the constraints, or rules, enumerated in an access control policy require the existence of an entity to enact them, this entity is termed *access control mechanism*. An access control or security mechanism defines the implementation of the controls stated in a security policy [202], and formally represented in a *security model*.

The term “*access control*" is connected with those of “*authorization*" and “*authentication*". *Authentication* refers to the processes by which a user’s identity is verified when that user attempts to use system resources. The identity is proven with credentials e.g., user id and password, digital certificate, or biometric data. Still, proving that a user has registered into the system at some point in the past is not enough to be granted access to a resource, since the user may not have the necessary rights to obtain such access. Hence, the term “*authorization*" refers to the processes that check if the authenticated user has enough permissions to access a resource. Such checks span from verifying group membership, to checking an access control list. Conversely, *access control* is a more general term that includes authentication, authorization, and audit (as a way to verify a posteriori the user’s actions).

The first abstraction in access control was the **access control matrix** [132]; with this model, in order to control how subjects acquire resource access, there are two helper notions: subject identifiers, and permissions. Permissions depend on subject and resource and define whether a standard operation (*read*/*write*/*execute*) is allowed or not for that particular subject and resource combination. Enumerating the allowed (or conversely, disallowed) operations on all possible subject-resource pairs results in a matrix also known as *access control matrix*. Two ways to implement the access control matrix are access control lists (ACLs) and capabilities [204]. Based on how the assignment between permissions and subjects is made, there are several flavours of access control policies:

Mandatory Access Control (MAC). MAC policies allow only for one single entity – the administrator – to define the permissions of subjects onto resources [231]. As opposed to DAC, this control is centrally maintained by system administrators, and cannot be changed by individual users. This access control model has been implemented in SELinux and Windows Vista.

Discretionary Access Control (DAC). DAC policies allow users to define permissions that other subjects can have onto system resources [231]. This access control model has been implemented in traditional UNIX systems. As opposed to non-discretionary access controls (e.g., MAC), this scheme can be modified or overwritten by all users with an access permission allowing them to modify their own subject permissions over resources.

Role-Based Access Control (RBAC). In systems with lots of users, updating the access permissions becomes cumbersome when users need their permissions changed, users appear or disappear, or resources change. RBAC was proposed as a combination of MAC and DAC whereby the notion of *organisational role* comes as an abstraction between users, and permissions over resources [66]; the roles are already-made associations of subjects and permissions, hence users of the computer system are no longer associated with their permissions directly (see Figure 2.4). In this way, adding or removing users and deleting existing ones is much easier than before, since the roles in an organisation tend to change much less frequently than the users and resources. The NIST model proposed for RBAC in 2000 by Sandhu, Ferraiolo and Kuhn refers to four main variations of RBAC: **flat**, **hierarchical**, **constrained** and **symmetric RBAC** [203]. Flat RBAC refers to many-to-many user to role and permission to role assignments. Hierarchical RBAC adds to flat RBAC partial ordering between roles. Constrained RBAC considers *separation of duties* as a technique by which there can be constraints on what

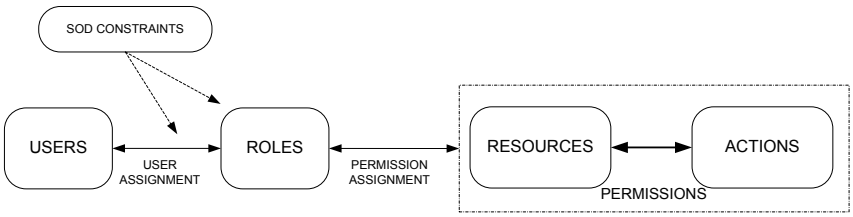


Figure 2.4: Diagram of the Role-Based Access Control model [203].

roles are mutually exclusive when a certain action is performed. Last, symmetric RBAC adds the permission-role review for distributed applications in which it is needed to verify and possibly change the permission assignments.

Attribute-Based Access Control (ABAC). The ABAC model makes the access decision based on attributes, that is, features of the subject and user ("a silver customer", "located in the US"), resource being accessed "located in Japan") and sometimes the context ("now it's 3 am") [213]. This model was suggested in the context of authorisation in distributed systems [138] and then in the context of Web services [213]. Li et al. emphasised the existence of distributed information used by RBAC schemes (roles, delegations of permissions, linked roles, parameterised roles). Since this data is scattered across different entities in a network, the ABAC model proposed *attribute authorities* to manage this data; management includes extracting the useful data from its signed containers, making inferences about attributes and authorities, and delegating the attribute authority to trusted entities.

2.4.2 Usage Control

More recently, the notion of access control has been extended into that of *usage control*. Park and Sandhu [183, 184] argued that the access control model is unable to cover what happens after the authorization of a user succeeds; that rights should be consumable per requested action and using some rights to perform an action could have an impact over other rights of a user. To that matter, *usage control* models the notion of usage of a resource, before, during and after the access request. As shown in Figure 2.5, the subject is being authorised to use a resource depending on several relations between the user rights, authorization rules, conditions and obligations. These relations are not static but they may evolve over time. For instance a usage policy would be:

> Due to health hazards, no radiologist is allowed to use the X-ray scanner for more than 10 hours per week.

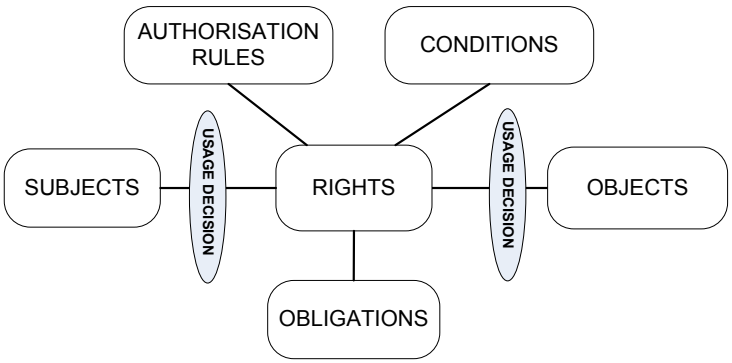


Figure 2.5: The elements in the UCON model [183].

This policy is more complex than allowing or disallowing a subject (the radiologist role) from performing an action (using, i.e. performing an X-ray scan) over a resource (the X-ray scanner); this policy considers the action of performing a scan as continuous, and as increasing the total scan duration. Even if the access was authorised when the total duration was already 9 hours and 59 minutes, as soon as another minute passes, the decision would be revoked because of the total scan time condition. This example illustrates one main novelty of the usage control model: *decision continuity*. Decision continuity means that the authorization decision can be taken before, during or after a resource access, and can change depending on user actions (or their lack), as well as environment conditions. Other aspects that usage control considers anew are *attribute mutability* and *obligations*. The former refers to the properties of attributes about the subject, object or environment that can change depending on the user actions (apart from affecting the authorization decision in the first place). The latter refers to conditions on the future actions of the user. Obligations are defined as "predicates that verify mandatory requirements a subject has to perform before or during a usage exercise" [184]. The same paper introduces the UCON$_{ABC}$ (Authorizations, oBligations and Conditions) model family, useful in Digital Rights Management and distributed access control; this family of core models includes:

- pre-authorization models (the existing access control models), whereby the authorization decision is made before the action is allowed;

- ongoing-authorization models, whereby authorization decisions are made continuously, while the action onto a resource is taking place;

- pre-obligation models, whereby pre-obligations are introduced (actions or conditions that have to be fulfilled at request time, before the authorization happens;

- ongoing-obligations models, whereby there are obligations to be satisfied as the usage of the resource is taking place;
- pre-conditions models, where certain environment conditions have to be fulfilled before the resource usage;
- ongoing-conditions models, where environmental conditions needs to be satisfied while the resource usage is taking place.

Ensuring that usage control policies are satisfied is more difficult that for classic access control policies. While access control policies only require a one time decision on (dis)allowing the access to a resource, usage control enforcement involves monitoring of the usage of a resource, and updating authorization decisions at runtime.

Relation to this thesis The enforcement mechanisms described in this thesis enact access control (mostly ABAC and RBAC) and usage control policies in an SOA environment. While usage control is a more precise authorization model for complex distributed scenarios, it is yet new; the traditional access control remains the established target model when translating business regulations (usually stated in English) into an implementable set of requirements.

2.5 Security Policy Enforcement

Enforcing a security policy means how to make sure that a system (a machine, an application or a distributed application) adheres to that security policy. Admitting that a security policy is one or more statements whose satisfaction improves the system's protection, then enforcing a security policy is synonymous with securing the system as the policy requests. Since this thesis is concerned with policy enforcement mechanisms, in what follows we describe the concept of *reference monitor*, as an abstraction inspired in security (research and practice) from operating systems. We then briefly sketch some classic enforcement mechanisms used in practice, some mechanisms used in theory, as well as some machine-readable languages to express and enforce security policies.

2.5.1 The Reference Monitor

The *reference monitor* models a guardian of a resource: its purpose is to prevent users that are not authorised from accessing a system resource. The reference monitor was introduced in 1972 in the report of Anderson [6], whereby an entity (whose abstract model is the reference monitor) intercepts all attempts of users to use a resource,

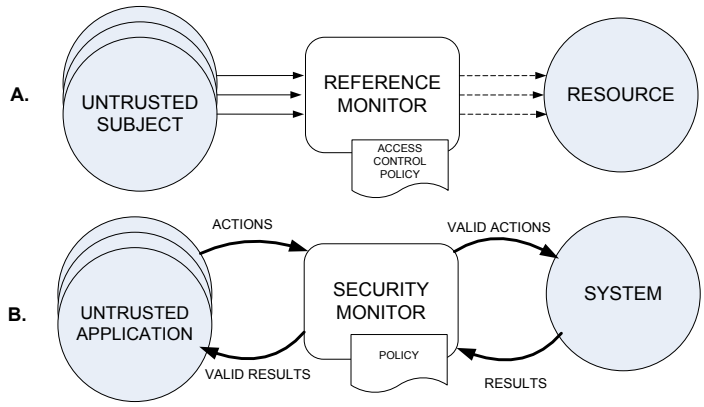


Figure 2.6: The reference monitor (A) and a security monitor proposed in [143] (B).

and after determining if the attempt is legitimate or not, it would allow it to happen (otherwise it would block it). A reference monitor is shown in Figure 2.6 (A). As mentioned in the original report, there are three main properties that an entity that implements the reference monitor concept, should satisfy:

- C1. **complete mediation** refers to the the ability of the monitor to intercept *all* the attempts to use the resource it is guarding; sometimes complete mediation is referred to as *non-bypassability*, in the sense that the monitor cannot be bypassed when accessing the resource;
- C2. **tamperproofness** means that the monitor cannot be modified, or altered, by external entities;
- C3. **small enough to be verifiable** means that the monitor can be tested for correctness; this is to check that the monitor is free of bugs or flaws.

These properties have been considered to be necessary and sufficient to enforce any set of access control policies on an object [107]. The concept of the reference monitor has been associated with that of the *security kernel* – a concrete implementation of the reference monitor – and *Trusted Computer Base* (TCB), as a set of protection mechanisms in a computer system. The TCB delimits a trusted security area outside of which the system is untrusted.

2.5.2 Execution Monitors

In security enforcement as seen from a formal methods perspective, a security policy is defined as a predicate over a set of program executions. Alpern and Schneider [4] have separated policies from properties; a *property* is a policy so that every individual

program execution satisfies a certain predicate related to the policy. Hence, whereas a policy is defined as a relation over a set of executions, a property is defined for an individual execution. The only policy that is not a property from the enumeration in Section 2.4 is information flow. As per [22, 142], there are several types of properties:

- **safety** properties state that “bad executions cannot be remedied" in the program, or if a disallowed action happened, the program cannot continue since the situation cannot be amended. Access control is a safety property;
- **liveness** properties specify that “finite executions can be remedied", or no disallowed action can happen in any finite amount of time. Availability is a liveness property;
- **renewal** properties stipulate that “infinite executions are good if and only if they are good infinitely often" [142]; renewal properties include all safety properties, and some liveness properties;
- **execution monitoring** (EM) properties are those that can be enforced by an *execution monitor*. A EM monitor enforces a property by interposing between the target and the trusted system, and allowing only safe actions to execute [142]. Despite Schneider’s claim that monitors can only enforce safety properties, Ligatti et al. show that some types of monitors can enforce non-safety properties [141] .

Investigating execution monitoring reveals a more formal notion of property enforcement than the one given at the beginning of this section. Among other authors, Hamlen et al. [95] recognize that monitors have two important properties: (1) **soundness** and (2) **transparency**. The former means that the output of the mechanism are always compliant with the property to be enforced, while the latter, that the mechanism does not alter the semantics of the target. Hence, we have the following definition of a runtime enforcement mechanism:

> *a monitor* (in [141] as an automaton) *enforces a policy onto a target, if it is sound and transparent*.

In what follows, we will give three examples of execution monitors: security automata, edit automata, and mandatory-results automata.

Security automata were introduced by Schneider [4, 207]; a security automaton is a conceptual execution monitoring mechanism that analyses security events of a target program, and terminates it if it recognizes a malicious action. More formally, a security automaton S=(Q, q_0, δ) is a finite or countably infinite state machine which maps onto the execution of the target application. Both target and the automaton

have a current state q_0; if the security policy allows the transition to the next state, based on the presence of an input symbol determined from the transition function δ, then the automaton allows it too. Otherwise, it terminates the target. The set of input symbols is given by the security policy, which accepts only a limited number of operations onto the target. The expressive power of security automata is given by the diverse ways in which the transition function δ can be described. Security automata were shown to be able to enforce access control policies, but not information flow nor availability or bounded availability policies [54].

The enforcement capabilities of security automata evolved from halting the execution of the target (security automata) to operation suppressing or injecting in order to remedy the “bad" execution. Alleviating the halting behaviour – rather draconic, as it was – of the original security automata are the *edit automata* [140] that were introduced later by Ligatti, Bauer and Walker. These new automata can truncate, insert and modify, if needed, the execution of the target. Introducing edit automata has led the way to more powerful monitors: edit automata can ‘pretend’ they allow the target to execute until the monitor eventually accepts it as legal (in other words, a sequence of untrusted actions is suppressed and only if it is proved not to violate any policy, it is reinserted in the program flow).

More recent work has uncovered the difficulty of implementing edit automata in the real world [143]. Ligatti and Reddy argue that monitors based on edit automata are supposed to predetermine all results of any actions without executing them (so that they could safely suppress any malicious action). This is impossible since such actions may be intractable, but also since the automata cannot buffer indefinitely many predicted actions, before producing new ones. The authors then propose Mandatory Results Automata (MRA); these monitors are obliged to return a result to the target application before analysing the next action to be executed by the target [143]. The authors also argue that MRAs no longer require the notion of transparent enforcement, since the monitor is free to change the actions and results as it sees fit. Such monitors are shown in Figure 2.6 (B).

Relation to this thesis As already mentioned, this thesis concentrates on security policy enforcement at runtime. One of the novelties in this work is that we extend the classic single-machine-based execution monitoring, to a distributed scenario. Since in practice bad executions do occur, we also tackle ways to remedy such bad executions. This work does not approach non-safety properties and execution monitoring from a formal perspective, but rather from a practical view; we aim to provide users and administrators with concrete ways to be more in control of their systems.

2.6 The XACML Language and Standard

To express security policies, in practice there is one standard policy language – eXtensible Access Control Markup Language (XACML) – proposed by OASIS and now in its third version [174]. XACML has been proposed as an XML-based syntax to express access control restrictions that map to ABAC and RBAC, but with which usage control policies can also be formulated.

XACML policies are organised in sets, and *policy sets* come with *policy-combining algorithms* that show how to combine the decisions and obligations from multiple policies in the same set. In its turn, a policy has a number of *rules* with an identifier for the rule-combining algorithms, one or more obligations, and a *target*. The rule is a target, when a condition is true or false, and an *effect*. An effect can only be a *Permit* or a *Deny* for that combination of requester, action and resource. A policy can have several rules. Conditions, as part of a rule, are statements that involve attributes and can be evaluated to *true*, $false$, or *indeterminate*. The target refers to the set of decision requests (tuples resource-action-subject) to be evaluated by a rule, policy, or policy set; the target is relevant to evaluate if an incoming request is relevant for a policy or not. A policy can only have one target. The policy decision depends in the conditions, in that if the condition returns *indeterminate*, then the rule will also return that value when evaluated. If the condition returns $false$, the rule will return $NotApplicable$; lastly, if the condition returns *true*, then the effect of the rule will be *Permit* or *Deny*. There are several XACML evaluation engines, of which the most notable are Sun's XACML Implementation [222], and Google XACML.

XACML also comes with an enforcement architecture to enforce policies on distributed systems. This architecture is a standard enforcement blueprint, and makes reference to the following conceptual components:

Policy Enforcement Point (PEP). The PEP is the entity that applies the access control decision received from the PDP. Apart from allowing and disallowing the request to proceed, the PEP can also check or update *obligations*. Obligations, in XACML, are actions that should be performed when enforcing the authorization decision.

Policy Decision Point (PDP). The PDP is the entity that decides whether a request should be allowed or not. The PDP receives a decision request from the PEP and produces an authorization decision. It first evaluates what is the applicable policy for a decision request, and then makes the decision based on information from the system and from the request itself.

Policy Information Point (PIP). The PIP is the entity that stores attribute values. In

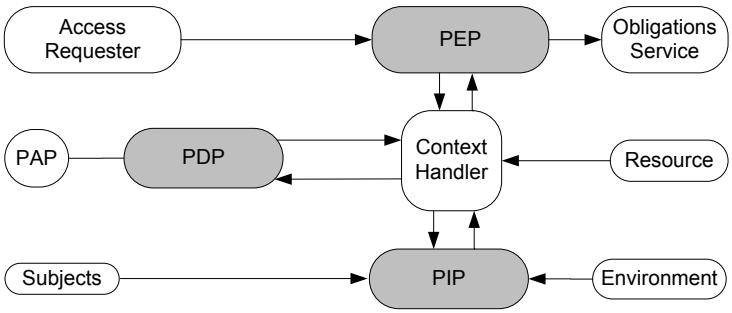


Figure 2.7: The XACML architecture diagram in the OASIS standard [174].

XACML, attributes can be group or role information or the subjects.

Policy Administration Point (PAP). The PAP is the entity that creates a policy or a policy set, and makes them available (e.g., deploys them) to the PDPs [174].

As shown in Figure 2.7, whenever an access request for a resource is intercepted by the PEP, a context handler would mediate the request, along with any extra information about the resource being requested, on its way to the PDP. The Context Handler gives to the PDP any information that is required about the subject, the environment, resource, etc. Once the PDP has reached a decision on whether to allows the access of the requester, to the resource, the context handler would forward it to the PEP that would enforce such decision, before or after updating any associated obligations.

Relation to this thesis. The work reported in this thesis is focused on security policy enforcement at program runtime. Consequently, we will examine several possible mechanisms of performing such enforcement in a distributed environment; we will also employ XACML both as language for expressing ABAC and RBAC policies but also as a standard architecture of policy enforcement. Even though the features of the security monitor as enumerated above are fully approachable from a formal methods perspective, we will make an informal argument for the formal provability of the features of our proposed mechanisms.

2.7 Policy Management Aspects: Policy Configuration

In the context of large networks that are expanding, deploying and maintaining policies becomes increasingly difficult. This difficulty is due to various reasons: there are many policies, there are many business domains, there are many administrators, the policies can change, etc. The system needs to be managed in such a way that if the system behaviour changes, or if the policies change, the operation of the system should not be disrupted. It is the task of the security or system administrator to

discover and address this problem, but the manual approach to such administration issues is cumbersome and impractical.

To automate the policy administration issue, there is the notion of *policy management*, or *policy-based management*, which refers to the management of a distributed system by means of policies. Defining policy management is not easy since to our knowledge there is no definite reference in this area. Nevertheless, we hereby describe three main viewpoints that converge:

IETF standards. In IETF's policy architecture [104], policy-based management resembles XACML policy enforcement because of the separation between enforcement and decision points. The difference is in the existence of interfaces to define and update policy rules (a policy rule is a policy subpart) and of a repository for policy storage and retrieval. IETF's architecture also addresses *policy conflicts* and *policy consistency*. The former refers to the situation when two or more policies apply in the same time but their actions cannot be executed at the same time; the latter refers to supplying system notifications when policies are in conflict.

Ponder community. The Ponder community suggested using the Ponder policy language for the specification, deployment and policy association between the devices to implement them [44]. This means that the same policy-specification language can cover syntactically and semantically not only the resources to be protected (e.g., organizational roles, LDAP entries), but also policy dissemination over resources, policy changes, and policy conflicts.

Other authors. In 1993, Maullo and Calo stated that policy management involves

> being able to raise the level of abstraction at which interactions can occur, so that people responsible for managing the system do not become overburdened or overwhelmed by excessive detail. This requirement is commonly described as the ability to deal with systems in terms of policies rather than explicit controls [148].

Maullo and Calo suggested an object-oriented mechanism to describe system elements like resources, access methods, and communication links. With such model, the administrators can create a model of the system, track data within the system, and check if the system representation complies with business goals.

From the viewpoints above, we observe that policy management is defined in two slightly different ways:

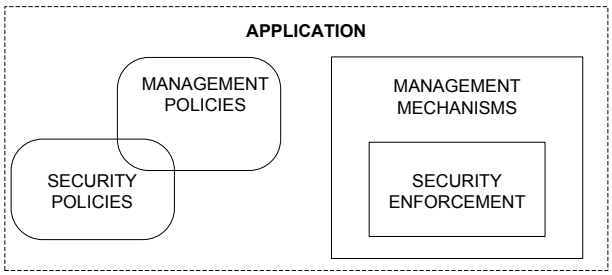


Figure 2.8: The problem of policy management as a superset of policy enforcement.

- the management of a system *by means of policies* (for a system that does not already require any policies). As mentioned before, we do not look into how to obtain these policies from organizational goals.
- the management of a system that *already* involves policies.

In this thesis we look more into the management of a system that already has security policies in place. Without the purpose to make a formal policy taxonomy, we make the observation that not all policies are security policies, so not all policies are enforced by security enforcement mechanisms. There are also other policies that are important for system management, that are not necessarily for security purposes. For instance, the requirement to communicate only to services from a particular service provider, or to use only a certain communication protocol with some service endpoints – these are system requirements that impact the application runtime, yet are not directly linked with system protection aspects. We can call them *management policies* – those requirements on the system runtime that cover resource allocation, resource lifecycle, communication links and other aspects like Quality of Service. However, there are also some policies at the intersection of security and management policies, e.g., logging can be done for security purposes. We therefore say that policy and management policies are not disjoint; this situation is shown in Figure 2.8. Since other authors see enforcement as a subset of management, we can consider security enforcement mechanisms as a subset of management mechanisms.

Policy configuration. The most approached policy management subproblems are the problem of building a vast policy model to account for both security and management aspects, and policy conflict management. Yet to our knowledge less effort has been invested in *policy configuration* on the setting got a distributed system. To us, *policy configuration* refers to how to fit the policy specification onto a distributed system that can enforce it. For instance, it is a valid and yet unexplored problem how to split the policy enforcement tasks among the components with related or overlap-

ping capabilities: what security controls (or components) should perform attribute retrieval, how the security data should be propagated, or what happens when a security component is no longer available or changes.

In this thesis, we maintain that the aspects above do concern, albeit indirectly, the application runtime. From what we consider *policy configuration*, we will examine in particular security data dissemination. To that respect, we will consider the two unofficial variations of the XACML model based on how to retrieve the data that is useful for decision making on the PDP side:

- the *push model*, where the PEP collects the PIP data and sends it to the PDP;
- the *pull model*, when the PDP collects all relevant data in order to make a decision.

The pull model has a minimal load on the client and PEP and is designed for cases when the authorisation process is entirely performed on the PDP; the push model, conversely, allows for more flexibility to the client and the PEP needs to ensure all required data reaches the PDP. In practice, a *hybrid model* is used, whereby some data is pushed and some is pulled.

Apart from these two informal schemes, the XACML standard does not make any recommendations about how the rules of a policy should be treated on each of the policy components detailed above. While it does specify the interaction between the PDP and the PEP, the standard does not specify how the PEP, PDP, PIP and PAP interact among each other to trade security data.

Relation to this thesis This thesis proposes to address the policy configuration problem by examining the different ways in which security policy data should be treated. Also, the work reported in this thesis suggests a configuration tool that can modify the connections between the system components in charge of policy components, so as to preserve correct and consistent enforcement when the application spans over several domains and administrations.

2.8 Security Evaluation: Security Metrics

Security developers and administrators have always needed to verify that their security mechanisms work properly. Security evaluations have emerged relatively late and they are related to *assurance*. Defining assurance as “the likelihood that a system would fail in a particular way", Anderson sees security evaluation as “the process of assembling evidence that a system meets, or fails to meet, a prescribed assurance target" [8, p.858,869]. In his view, the need for security evaluations appeared when the

entity that needs the security mechanism and the entity that implements the mechanism, are different. Nevertheless, we believe that the need for assurance exists even when the same entity implements and uses a security mechanism.

In practice there are several ways to perform security evaluations. We describe these different evaluation options, starting from Anderson's classification in [8, Chapter 26] and adding one more element to it:

- *security testing* is usually performed by the same entity that implements the security mechanism, because it has access to the code. Testing means reviewing the code and running tests against it to check that it complies with the requirements of a target environment. Tests depend on the target environment, test suites are not standardised, and they often fail to predict what can happen in practice.

- *Evaluation Criteria* such as Trusted Computer System Evaluation Criteria (also known as "The Orange book" [48], and the Common Criteria for Information Technology Security Evaluation [218]. These are evaluations that the issuer of the software would undergo to certify its application or mechanisms. The Orange Book supported several evaluation criteria ranging from discretionary access control, to mandatory access control and security domains; however, due to economic reasons, the audits based on the Orange Book criteria weren't very successful. The Common Criteria were a later set of more flexible evaluation checklists, based on *protection profiles* and on an evaluation assurance level.

Anderson gives a list of the shortcomings of Common Criteria: that they do not insist on having realistic protected profiles, they are very limited in evaluating products, and focus on the design rather than on aspects such as administration of security controls. A third element that we are adding to the list is *security metrics*. Security metrics can evaluate security assurance not by design but at runtime. They are detailed below.

Security metrics. Defining security metrics is still a subject of debate, because the security properties that are target of assessment are not always either qualitative or quantitative (e.g., code complexity can be measured in lines of code, together with the number of control paths; but the criticality of a software vulnerability is more complicated to estimate). Jaquith sees a security metric as a consistent standard of measurement [114] of the security of an IT system, that would answer questions such as how effective are the security controls of that application. Jansen argues that security metrics relate to using a method to measure an assessable security property of a system, and for an organisation the use of such assessment should give hints of where extra efforts would go in terms of security processes and control. Jaquith's view of

what he calls “a good security metric" refers to the following characteristics:

- a security metric should be reproducible, which refers to its objectivity, or the fact that measurement done by different subjects over the same should render the same quantification;
- the data to compute a security metrics should be cheap to gather; this involves the notion of measurement frequency, in line with the application changes;
- a security metric should be expressed in one number rather than *high* or *low*;
- a security metric should have a unit of measurement to allow for comparisons;
- a security metric should be context specific in that it integrates in the semantics of the application whose security it evaluates, and provides hints of what can be done to improve the security controls as such.

Jansen separates between leading and lagging indicators, i.e. security conditions before and after a security-relevant event; a latency period for a lagging indicator must be as short as possible, since that indicates a timely response of the organisation to potential security problems; still, the author argues that the current state of this research field deals with lagging indicators that can be obtained as of now by a subjective evaluation, and that are not properly contextualised to the security objectives of an organisation [112]. Some examples of security metrics are:

The software attack surface. The attack surface, coined by Michael Howard in 2003 [100], has been defined as the subset of system resources that could be used in an attack against the system [147]. Such resources can be software interfaces, communication channels, or data. Even though this metric is popular in the security community, it has not been empirically proved that there is a correlation between reducing the attack surface of a program, and its state of being more secure.

Number of attacks. The number of attacks over a system or network is an indication of the efficiency of the security measures in place [114].

Vulnerability counts. The number of software vulnerabilities (i.e. weaknesses that can allow an attacker to thwart the system’s security controls) can be classified by criticality, type, unit of time, or asset class that they affect. They indicate the level of security of the system [114].

In NIST’s view, some techniques to measure various security metrics are: code analysis; cooperative bug isolation as a technique that uses software instrumentation and random sampling from already deployed software; black box security testing.

Relation to this thesis. Even though there are numerous examples of metrics as well as their desired features, the literature does not yet show applications of these measurements in specialised contexts, nor demonstrate their usefulness in each case. This thesis addresses this problem by suggesting a particular kind of security metric that we call a *security indicator*. Security indicators can monitor the overall security levels of an application in a particular context, and trigger reactions when certain thresholds are reached. Thus, in our view, security indicators are not only informative – they not only supply an estimation of what they measure, to the administrators and managers – but also trigger appropriate system behaviour.

2.9 Summary

This chapter has described the most important technology and concepts used in this thesis. First, we described Service Oriented Architectures and connected technologies such as ESB and Business Process Management, as a yet flourishing environment for enterprise applications. We then discussed security regulations governing large-scale applications whose components can be reused, that need to ensure the safety of their operations and data. Such regulations are refined into security policies that include access and usage control policies, and such policies need to be enforced on the applications subject to regulations and security contracts. Policy enforcement can be realised by special security entities that also act at the program runtime; such entities are called execution monitors. The services of such application enforcers can in their turn be exchanged across applications, hence the same security tool can be used with different policy sets and onto different systems with different purposes. As such, each execution monitor should be adapted, or configured, to the scenario and concrete application context where it will operate. Moreover, each organisation needs to prove that the rules and regulations that govern their domain of the SOA world are indeed respected; security evaluation can be done based on common evaluation criteria and security metrics.

Chapter 3

Case Study

As every man goes through life he fills in a number of forms for the record, each containing a number of questions. [..] There are thus hundreds of little threads radiating from every man, millions of threads in all. [..] Each man, permanently aware of his own invisible threads, naturally develops a respect for the people who manipulate the threads.

Alexander Solzhenitsyn, *Cancer Ward*, 1968

3.1 Introduction

Healthcare institutions across different countries, either in the private sector (e.g., hospitals, assistance providers, clinics, pharmacies and pharmaceutical companies), or in the public one (e.g. health boards, healthcare authorities and health ministries), are all making efforts to cooperate to provide better healthcare services. A *public-private partnership* (PPP) is one of the most successful collaboration ideas to boost healthcare education and healthcare provision, with a wealth of initiatives instituted around the world, for instance in California [133], in wider US [234], in Italy [34], and in China [224]. The cooperation among these entities has become mostly electronic and covers the use of shared data and services. A few examples are:

- doctors and nurses have their own PCs or PDAs that they use to enter or consult data about patients;
- *telemedicine* services e.g., tele-radiology, tele-psychiatry, tele-cardiology, are being provided to patients in other countries than the doctors;
- doctors can remotely prescribe drugs to some patients, send them an electronic prescription, and monitor the drug therapy remotely (tele-pharmacy).

Electronic health systems manipulate *electronic health records*. These records are electronic patient health records that can help clinicians make decisions [155], and contain private data about patients, including personal identification data, current health state, medications, and medical history. Standardizing the format of this data, along with standardizing the ways such data can be exchanged across medical systems, is an essential enabler of healthcare collaborations.

However, sharing such data across different channels gives rise to a multitude of threats. These threats concern data confidentiality (when patient data reaches unwanted parties), data integrity (when undesired modifications to the data can lead to wrong medical treatments), or accountability (when medical actions cannot be accounted for). Medical data privacy is perhaps the most popular research area, compared to the other threats. A couple of examples are given by Anderson [7], who reports how private investigators can bribe hospital tellers to supply classified information about patients, in exchange of money; how stealing a computer can lead to blackmailing female public figures by threatening to disclose abortion data; or how linkable data such as age, gender and medical condition of children can be used to identify a parent whose name is not to be directly disclosed. Moreover, the American Society for Testing and Materials has a committee for healthcare informatics that identified other threats to healthcare data [5], of which there are:

- modification of information content, destruction of messages or auxiliary data;
- message sequencing threats such as replay and delay of messages;
- unauthorised disclosure of data, or information derived from information flows;
- repudiation, i.e. when it is denied that an action has been performed;
- denial of service, that makes a system unresponsive and unable to perform legitimate actions;

Hence, it comes to no surprise that healthcare is a very regulated area. The regulations in this field are strict and cover not only the protection of patient data, but also safety and security aspects of IT systems that operate with such data. For instance, the European Directive 95/46 [62] forbids the exchange of patient data without the patient's consent. In the US, the Health Insurance Portability and Accountability Act (HIPAA) [233] specifies sets of policies and procedures to limit access to computer systems that contain sensitive patient data, and to prevent interception or deletion of such data by unauthorised parties. Also, Health Level Seven (HL7) [97] defined a series of standards for the electronic exchange of medical data, of which there is emphasis on secure bidirectional communication flows, that also ensure non-repudiation.

Determining if these regulations and procedures are properly met is the purpose of *audit*. Audit can be of several types: organisational audit, clinical audit, internal audit. Organizational audit is an external audit that, based on a set of standards, examines how well an organisation functions. Clinical audit is a multi-disciplinary activity that checks how well clinical standards are respected in certain medical field, and how organisations can compare their performance with others. In the UK, for example, the largest provider of clinical audits is the National Health System [164].

This thesis deals with how to ensure that software systems adhere to security policies that span several systems. To that respect, we have chosen to analyse healthcare systems as they are impacted by healthcare regulations. Healthcare makes a relevant case study because of two main reasons: it is highly distributed and highly regulated. As described above, healthcare systems span different business domains and geographical boundaries, and are dynamic in that changes in the context and in the provided services are likely to occur frequently. Moreover, the numerous healthcare regulations are the generators of security policies, and software audits are also verifying the compliance to those policies. Studying how to ensure that security policies are enforced in a distributed application is useful both for the stakeholders (hospitals, clinics, etc) so that they can react to security issues in time, but also to auditors in their audit evaluation.

3.2 The Electronic Healthcare System

We base our analysis on a scenario inspired from two hospitals in Italy: one in Trentino, referenced in [12], and another one in Lombardia, called Hospital San Raffaele Milano (HSR). The actors and interactions presented in our case study are based on those of these two hospitals. In order not to limit the generality of our work, in our study we considered that the activity of the hospital in our example is impacted by both Italian and non-Italian regulations.

In what follows we will first describe the actors in our scenario and their goals in providing or receiving health care. Then, based on the regulations that control security and privacy aspects in this area of operation, we formulate the requirements and challenges that the electronic system needs to address in order to enforce such regulations.

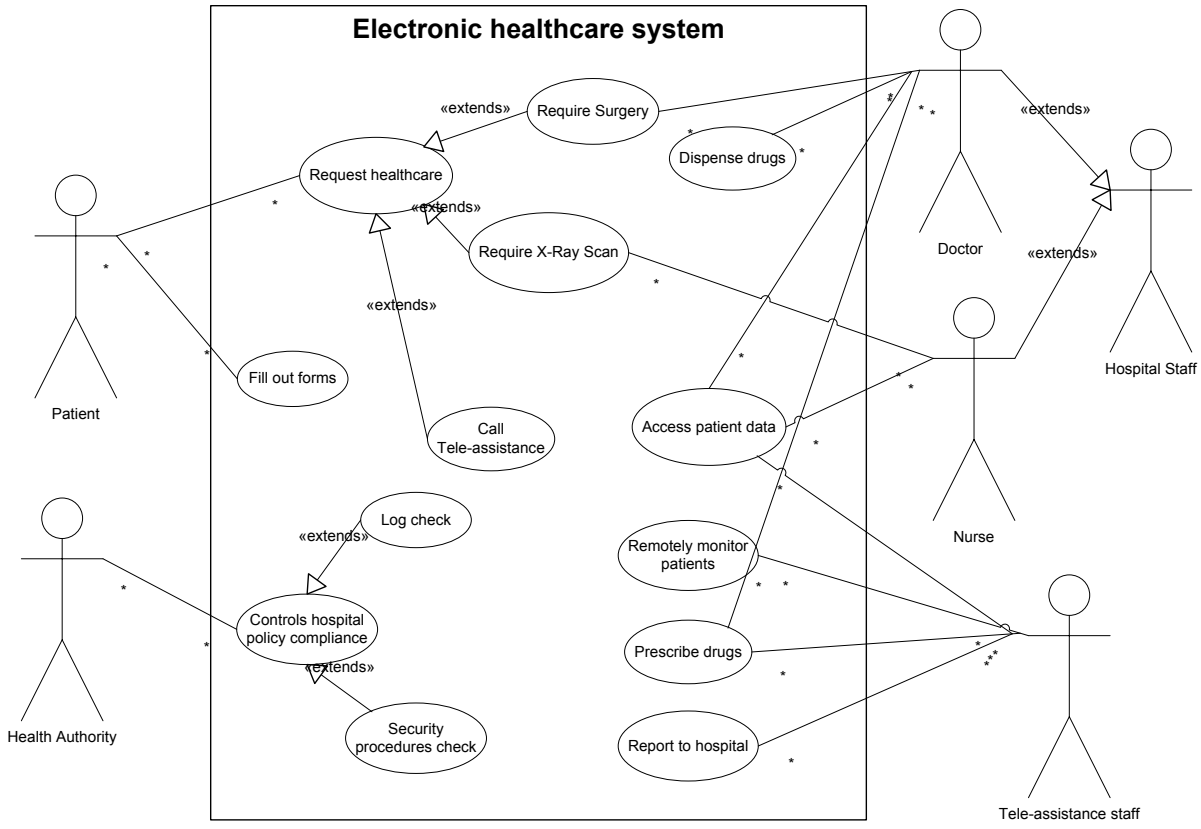


Figure 3.1: UML use case diagram for the electronic healthcare system.

3.2.1 Actors and Goals

Figure 3.1 shows a scenario in which patients, health authorities, and medical staff from two collaborating institutions (a hospital and a medical tele-assistance company) use an electronic system to manage the usage of healthcare services. The use cases and actors are as follows:

Patients. Patients request healthcare services provided by the hospital, by contacting the hospital or medical staff able to assist them remotely. These healthcare services include diagnosis, scans (e.g., X-ray scans), surgeries. In some cases, patients volunteer to research programs of the hospital, whereby they undergo drug treatment at home, and have to periodically report their health state. Similarly, patients can be outpatients (i.e., they are hospital patients but instead of being admitted to the hospital during treatment, they receive the treatment at home) and in that case they require either medical monitoring, or they call medical staff in case they have medical problems. In all these cases, patients have to consent to medical treatment and to supply personal data. As of now, they are required to fill out forms (e.g., surgery consents, data treatment consents).

Hospital staff. Hospital staff comprises doctors and nurses to perform hospital duties. These duties are of two kinds: treating patients, and preparing the medical equipment or drugs required for treatments. Doctors' duties involve decisions about treatment and medication, while nurses' duties are more on administering drugs, helping with the treatments, and monitoring patients. The doctors and the nurses can sometimes have overlapping responsibilities (e.g., monitor the state of patients), and it can be that in emergencies, nurses can take over duties that normally belong to doctors. To properly provide patient care, the hospital requires patient data to be processed from the patient forms; the hospital staff will have to access such data on a need to know basis.

For preparing drugs or obtaining medical equipment, hospital staff have to follow standard procedures that involve security aspects such as *separation of duties*. Separation of duties is a rule that helps eliminate frauds that arise when the same employee can exercise more of its privileges over connected tasks. In the hospital, for instance, the person that can prescribe drugs to patients should be different from the person who dispenses the drug. In absence of this rule, there is the risk that the employee tends to prescribe the same medicines that they can dispense, even when not required or appropriate.

Tele-assistance staff. Tele-assistance staff are employed by a tele-assistance company in partnership with the hospital. This company provides medical care remotely

for the hospital's outpatients. These patients can either call an assistance number in case of emergency, or be monitored and consulted about their treatment and state. For these purposes, the hospital needs to provide access to medical data to the tele-assistance staff, so that the latter can take correct medical decisions (such as prescribe drugs or make treatment modifications). Nevertheless, all changes or additions need to be accounted for to the hospital. The remote assistance thus provided uses electronic services and data that either belong to the company, or to the hospital. What medical services can be used in common, the usage terms of such services and of patient data, and company responsibilities, are all described in a contract between the hospital and the tele-assistance company.

Health authorities. Local, regional or national health authorities are interested to check that all medical institutions are complying to the healthcare regulations of that particular area or country. Compliance to healthcare regulations influences funding and investments. Consequently, from the point of view of this thesis, health authorities check that health units implement electronic systems that comply with existing security and privacy regulations. In particular, security procedures and system logs are checked for security violations e.g., unauthorised disclosures of information, unauthorised or non-compliant usage of medical equipment, etc.

The operation and the interactions among the actors in this scenario (patients, hospital, tele-assistance company and authorities) are influenced by legal requirements on security and privacy. Healthcare regulations include the following: HIPAA [233], HL7 [97]. Non-healthcare specific yet applicable security and privacy regulations include the new Italian Data Protection Code [16].

3.2.2 HIPAA Requirements

HIPAA [233] has been issued in the US to address frauds and offences in healthcare. Title II of HIPAA (called "Preventing health care fraud and abuse; administrative simplification; medical liability reform") suggests a standard for healthcare transactions when performed electronically, so that they would be safe for employers and providers. Hereafter, we enumerate several HIPAA requirements that the US Department of Health and Human Services considered important for the standardization of secure medical procedures.

HIPAA-1. First, HIPAA refers to the use and disclosure of any health or payment record, either electronic or not, that can be associated with a particular patient. Such data can be disclosed if the patient has authorised such disclosure, but the

information disclosed has to be kept to a minimum (a requirement of confidentiality). Also, whenever their medical data is used, the owners need be notified.

HIPAA-2. Second, HIPAA defines three kinds of controls for electronic data: administrative, physical and technical [233, Section 1173(d)(2)]. While the administrative controls cover policies and procedures for demonstrating compliance, the physical controls protect against undesired access to data on hardware supports. More interesting, technical controls concern electronic communication of patient data and access control of entities over such communication or data:

- **HIPAA-2.1** To ensure medical information integrity, confidentiality and availability in its electronic form;
- **HIPAA-2.2** Protection against unauthorised access, or information disclosures;
- **HIPAA-2.3** Protection against intrusion or hazards to the security and integrity of the data;
- **HIPAA-2.4** Documentation of activities so that compliance can be verified.

HIPAA-3. Medical entities can use *security measures* (i.e. information system controls) to implement the standard, but these measures need to be proportional with the capabilities of the system, its cost and security needs.

HIPAA-4. Security measures have to be periodically reviewed and modified to that they comply with the standard.

HIPAA took effect in the US in March 16, 2006; violations to HIPAA rules are fined.

3.2.3 HL7 Requirements

HL7 [97] is an international authority for standards of interoperability for healthcare technologies. HL7 addressed electronic health record formats and transactions in healthcare-related computer environments. An interoperability specification for electronic data exchange, in HL7's view, defines a message as a basic unit of information between disparate entities that need to exchange data.

For increased security, HL7 has suggested definitions for standard permissions in healthcare and adopted an ANSI RBAC-based set of models. As such, the HL7 RBAC Role engineering process [99] describes a way to derive healthcare staff permissions and associate them with roles and work profiles for roles. Section 5.2.3 of the same document mentions the existence of *constraints*, as "local rules that affect roles, rather than permissions". Some constraints as per [99, Section 5.2.3] are:

- **HL7-1.** *Separation of duties*: "A laboratory user can co-sign another Lab Technician's results, but cannot co-sign their own even if logged on as the Lab Technician Supervisor";
- **HL7-2.** *Time-dependency*: only one authenticated nurse can take on the permissions of a Head Nurse, for a shift of 12 hours on a hospital floor;
- **HL7-3.** *Mutual exclusivity*: a physician can either perform his or her scheduled clinic hours, that are being counted, or work in the emergency room for a continuous amount of time;
- **HL7-4.** *Cardinality*: "Only one physician may have access to the Chief of Staff permissions";
- **HL7-5.** *Location* refers to different permissions depending on location, e.g., a staff member of one hospital tries to access the same hospital remotely, or another hospital that is not the primary location.

Apart from this RBAC approach, HL7 specifies some requirements for services that secure healthcare transactions [89]: identification and authentication of individual users, authorisation, access control, integrity of data and systems, confidentiality, accountability, non-repudiation. Some other needs of HL7 cover the security of message routers or gateways, since the router might be unsafe for the messages transmitted.

HL7 does not define ways to enforce access control, privacy and accountability.

3.2.4 Other Requirements

While HIPAA applies to American healthcare institutions, there are also regulations in Europe that approach similar aspects as HIPAA. An example is the Italian Data Protection Code (IDPC) [16]. It has been adopted since 2004, and concentrates on electronic data protection in Italy. One of its main features is the principle of *data minimization* [16, Section 3], in that public data about individuals should be used as much as possible, while personal data, only when absolutely required. This data protection code also mentions the conditions of use of health data, that need be authorisation of the data owner and from a guaranteeing party (the 'Garante'). Title V of the Code, called "Data and system security", touches on *security measures*:

- **IDPC-1.** The Code specifies that any organisation or body to process the electronic personal data must adopt minimum security measures to protect personal data against lost, destruction, or unauthorised access [16, Sections 31 and 33];

- **IDPC-2.** The Code specifies some minimum security measures when processing electronic data: authentication processes and authentication credential management, authorisation, protection of systems and data against misuse, safekeeping and restore features, encryption or “identification codes" for sensitive data (health and sex life) [16, Section 34] . Another point is the requirement to update security specifications and security policy documents.

From the point of view of data retention, electronic data can only be stored for a six-month period by communication service providers; telephone traffic data should be kept for four years to assist in detecting and preventing crimes. The code also specifies inspection rights of the Garante party, such as its right to access information when investigating regulation compliance.

3.3 Challenges

The regulations above impact electronic healthcare systems, and hence the activities of all the actors in our scenario. Patients receive better healthcare services if security and privacy requirements are respected. Hospitals and other healthcare providing organisations need to be aware, understand and appropriately implement such security and privacy regulations. Healthcare authorities need to assess how correctly healthcare providers have integrated regulations into their electronic systems, and how they are being maintained.

Moreover, the assessment made by authorities (e.g., the NHS if the hospital operates in the UK) has to follow privacy and security regulations itself. Auditing a healthcare institution requires authorisations, collection and correct usage of patient data for audit purposes. In some cases like that of the NHS Information Centre in the UK [164], the assessment entities provide secure systems (infrastructure) for the audit: secure clinical audit databases, secure coordination of processes that perform clinical audit of official organisations.

The viewpoint we take in this thesis is that of the hospital. The hospital needs to secure its existing IT system based on security regulations that change over time. Also, it needs to gather way to evaluate its performance and how well it provides acceptable security guarantees. Table 3.1 separates the security areas that the hospital needs to investigate: the interactions with healthcare authorities, the interactions with third-parties (i.e., with the tele-assistance company), and internal security processes. The table shows the salient aspects that the hospital needs to comply with, as required by existing healthcare regulations, some of which have been presented above.

To accomplish these requirements, the hospital is faced with several challenges:

Category of security aspects	Security aspect	Regulation
Interaction with authorities	protection of patient data	HIPAA-1,IDPC1
Interaction with third-parties	disclosure of patient data	HIPAA-1,IDPC1,IDPC2
Internal security	authorisation system	HIPAA-2.1–2.3,HL7-1–5,IDPC1-2
	accountability and audit records	HIPAA-2.4
	security updates and management	HIPAA-4,IDPC2

Table 3.1: Regulated security aspects and how they occur in our scenario.

How to obtain security policies from security regulations? The security requirements in HL7, HIPAA, the Italian Data Protection Code or in EU directives, are all formulated in English and are ambiguous (e.g., what makes security measures 'minimal', when and how should a policy be updated, or when is medical data not available). The hospital needs to understand the concepts, adapt and split the legal requirements into implementable rules. Hereafter, we will call these implementable rules *policies*. *In this thesis we make the assumption that security and privacy policies have already been synthesised from security and privacy regulations*. We motivate this assumption on the ongoing efforts of HL7 to update or issue RFC-s (RFC 3881 on security audit and accountability, RFC 1767 on types for messages in healthcare), as well as of European projects like FP7-MASTER.

How to implement security mechanisms to enact satisfy security policies? None of the standards above explain how to comply to their requirements, but rather what these requirements would be on the general case. Therefore, the main difficulty that the hospital developers and administrators are faced with is what security controls to put in place to enforce such security policies. This issue is exacerbated when: 1) the hospital system is distributed along several locations, 2) security services are needed to control not only the hospital's own system, but also collaborating systems like that of the tele-assistance company. The tele-assistance services need to reach medical data in the hospital databases, and abide by the security policies in the contract with the hospital.

How to verify that security policies are enforced? The hospital's compliance to existing security and privacy regulations also implies the compliance of its partners (e.g., the tele-assistance company) to the same rules. Automating accountability processes is an important step towards complying with the Italian Data Protection Code or HIPAA-2.4. Knowing how compliant the system is in the present is essential for evaluating how the security policies can be better implemented.

3.4 Summary

Sensitive medical data can be easily misused or abused in IT systems. Patients can only consent to the use of their data in exchange for medical care (since they have no other form of control over how their data is being handled), and trust the healthcare providers not to misuse it. Health standards like HIPAA, HL7, the Italian Data Protection Code of 2003 have been proposed to bridge such gap. These regulations describe the need to have security measures in place to control unwanted disclosure of information, unauthorised access to the IT systems and data, and to keep records of system activity for accountability and audit purposes.

Healthcare institutions have to abide by the healthcare standards of their region or country. We have discussed an example, based on two real hospital cases, whereby a hospital that collaborates with a tele-assistance company is supplying care services to patients and is controlled by a healthcare authority. This system has to comply with HL7-1–5, IDPC-1 and IDPC-2, HIPAA-1–4. In this thesis we are considering that these requirements have already been translated into a form that can be implemented, that we call security policies. Security policies have to be implemented at the level of the hospital system, and, to a limited extent, at the level of the accessible components of partner systems. Since the official regulations and standards do not state how to achieve these tasks, the hospital system has to choose a way to enforce the above mentioned security policies in a flexible and robust manner. The rest of this thesis proposes a framework to enforce, assess and maintain the enforcement of security policies in a distributed service-based environment, in line with the legal requirements that now exist in the healthcare domain.

Chapter 4

Message-Level Enforcement of Distributed Policies

The king hath note of all that they intend,
By interception which they dream not of.
William Shakespeare, *Henry V*, Act 2, Scene 2

4.1 Introduction

A significant proportion of healthcare applications are nowadays SOA-based. Healthcare services, designed for very specific tasks from data processing, to medical procedures, can be packaged as Web services and made reusable across organisations. This approach makes sense in an environment where it is easier and cheaper to compose existing functionalities rather than to create them anew every time. Service-orientation in healthcare is supported by a wealth of examples, of which:

- Microsoft's reference architecture for information exchange among Web services for healthcare applications [197], covering Web service interactions, data formats, and roles of the communicating parties;
- a recent workshop organised by IEEE on Web services in healthcare [103];
- the Nationwide Health Information Network (NHIN), described as a super network of applications, a set of standards, services and policies across over 35 American healthcare organisations (as by 2010).

Healthcare applications, across organisations and countries, have growing demands of two types: interconnection possibilities, and security. Hospitals tend to buy third-party services such as radiological data processing; for their own management, hospital or clinic systems require integrating new reporting services, new devices, or new

medical data aggregation services. Also, hospital or clinic systems can outsource services, in their turn, to third-parties. For remote surgeries, connection parameters like bandwidth need to be guaranteed, without disrupting the rest of the system. Integration, quality of service, load balancing, and intelligent service provisions (varying functionality depending on runtime system conditions e.g., deviating traffic to different networks if a remote surgery is being performed) are required for such systems that are dynamic and span much more than just a local hospital network. The other type of demands are security needs. Recall from Chapter 3 that there are multiple healthcare regulations that healthcare institutions and software need to comply with.

In other sectors but in healthcare as well, the integration needs are supported by a state of the art technology called the Enterprise Service Bus (ESB). As described in Section 2.2.2, the ESB offers standard ways to integrate services and tune quality of service parameters. It deals with all protocol-level aspects, and acts as a message highway, making inter-service communication smooth and painless. As a technology to integrate services across networks, the ESB usually knows nothing about the logic of the services it connects and about their security demands. This is only to be expected, since the purpose of the ESB is to enable inter-communication rather than understand and protect it. In healthcare applications, the ESB has already been used to enable service integration [12].

Under the pressure to respect security and privacy regulations, each healthcare organisation implements security mechanisms, or controls, separately. This approach is economically reasonable and obvious in a world where data privacy is paramount – be it sensitive patient data or company data. Moreover, from a trust perspective, there are few companies who would agree to put security development into the hands of another. Hence the result is that we would find each regulation from the list in Chapter 3, implemented differently, and multiplied on the sites of different owners. Hence, for the services of each organisation, there would be a different authorisation subsystem that implements similar if not identical security constraints as other security subsystems of partner systems. This situation is depicted on the left in Figure 4.1.

Apart from the redundancy itself, implementing the enforcement of the same policy in different controls (that are later being used) is dangerous and tricky. If we allow that a system is as as secure as its weakest link, a cross-institution enterprise application is as secure as its most defective enforcement mechanism. From the perspective of global returns, this situation is unfair for those organisations that have invested more than others in security development. What is worse, it is not improbable that regulation statements (and hence, policies that stem from regulations) change. When the official text is amended, the updates need to be propagated at the same time, in

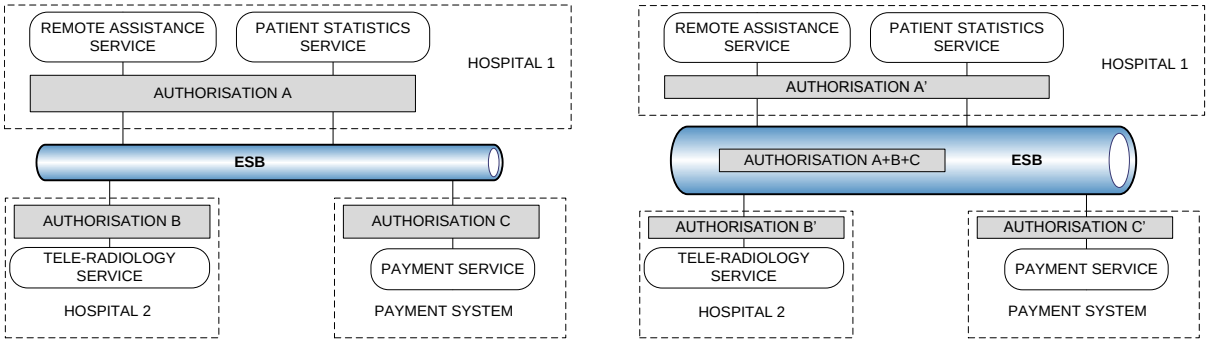


Figure 4.1: Within the same enterprise application, security subsystems that enforce the same policy independently (left); alternatively, the same policy can be enforced centrally (right).

the same way, to all controls that enforce that policy – and this is quite difficult to achieve. In the same line of argument, the more the disparate implementations of a requirement, the bigger the effort to manage all implementations and ensure that they interpret the rules in the same way.

Enforcing common policies in a central point solves these issues, simplifies policy management and increases security guarantees. This chapter describes xESB, our solution to enforcing security policies in a distributed SOA setting. This approach is shown on the right of Figure 4.1, where part of the authorisation controls previously known as A, B and C, have been replaced and deployed on the message bus to enforce a common security policy. Note that in this thesis we separate between *the messaging level* and *the application level* of a distributed application. The former – the“ESB level" or “infrastructure level" – is concerned with message flows, message routing and other protocol-specific aspects of services. xESB enforces message-level policies. The latter concerns aspects such as user permissions and role separation, workflows and business processes, etc. Also, hereafter we will mean Web services whenever using the word “services". This chapter is partly based on two previous publications [83, 82].

4.2 Requirements from the Case Study

In healthcare, malicious messages can be very damaging. Imagine what would happen if a Web service that needs to be used urgently for a remote surgery is flooded with requests in a Denial of Service attack so that it becomes unresponsive. Or if patient data would be sent in clear to the authorities databases during the weekly statistics report. Or, again, if hospital activity wouldn't be logged for a week because of a faulty logging service.

To protect against malicious message events as above, there are healthcare regula-

tions like HIPAA or IDPC, HL7, and the Privacy and Electronic Communications EC Directive [62]. These regulations concern various aspects on availability, authorisation, and accountability as described in the previous chapter. Deriving *security policies* from legal regulations benefits in that the malicious behaviour presented above can be identified as violating concrete rules stated in policies. As stated previously, in this thesis we do not consider how concrete policies can be derived from regulations, but rather how to enact concrete policies that stem from these regulations. Some examples of security policies that the hospital needs to comply with are:

Log all requests to tele-assistance and tele-dermatology services. This policy is motivated by the need to record system activity for accountability reasons, as required by regulations such as HIPAA-2.4 and HL7 accountability requirements. To enact such a policy, a simple mechanism should detect incoming and outgoing traffic to healthcare services and *duplicate* these requests to one or more known logging services.

Hide internal IP data when transmitting statistics reports. This requirement stems from HL7-5's protection of data with respect to location, as well as IDPC's data minimization principle. To enact such a requirement, a mechanism is needed to *filter* traffic outside of the medical network, that is targeted to a statistics reporting service. This can be achieved by *modifying* those parts of the messages being transmitted that can identify the IP or any other sensitive data.

A radiologist cannot perform X-ray scans that amount to more than 10 hours a week. It is known that X-ray scans present a health risk not only to patients but mainly to hospital staff who operate scanners for several patients. In line with HL7-2, HL7-5 and radiology hazard standards, hospital management decided that, given the radiation emissions of hospital scanners, the staff's exposure should be limited per week. Since radiologists carry smartcards that allow them to operate X-ray scanners, the system should not authorise that the scanners are operated by a staff member who has surpassed the allowed weekly exposure. To enact such a policy, the system should be capable to differentiate between its users, and should need logic to compute the duration of an X-ray scan operated by a radiologist. Anytime a radiologist would request access to the scanner, the request should not be serviced unless the 10 hours a week have not been exceeded.

Tele-assistance staff can access patient data only while there is an ongoing call from a patient. In line with privacy and security requirements such as HIPAA-1, HIPAA-2.2 and IDPC-2, and in order to prevent accidental access to patient data, the system should condition access to a data service based on the existence of patient-initiated requests. This assumes that the system can detect both patient requests and

tele-assistance staff requests for patient data, and can also determine their sequencing.

Enforcing these policies at application-level would be inefficient because of two reasons. First, these policies rely on message-level information that is not directly accessible at application-level: location, domain and IP; request types, operation duration, low-level event sequencing. Second, enforcing these policies needs to cover several endpoints. As explained above, it is very difficult to maintain multiple application-level enforcement components and change them simultaneously when the policy changes. At the same time, these policies concern security issues that the hospital system should obey, irrespective of its architecture and deployment. We argue that enacting the policies above can be done by a dedicated component at infrastructure level that can interpose between healthcare service providers and consumers. In this way, the *enforcement process* (i.e., the set of actions that the system performs to enforce a security policy) would obtain more efficiently the needed low-level data. Moreover, the changes in the policies and in the security mechanism would be independent of the application endpoints, and the application logic.

4.3 Conceptual Problem and Approach

When applications that span various domains and technologies interconnect, the resulting communication links among them face increased risks of misuse or abuse. For instance, unsuspecting services might get requests that are not authorised, not valid or that are purposefully malicious. Such requests can be malicious in several ways:

- they can violate terms of service usage agreement, e.g., the limit of X-ray scan exposure a week, or the condition of accessing patient data only after a call;
- they can violate confidentiality policies across the system, e.g. leave the patient data in clear when sending reports to third-parties;
- they can bypass the system so that they couldn't be logged;
- they can cause runtime errors in the system, e.g., cause abnormal loads to the service provider, rendering it unable to cope with legitimate requests.

These misbehaviours are observable at the message level, and can be either purposefully malicious, or genuine errors of users or administrators when implementing security policies. As of now there is little control over *when* such violations happen and how they can be caught.

Existing SOA approaches to prevent generic security policy violations fall short of this expectation from two points of view. First, most of existing work locates

policy enforcement mechanisms inside an orchestration engine such as BPEL, but such a view is not suited to scenarios where policies concern the messages themselves instead of the effect they have on the business process. Second, current approaches focus mostly on simple access control policies instead of complex organisation-wide policies that also include controlling service usage. What is hence needed is a way to formulate and enforce security policies that make service misuse or abuse less likely.

The ESB should be in charge of enforcing message-level policies. It is true that ESBs were designed for integrating services, rather than inspecting the adherence of communication flows to security policies. Yet, up to now nothing verifies messages *in transit* once the transport session has been established with an authenticated service point (when no special end-to-end measures have been used for encryption and authentication). Access control policies, that check properties of messages rather than their payload, have not yet been considered. We propose to address this issue by implementing flexible, instrumentable, and highly configurable policy enforcement mechanisms at the ESB level. In this way, security policies can be enforced from a central place, onto transiting messages.

This approach gives organisations such as the hospital in our healthcare study the opportunity to separate ESB operators from service providers, and therefore to gain an additional degree of transparency. Moreover, the approach also decouples the enforcement logic from the application logic. This way, process and application designers can focus on the semantic (or functional) aspects of their applications and how they must be used, but not how this is technically enforced.

In what follows, we will present our solution to enforce security policies on message flows, at the level of the ESB. Our solution is named *xESB*. The next section describes what xESB calls the *enforcement process*, i.e. the sequence of actions performed by security mechanisms in order to collaboratively enforce a policy.

4.4 Enforcing Policies at Message-Level

Generally, the runtime enforcement process starts the moment a policy-relevant event is found; this step is called *detection*, or *interception*. Once obtained, the event is evaluated against the deployed security policies. This evaluation relies on message metadata that the ESB can process, rather than on the message content. This step is called *decision making*, or just *decision*. The result is a decision that the enforcer translates into a series of actions; the enforcement system is in charge of performing those actions (either directly or by delegating them to a trusted third-party). This last step is called *reaction* to the interception event, or (enforcement) *action performing*. This three step

```
1 InOnly[ id: ID: 192.168.233.83-1228875ebcb-8.0
2  status: Active
3  role: provider
4  service: {http://www.microsoft.com}acceptReceiver1
5  endpoint: acceptor1
6  in: <?xml version="1.0" encoding="UTF-8"?><example id="123"/> ]
```

Listing 4.1: A sampled JBI-normalised message: the pattern of the message exchange is InOnly; the message has an identifier; the status of the exchange is “active"; the source of the message is the endpoint “acceptor1"; the destination of the exchange is a service endpoint “acceptReceiver1" at the indicated URL; the XML payload is in the “in" field.

process is called *enforcement life cycle* in Chapter 5.

This section describes how we modelled the policy enforcement life cycle that we later implemented in xESB. The policy language and its interpretation are the subject of Section 4.6.

4.4.1 Interception

Enforcement starts by intercepting messages onto which a policy applies. The policies dictate what message types and parameters about the message to analyse: message destination, source, size, or metadata like annotation information. Irrespective of the message format (usually XML-based), the elements described above are usually easy to inspect on every message simply because all messages on the bus have the same format. A prefiltering mechanism can help the interceptor catch messages with a higher probability of being relevant than any other message; for instance, if a policy refers to requests that are simultaneously outgoing *and* have a valid security signature, then the interceptor could just check if the current call is outgoing. This is a condition which is inexpensive to check compared to the validity of the signature, and if it does not hold then the signature need not be validated.

Listing 4.1 shows the format of a message on the NMR; because of the normalised format, split into structured metadata and payload, the message destination and the direction of the message can be easily extracted and processed. This simple mechanism can separate easily incoming calls from outgoing ones.

4.4.2 Decision

Once a policy-relevant message is intercepted, it needs to be evaluated against the applicable policy (see Figure 4.2). The decision component does policy matching: it examines all policies in the policy base (or repository) by evaluating them against the

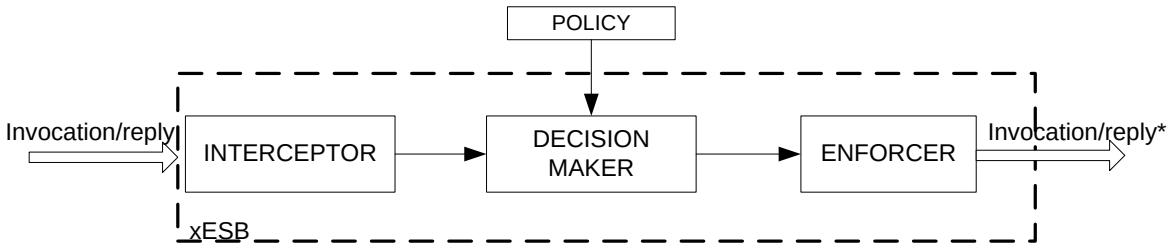


Figure 4.2: The enforcement process behind xESB.

current message. The evaluation is done by comparing message context and actual parameters (e.g., destination service, source service, message type, etc) with the conditions required by interested policies. With the xESB language, we considered the simple case of comparing the message against all policies in the policy base, until the first match is found. With other policy specification languages that xESB, the decision engine might have a different heuristics of reaching a decision.

The output of the decision phase is called the *verdict*. This enforcement decision is binary: either the message is legitimate, or violates a policy. In the first case implies that are no consequences onto the message flow; the second case calls for one or more enforcement actions. These actions are detailed below.

4.4.3 Actions

The third step of enforcement is to *act* based on the verdict reached in step 2. We have implemented five basic enforcement actions that implement the four mechanisms formalised by Pretschner et al. [192] to approach usage control enforcement:

The acceptor accepts whole messages. If the verdict does not indicate any policy violation, then the acceptor is invoked, with the effect that the message is allowed without any modification.

The blocker rejects whole messages. Contrary to the previous case, the blocker mechanism is invoked to react to a policy violation by rejecting the entire message and sending back an error message.

The modifier modifies a message. We want to go beyond the classic "all or nothing" enforcement approach and modify the message to conform to the given policy.

The delayer postpones a message until a condition is satisfied. This mechanism maps to the idea of obligation enforcement, where an actor would be allowed to perform an action only after a condition has been verified. For instance the policy

"Delay authorizing the use medical equipment until there is a supervisor staff member in the room" can be implemented by delaying the request to use that equipment until there is a signal for an authorised staff member in the room.

The executor plugs-in enforcement actions that are application dependent, rather than the predefined ones (delay, modify, block, allow). In some cases, the reaction to a violation may require the execution of a complex recovery process that requires more than the basic mechanisms implemented by xESB. Implementing these mechanisms in xESB would make it inflexible, so xESB uses the executor to invoke an external service or process, which can also be an orchestration engine.

Actions can be differentiated as *preventive* or *corrective*. Preventive actions ensure that a policy violation will not happen: prior to allowing a sensitive action, they check its compliance with the policies. Corrective actions, on the other hand, try to compensate a violation that already happened. The blocker is a preventive mechanism: if a message on the ESB is not allowed to reach its destination, then it is simply dropped. The other actions – modifying, delaying, calling an external entity – are by their nature compensatory: if an intercepted message or a group of messages already constitute a policy violation, an appropriate action must be taken to *correct* the respective message or message flow (e.g., reroute to a secure service, deny further messages on that route until next day, etc.). The executor mechanism can be both preventive and corrective.

We see the actions described above as *enforcement primitives* on ESB messaging. Because implementing a blocking, modifying, delaying and executing mechanism should be different from application to application, we argue that our design caters to a *customizable* ESB enforcement framework. Our model provides a set of basic components – the interceptor, the decision maker, the action performer – and the wiring between them; the semantic behind the enforcement actions and the policies are independent of the solution design.

4.5 xESB Implementation

In a JBI service bus, the component that mediates all communication is the Normalised Message Router (NMR). It is therefore natural to embed an *interposition mechanism* within the NMR, in order to capture incoming messages on their way to their destination, be them service requests) or service replies.

Once the message has been intercepted, the next step is to *analyse* the message just captured. The JBI standard helps in that the messages being routed are normalised before they get to the router; this means that they are transformed to a fixed format that

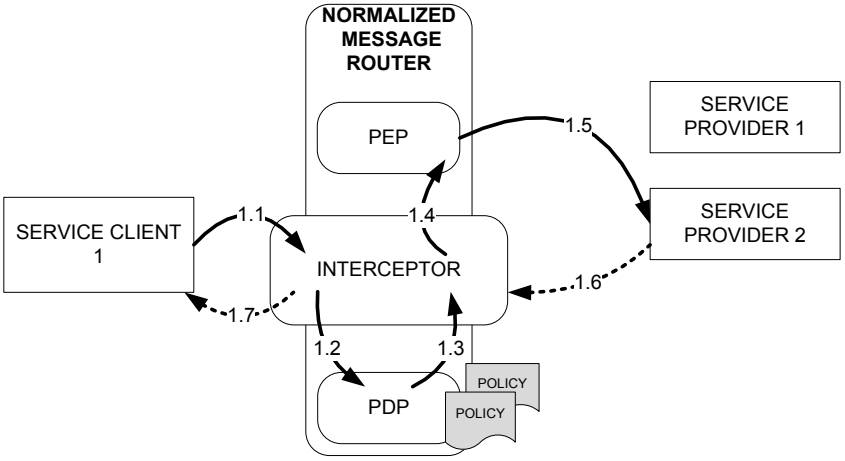


Figure 4.3: The xESB enforcement architecture. The solid arrows indicate the invoke chain, while the dotted arrows show the response chain. Both request and response are intercepted. The policy is only on requests from Client 1 to Provider 2, hence the interceptor gives the request to the PDP, and then passes the response from the PDP to the PEP.

separates clearly the XML payload of a message, the message context (metadata) and message attachments (a format is given in Figure 4.1). This feature makes it feasible to analyse parts of a normalised message. In order to derive a verdict, we designed the analysis component to compare the metadata of a current message with the metadata specified by the policies in the policy base. By “metadata", we mean information such as message source and destination. By “context" we mean flow-related conditions, such as a message flow precedence (e.g., an incoming call was not logged before it was accepted), usage control-specific constraints that mainly involve counting (e.g., a radiologist reached 10 hours of usage of the X-ray scanner) or obligations (e.g., the obligation to monthly send a statistics report to the regional healthcare authority).

Once a verdict is reached, the *action performing* phase consists of one of the actions given in Section 4.4.3. xESB implements message blocking by sending the current message not to its desired endpoint but to a loopback interface (the request initiator can receive nothing or a fault message). The xESB modifier simply replaces parts of the message header as specified by the policy. The delayer routes the message to a delayer component that puts the message in a queue and removes it again when the condition associated with it is satisfied (time or boolean condition). Lastly, the executor is implemented by routing the current message to the endpoint of the desired service. An underlying assumption of the design is that the xESB mechanism is always invoked because the NMR processes every message. The xESB is trusted to

always invoke the PEP, and the enforcement actions are on services deployed on the ESB (a service not deployed is not known to xESB). Hence xESB acts as a centralised execution monitor.

Performance considerations made us embed the modifying and rerouting functionality into the component that does message interception on the bus. The specific delaying and blocking behaviour were implemented as separate components (namely, plain Java objects) to which the NMR would direct messages. Thus, while the intercepting mechanism is ESB platform-specific, we aimed to make the analysis and action hooks reusable across applications. The logic to decide on the enforcement verdict can be reused irrespective of the deployed policies, but the actions matching this verdict are application-dependent. This design is *flexible*: it can support the call-back to an analysis component that is application-logic aware, that might be interested to analyse message payload in order to make a more informed enforcement decision.

4.6 Enforcement Language

As we have described in Section 4.4.3, we wish to enforce not only by control, but also by reaction. Reaction covers temporal and transient obligations to which any entity in the distributed environment can be bound. Existing languages and language frameworks address only some of these aspects (see Section 4.10). In what follows, we will describe two policy languages that we have used with xESB.

4.6.1 The xESB Language

To express message-level policies easier, we have created our own policy language. Unlike other languages, the semantics of the xESB language refers to messages or message flows on the bus, and allows the policy writer to easily express conditions on message-related information. These conditions can involve message sequencing, message cardinality, time triggers.

A policy is written as text and then compiled into binary form. During execution, the compiled policy file resides in memory and is interpreted by a stack machine. In our implementation, we have considered that at most one policy applies to one message from an exchange. This also means we can handle the case when every message of an exchange is subject of a different policy. In terms of *syntax*, a policy is a file that contains several elements:

A default-action. A default action means what decision to take by default when a message arrives and does not match a concrete policy. This allows *allow-based* or

```
default-action { allow; }
// Total duration of X-ray scans, in seconds
hash scanDuration = 0;
timer resetDuration = next week;
obligation {
  if invocation
  when { resetDuration.fired }
  do {
    clear scanDuration;
    arm resetDuration fire next week; }
}
obligation {
  if response
  when { h "Type" equals "X-RayScan" && h "Success" equals "True"
    && source contains "X-RayMachine"
    && h "User-Role" equals "Radiologist" }
  do { update-counter scanDuration[source] += h "Duration"; }
}
rule {
  if invocation
  when { h "Type" equals "X-RayScan" && h "User-Role" equals "Radiologist"
    && source contains "X-RayMachine"
    && scanDuration[source] > 36000 }
  do { block; }
}
```

Listing 4.2: xESB policy for hospital radiologists cannot perform X-ray scans for more than 10 hours a week. We assume the message metadata on the bus contains the necessary keywords.

deny-based policies.

Zero or more initializations. Initializations are of items of state, which are pieces of data that keep their values across policy checks. There are three types of state: *counters*, *timers*, and *hashes*. The latter keep arrays of state, thus allowing state per user or state per message source etc.

Zero or more obligations. Obligations are performed by the system at runtime.

One or more rules. Rules are formulated in an event-condition-action format.

- The event can be either a SOAP request or a SOAP reply.
- The condition can be a complex predicate that evaluates to true or false, and consists of one or more sub-conditions on state and/or message properties.
- The action can be either one of a set $allow, block, modify, delay$ and can also include a separate message duplication action.

Rules and obligations are very similar. *Rules* compute verdicts such as `block`, `allow` and so on. When a rule that carries a verdict matches a message, the action part of

that rule is executed and processing is stopped. On the other hand, *obligations* exist solely for the purpose of updating state, so processing continues.

The event part of a rule or obligation checks if the message is a *request* (or invocation) or a *response*. Request and responses can be easily detected at Conditions are part of a rule or obligation and *checks whether the rule or obligation applies* to the current message. The action specification of a rule or obligation can *update state* (both rules and obligations) or *return a verdict* (rules only)

Listing 4.2 shows how the policy "Hospital radiologists cannot perform X-ray scans for more than 10 hours a week" would be expressed in this language. What is the overall effect of this policy file in Figure 4.2? The first obligation takes care of re-arming the timer if it has fired. The second obligation updates the total length of X-Ray scans, and the rule blocks scan requests (invocations) in excess of ten hours a week. Identifiers such as 'type', 'source', 'destination', etc. refer to names of the metadata fields in the normalised message. While this example illustrates the main features of the language, some other features of interest include:

Modifying message metadata. The language construct "`modify h` *metadata-name* = *string-expression*" modifies parts of a message's metadata, i.e., anything outside the message payload. Modification lets the message pass after modification.

Delaying a message. One possible verdict is "`delay` *n*", which means to delay the message by a specified amount of time. This implies allowing the message to pass eventually.

Delay until a condition is met. Another innovative verdict is "`delay until` *condition*", which will delay a message until a certain condition is met. The condition can be any boolean expression on the state (but not on any message headers). This is not the same as blocking a message, since a message is completely *discarded* when it is blocked, whereas here it is merely *delayed* on the way to its destination.

To show that we can use xESB to enforce cross-service and hence potentially inter-organisational policies, let us consider the regulatory requirement to hide local IP from message ID when transmitting statistics reports. Since this policy holds for all hospital services that can communicate with one or more external statistics services, it affects potentially many services and therefore also potentially different organisations connected to the hospital. Listing 4.3 shows how to express this policy. Note how simple this policy is to implement on the ESB level. On the BPEL level, it would be much more complicated, because a generic IP-hiding service would have to be written and deployed, and message transformation would have to be performed at the BPEL

```
default-action { allow; }
rule {
  if invocation
  when { h "Type" contains "statistics-report" }
  do { modify source "ID" = "0.0.0.0"; }
}
```

Listing 4.3: Policy expressing “Hide local IP when transmitting statistics reports".

```
default-action { allow; }
rule {
  if invocation
  when {
     destination contains "tele-assistance"
     || destination contains "tele-derm"}
  do { duplicate
    "http://internal.hospital.it/log"; }
}
```

Listing 4.4: Policy for “Log all requests to tele-assistance and tele-dermatology services.”

level. (Note that the modification of the source address might be dangerous when a response is required; in this case we assume the message exchange is asynchronous).

Listing 4.4 and Listing 4.5 shows the two remaining policies from Section 4.2, namely “Log all requests to tele-assistance and all requests for tele-dermatology services.”, and “Tele-assistance staff can access patient data only if there has been a patient call".

A compilable Java-based description of the xESB language is given in Annex B.

4.6.2 POLPA

The xESB policy specification language is just one of the possible policy languages that can be used with the xESB model. Other languages can be plugged in easily to replace our own. For instance, in our previous work [82] we used another language for expressing SOA message-level requirements, called *PO*licy *L*anguage based on *P*rocess *A*lgebra (POLPA). With POLPA, we can correctly express any constraint on the usage of the enterprise infrastructure, because we showed that the POLPA model is equivalent with the UCON model.

POLPA represents the allowed behaviour of users by defining sequences of allowed actions, along with any authorisations, conditions and obligations that must hold before, during and after the execution of each action. Deriving from process algebra for concurrent systems, POLPA can naturally represent complex execution patterns e.g., sequencing, parallelism, alternative events. Also, unlike other policy languages

[0]The `modify` action implies a verdict, hence this is indeed a rule, not an obligation.

```
1  default-action { allow; }
2  hash callOngoing = 0;
3  obligation {
4    if invocation
5    when { h "Type" equals "external-call" }
6    do { update-counter callOngoing[destination] := 1; }
7  }
8  obligation {
9    if response
10   when { h "Type" equals "external-call"
11       && h "Success" equals "True"}
12   do { update-counter callOngoing[destination] := 0; }
13 }
14 rule {
15   if invocation
16   when { h "Type" equals "get-patient-info"
17       && callOngoing[destination] == 0 }
18   do { block; }
19 }
```

Listing 4.5: Policy for "Tele-assistance staff can access patient data only if there has been a patient call."

(e.g., XACML), POLPA can express *continuous usage control*: resource usage can be ceased if initial access conditions are no longer valid. In a previous paper [82] we have discussed the formal syntax of this language. In this chapter we will only briefly describe how continuous usage control is expressed, based on the work mentioned before [82]. Given that the triple (s, o, r) represents the access performed by a user s to execute the operation r on the object o, the main POLPA primitives are:

$tryaccess(s, o, r)$ performed when the subject s requests the access (s, o, r).

$endaccess(s, o, r)$ performed when the access (s, o, r) ends.

$permitaccess(s, o, r)$ performed by the system to grant the access (s, o, r).

$denyaccess(s, o, r)$ performed by the system to reject the access (s, o, r).

$revokeaccess(s, o, r)$ performed by the system to revoke the ongoing access (s, o, r) .

$update(s, a)$ performed by the system to update the attribute a of subject or object s.

These primitives can be grouped in a policy, together with the sequencing('.'), parallel ('rep'), and alternative operator ('or'). Like in the xESB language, POLPA also has the 'deny-all' and 'allow-all' operators. For example, the policy in Listing 4.6 denies the user to access the service called *getActivityReport/*, because it is known to the security system that the server is overloaded. In particular, line 1 of the policy refers to the access request for a service. If the access is detected, the policy enforcement proceeds

```
1 tryaccess(user, net, get_destination(url)).
2 [(url == "http://www.hospitals-trento.it/getActivityReport/")].
3 denyaccess(user, net, get_destination(url)).
4 endaccess(user, net, get_destination(url)).
```

Listing 4.6: POLPA policy that blocks access to a known service.

```
1 tryaccess(user, net, get_destination(url)).
2 [(url == "http://www.hospitals-trento.it/getActivityReport/")].
3 execute(delay).
4 permitaccess(user, net, get_destination(url)).
5 endaccess(user, net, get_destination(url)).
```

Listing 4.7: POLPA policy that delays access to a known service.

sequentially (the '.' operator) to the next instruction. Line 2 checks whether the URL corresponds to the one known to be overloaded (or security sensitive). If the condition holds, the execution continues to line 3. Line 3 disallows the access, and line 4 ends the access. In this policy and in the others we do not use the default operators.

To an extent, the pure UCON model puts some limitations over policy enforcement. First, UCON enforcement is binary: it either allows or disallows the usage of an object by a subject. This might be draconic, for instance blocking the *getActivityReport/* might have side-effects over the execution of the calling program. Second, the UCON model does not separate between enforcement actions delegated to a third-party or performed on the fly. Worse, the original model makes no separation between system and subject obligations, nor between pre- and post-obligations, as noticed also in [118]. To address these issues, in [82] we emphasised the executor as an active mechanism that can either execute an action or delegate it and obtain the results.

Consequently, we extended the initial POLPA semantics with the $execute(c)$ statement, that gives the possibility to execute an external trusted action. This action is an addition to previous approaches and is the core of the system obligations. Hence, instead of denying access to a service because of load conditions, it would be more flexible to delay it. This is shown in Listing 4.7 and could not have been done before.

Listing 4.8 shows two connected policies from Section 4.2 – the first and the last one. Namely, the requirement to log all requests that arrive to the tele-assistance and tele-dermatology services, policy which provides the grounds for the constraint to access patient-data only after a call to a patient has already been initiated. The *rep* operator in line 1 of the policy allows the parallel execution of any number of instances of the policy from line 2 to line 11. This means that enforcing the policy can handle any number of requests to and from the tele-assistance services, in parallel. The *tryaccess*

```
1  rep(
2   tryaccess(source, destination, request(req_type, caller_name)).
3   [(destination = "Tele-assistance") or (destination = "Tele-dermatology")].
4   permitaccess(source, destination, operation).
5   execute(writelog(source, operation)).
6   endaccess(source, destination, operation)).
7   tryaccess(staff, EHR-database, getRecord(patient_name)).
8   [(staff = "tele-assistance") and (req_type = "patient-call") and (patient_name = caller_name)].
9   permitaccess(destination, EHR-database, getRecord(patient_name)).
10  execute(writelog(staff, EHR-database, getRecord(patient_name))).
11  endaccess(destination, EHR-database, getRecord(patient_name))
12 );
```

Listing 4.8: Expressing policies 1 and 4 of Section 4.2 in POLPA.

primitive in line 2 checks for a request that, if is addressed to a tele-assistance or tele-dermatology service category (we assume the '=' operator is able to elicit that data, in line 3) then the request is permitted to pass (line 4), but immediately recorded in a log (line 5). If it happens that, *after* this sequence of actions, there is a request to the EHR database (line 7) from tele-assistance staff, looking for the data of the patient that has called before (line 8), only then is the access to the database allowed (line 9) and logged (line 10). Otherwise, the access to the EHR-database would not be allowed (we assume a deny-all default action).

By extending POLPA, we can support a more flexible way to enforce a usage control policy than existing approaches. Our enforcement action can vary from a set of possible ones (delay a message, log, modify message payload or metadata, etc.). Because the PDP gives a verdict but does not necessarily know what actions the policy associated to that verdict, these actions can be expressed in a language that the PDP is agnostic of. These actions will be eventually executed by the PEP, and the PEP can be an altogether different engine than the PDP.

4.7 Extending Predefined Enforcement Actions

To enact a verdict, xESB can perform predefined actions (accept, block, modify, delay for a time duration, or delay until a condition is met, duplicate), either explicitly in the policy file (the xESB language has a one-to-one mapping with the action keywords) or implicitly with the *execute* primitive in POLPA. Having presented *what* the actions can be, the next question is *how many* can they be?

Section 4.6.1 shows that both policy languages support expressing not necessarily one, but a set of predefined actions as to perform. Performing more than one action can be required when, e.g., both a system alarm to the administrator, and banning a

user's access, need to happen on the user's fifth failed login attempt. This policy is likely to exist in a system that needs to *both* react on the concrete event that caused a security issue, and on the system context. For such cases, the xESB language already supports, in its 'do' part of a rule, a 'duplicate' action to an endpoint before or after any other predefined action; POLPA can also be made to launch several reactions as a result of intercepting a message that causes a policy violation. This approach is certainly more flexible than a binary allow/block course of actions, but can xESB react in a customizable fashion to the incident just intercepted?

With either language, the xESB system can invoke a trusted Web service that in its turn, would trigger a more complex reaction to a violation. This reaction would no longer belong to the predefined actions above, but would allow the policy writer or security administrator to weave in their own incident response solutions. With this purpose in mind, in the EU-FP7-MASTER project we have integrated xESB with an external tool called ControlCockpit [42]. The ControlCockpit had the purpose to allow the system administrator track and influence the policy enforcement process on a SOA system at runtime. Hence, we have shown that the *executor* defined in Section 4.4.3 and already supported with POLPA, can execute not only a set of predefined actions, but also a customizable set of actions allowing human intervention.

4.8 Threat and Trust Model

The *trust model* of xESB comprises the software and data that are essential for correct xESB policy enforcement. We have assumed that the main ESB (and hence, also xESB) is owned by one authority - the hospital. This means that the NMR router, the interceptor and the action performer are trusted to perform their roles correctly, since they are built in with the ESB. The decision maker (either supporting POLPA or xESB language policies) should be owned by the hospital, or, if another one is plugged in instead, it should be fully trusted to correctly make enforcement decisions. The same holds for extra any action performer (e.g., the ControlCockpit, a BPEL process, etc.) added to extend the enforcement process. Apart from these system components, trust must be put in the policy writer. Correct policy enforcement relies on a correct specification of how the system should react in face of a potential violation. These assumptions hold for the work reported in this chapter, and also for the next chapters that build on it.

Since xESB is a centralised system that enforces security policies, it would be the main attack target. Generally, at message-level, attacks could exploit weaknesses in access control from one service to the router and to another service. The current

version of xESB is instrumenting ServiceMix3, which already supports authentication and authorisation for services and users. Despite not being able to act per operation on the endpoint side, the ServiceMix authentication and authorisation are by username/password on the invocation of a component deployed on the bus. This is supported with the WS-Security UsernameToken profile, which is included in the SOAP request[1]. Basically, for every message before being handled by the JBI interceptor, there are two security-relevant interceptors to process it: one interceptor parses the WS-Security header (XML Signature checks, extraction of the password and username), and another interceptor that is in charge with both authentication and authorisation based on Java Authentication and Authorization Service (JAAS). JAAS in ServiceMix makes name/password/role callbacks to a configuration file that stores this data. In all, the result of the JAAS interception can be either an authorisation (then the message goes to the NMR) or an exception (a fault is sent back to the client).

Having made this observation on ServiceMix, we assume that other ESB platforms would offer similar access control features at the service endpoint level. Next, we want to examine xESB's *threat model*. We list a number of possible attacks, and describe how xESB deals with them or how it can be protected against them:

Bypassing the service access control. As explained above, no message could bypass the authentication and authorisation that the NMR performs with respect to the message destination. This scheme is coarse because it is per destination service rather than per operation: a single username/password combination to access a service allows to perform all operations of the service. The mapping between users and their username/password pairs is located within the configuration file of xESB, so if xESB is completely trusted, then authentication cannot be subverted. Authorization maps of services with principal roles is by default off in ServiceMix, so it needs to be is populated in order to work.

Denial-of-service attack. ESBs can in general be subjected to a DoS attack, when they are flooded with legitimate requests that have been authenticated and authorised. xESB can actually help fend off such attacks by deployed the policy given in Listing 4.9. The policy keeps track of the number of times there is the same response (in this case, it is “failed" assuming there has been some fault on the reply chain) to a client trying to access a service (called “reportingService"). If there have been more than 10 such replies, further communication from that client to that service, is blocked.

[1]By construction, the username and password data is written in clear in the message header. ServiceMix recommends the additional usage of XML Signature and XML Encryption.

Man-in-the-middle attack. If we assume that the ESB is trusted and cannot be tampered with, there are two possible locations for a man-in-the-middle attack. First, an attacker can act as a proxy for the unsuspecting consumer: it can intercept the valid SOAP invocation that will later be authorised, and receive the response to give to the consumer. There is little risk in this case if the consumer uses encryption and signing from the beginning. The second case is trickier: when the attacker poses as a bogus service as a destination service, and find some mechanism to make the NMR route the message from the consumer, to this bogus endpoint. This situation cannot be controlled by xESB as it is: what is required is a strict control over the services that are deployed on the bus.

Message injection. Messages cannot be injected within an existing message flow, because they would require a sequence number that only the NMR can attach. Since the NMR is trusted, and is the only one to control the sequence number allocation for each message within a message exchange, injection cannot happen.

Timing or time-related attack. A malicious consumer that has been authorised already can observe that some invocations take longer time than others to pass through. By sending different invocations to one or more services, the invocations could be undetected by the DoS constraints and the consumer might gather some data on response times in connection with the services that are invoked, and the structure of the SOAP invocation. Timing attacks are inherently difficult to counteract, yet it would be challenging to investigate first what an attacker can infer about the system, without knowing the exact policies enforced by xESB.

4.9 Performance Evaluation

To implement xESB, we chose Apache Servicemix 3.3, a JBI-compliant open source ESB. We used the Servicemix API to intercept messages according to Figure 4.3. This section describes the evaluation of our prototype implementation with the initial xESB language.

We followed a capacity testing model [229] to measure the lower bound of the instrumentation overhead. We used the sample SOAP that come with ServiceMix 3.3, and soapUI as a tool for load generation[2]. We used SOAP messages of 8 Kb size and a varying number of parallel clients. Our testbed PC was a 32-bit system with a 2.6GHz processor and 3GB of RAM. The JVM was allowed 1280MB of memory, with a ServiceMix queue size of 256 requests.

[2] `http://www.soapui.org/`

```
1  hash fails = 0;
2  obligation {
3    if response
4    when { source = "reportingService" && h "Status" = "Fail" }
5    do { update-counter fails[destination] += 1; } }
6  rule {
7    if response
8    when { source = "reportingService" && h "Status" = "Success" }
9    do { update-counter fails[destination] := 0; } }
10 rule {
11   if response
12   when { source = "Fail-Admin" && h "Status" = "Success" }
13   do { allow;
14        update-counter fails[h "Subject"] := 0; } }
15 rule {
16   if invocation
17   when { destination = "reportingService" && fails[source] > 10 }
18   do { block; } }
```

Listing 4.9: xESB policy to prevent a DoS or brute force attack to a service.

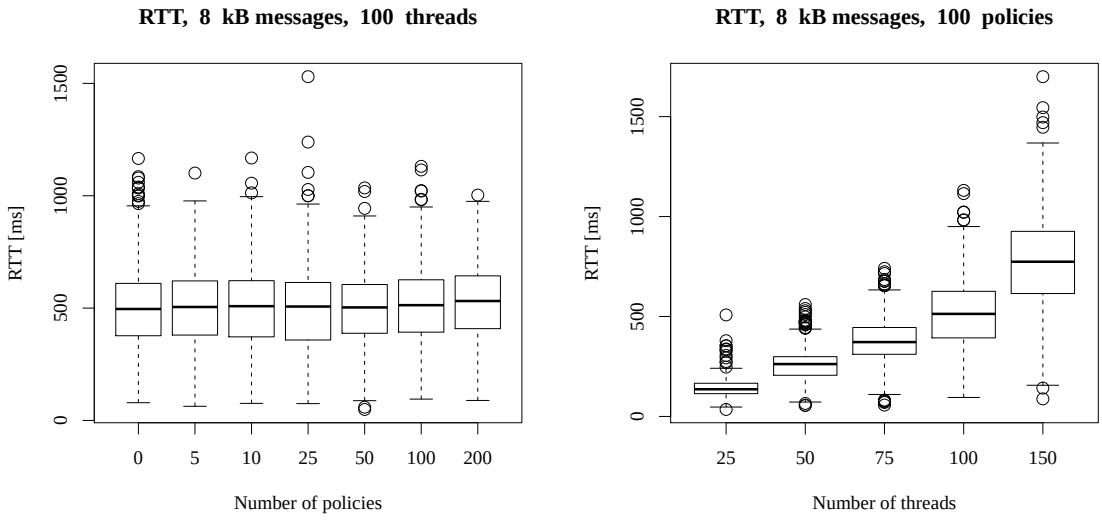


Figure 4.4: RTT for varying policy file sizes (left) and varying number of parallel connections (right), for 8 Kb messages. The x axes are not uniformly spaced.

To answer the question "how does the number of rules in a policy file affect the round-trip time (RTT) of messages?", we constructed policy files of 0, 5, 10, 25, 50, 100, and 200 rules such that in all files, only the last rule would ever match. Therefore we are actually measuring the effect of checking the rules, not simply loading a larger ruleset, which would have almost no effect. For each policy file, we then used SoapUI to send 8 Kb messages to xESB using 100 parallel threads for 3 minutes. We repeated this process 3 – 5 times. Since SoapUI does not return the RTT for individual messages, we looked at the average RTT. The results are plotted in Figure 4.4, left. The most important conclusion is that xESB is almost unaffected by the size of the policy file: the main delay seems to be in message processing, not policy enforcement. In fact, profiling shows that policy enforcement takes only about 0.2% of CPU time.

The next question we asked is "How does the number of parallel clients affect the RTT?". Using a similar method as above (8 Kb messages using 100 rules for 3 minutes, repeat 3–5 times, then look at the average RTTs), we arrived at Figure 4.4, right. As expected, both the average RTT and the variability rise linearly.

A curious feature that can be seen in the figure is the tremendous variability in RTT. Since this variability also shows in the uninstrumented ServiceMix, we conclude that policy enforcement is not responsible for this. We conjecture that this is due to the staged-event architecture of ServiceMix, where processing is done in bursts because computation happens in stages[3].

4.10 Related Work

There is a large number of security standards that cover XML message validation, authorisation, encryption or even federated authentication: OASIS's WS-Security, SAML, XACML[4], and other WS-* specifications. These standards only deal with particular narrow issues of SOA scenarios (point-to-point authentication, authorisation, message integrity, etc.) and not with adhering to security policies that apply across the SOA enterprise.

The work on enforcing security at the message-level is limited to considering access control policies. The solution of Svirskas et al. [221] is limited to controlling service access and logging on the ESB. A similar approach [137] suggests several infrastructure security services to act on different types of events by means of a gateway, but the separation between business and security concerns is not clear. Maierhofer et al. [146] describe a dynamic enforcement framework for security at the message-level, but do

[3]Staged Event-Driven Architecture `http://www.eecs.harvard.edu/~mdw/proj/seda/`

[4]These OASIS standards can be found at `http://docs.oasis-open.org`

not discuss interoperability nor implementation. Another solution [86] suggests a custom security service bus for enforcement of complex policies. Other conceptual approaches based on law-governed interactions [130] aim to model enforcement of laws onto communication between servers and clients, but they consider generic distributed dedicated entities to perform the law realization. Our assumption differs in that it uses the ESB as a central-view mechanism that either performs on-the-fly enforcement or delegates it to a trusted entity. We explicitly focus on access and usage control policies, and unlike other approaches, we offer a concrete model and a proof-of-concept implementation.

Concerning the policy language, elaborate access control languages (e.g., Ponder [45], EPAL [18], SPL [200]) are unable to express obligations that pertain to usage control. A more generic usage control centric language is OSL [98], but from our knowledge it is not supported by an implementation. Unaware of any implementation of a generic usage control language that supports compensations, we have developed a proof-of-concept policy language fit for the ESB, and also integrated POLPA [19]. Compared to theoretic works on access control compensations [191], violation management [32] and obligation assessment [109, 108], we go beyond access control and provide an implementation of compensations on the fly. This means that whenever a violation of some service usage rule is detected, the correction happens as the event travels through the system, before it reaches an interested party.

In usage control enforcement [185], Katt et al. [118] add the notion of post-obligations to the obligation model; they consider and implement a mechanism for ongoing enforcement. The work of Pretschner et al. [192, 193] describes a formalised usage control language and the mechanisms to enforce such a language, but do not cover an enforcement model for SOA. xESB can support all these concepts (pre-, on- and post-obligations, either to the system, or to a third-party), with either the xESB language, or POLPA. Yet xESB is the first concrete usage control policy enforcer at the ESB level.

4.11 Summary

This chapter presented xESB, an instrumented JBI-based ESB for the enforcement of *cross-service message-level policies*. xESB is able to enforce both access and usage controls policies. We have shown how xESB can work with two policy languages: our own language for message-level constraints, and an existing language called POLPA. The rich enforcement semantics of xESB allows not only to reject ESB messages that violate a policy but also to react to that violation in several ways: by performing one or more predefined actions on a message or message flow that is about to violate a policy; or

by performing a complex set of actions that can include human intervention.

One other advantage brought by the xESB approach is that policy enforcement can seamlessly adapt to evolving policies. If a new security policy replaces an existing one (e.g., not hide, but *encrypt* sensitive data in outgoing traffic), then the changes on the enforcement logic are as light as changing a service endpoint (from one that hides data to one that encrypts data). This behaviour is in resonance with service-orientation and reuse. Such a decoupling has not been previously addressed at the message level, and our solution is flexible in that any trusted components able to perform security actions or make security decisions can be seamlessly plugged in.

A natural limitation of xESB is that of using message payload (rather than message meta-data) when making security decisions. It is with no doubt possible to inspect message content when it is not encrypted, but xESB cannot understand inter-service communication more than what an ESB can natively understand. To address this limitation, the next chapter describes a novel way to combine enforcement capabilities at message-level with those at business process level, in order to achieve a more powerful execution monitor for distributed applications.

Chapter 5

Cross-Layer Policy Enforcement

5.1 Introduction

xESB offers a powerful capability to monitor communication across a distributed system that uses an ESB to integrate its components. Using an ESB offers transparence to the communicating parties and non-bypassability, and these are very strong advantages for a security monitor. xESB uses this power to enforce constraints at the level of messages, such as the requirements to log system activity, not to disclose data to unknown or unauthorised endpoints, message sequencing and validation, message transformation (e.g., XSLT) or choice of message routes.

However, xESB is a security-aware ESB and is limited in two respects: first, it cannot natively understand message payload, since it is bound just to message metadata. For this reason, it should not normally bypass the encryption of payload or attachments, in order to use the data thus gathered for security purposes. Decryption of such data is not in the scope of a regular ESB. Second, the ESB does not have enough knowledge about the application in order to perform more *meaningful* actions. Such meaningful actions include more complex reactions to violations, such as invoking several Web services and orchestrating the sequence of calls and responses; can also include state keeping, session reasoning, obligation triggers that invoke Web service sequences, etc.

The above two aspects (message payload understanding and complex action performing) belong, traditionally, to the application level. Security at the application logic level is done with the help of business process engines. Business process engines can understand higher-layer data such as organisational roles, actors and permissions, operation modes for the application, levels of trust among business domains and actors, stateful sessions etc. It hence comes with no surprise that security considerations including security policy enforcement have been addressed using business process engines.

However, it is dangerous to address security constraints *only*, or *directly*, at the application level. The process engine *can* disallow a business process to analyse a request, but that is after the process has *already* received the request. If this request contains malicious code or sensitive information in the header, those will have been stripped off or processed before the business engine can reach the message payload that it is able to process. Similarly, the business process engine would not get to process an otherwise legitimate message payload if the metadata of the SOAP message contained e.g., an infinite XML recursion. The business process engine only cares about a part of the message – the payload – and cannot prevent the whole message from arriving to other destinations, to the wrong destinations, or to none at all. This is because the engine is not designed to control message-level aspects such as message flows between physical endpoints or binding virtual endpoints to actual endpoints. Enforcement at the application level cannot normally go lower than orchestrating actions inside the business flows managed by *the same engine*.

It follows that there are message-level security policies – on message flows and message processing – and business security policies on higher-level events. The former concern message sequencing, and message properties usually included in the metadata. The latter concern business process access control, among others. Enforcing message-level policies at the business process level or business process policies at the message-level is awkward. This is because it requires pulling data and associations that are not available to one level or the other: the business process engine does not normally access message metadata, while the message router cannot normally recognize business process identifiers and properties.

However, some current solutions [227, 117, 157, 20] either drag enforcement of lower-level aspects to the business process level, or only focus on higher-level aspects without considering the lower-level security issues involved. These approaches are also influenced by the ease of using the business processes modeling languages, usually very close to business logic and often intuitive to use (e.g., Business Process Modeling Language). In this thesis, we maintain that considering security directly at the business process engine is not always safe, because it does not enact *least privilege* when accessing potentially sensitive data. This chapter explains this problem, analyses enforcement strengths at the business level and ESB level, and proposes a model of security policy enforcement that transcends both levels. The discussion focuses on the BPEL engine as a particular business process engine; the reason for our choice was that WS-BPEL is a standard language for business process execution. The work reported in this chapter has been previously published in [80].

5.2 Requirements from the Case Study

As shown in Chapter 3, regulations that govern the healthcare area take generic forms for a range of security requirements. For the hospital in our case study, some policies that can be inferred from healthcare regulations are difficult to enforce either just at the ESB level or just at the BPEL level:

Policy 1. For reasons of separation of duties, the same doctor cannot approve a drug prescription and a drug dispensation request. This is in line with HL7-1.

Policy 2. As mentioned in HL7-2, only one authenticated nurse can undertake the duties of a Head Nurse, for a shift of 12 hours.

Policy 3. As mentioned in HL7-3, for mutual exclusivity, a physician can either perform clinical hours, or work in the emergency room.

Policy 4. Access to patient data can be allowed only after there's an emergency call from a patient. This constraint is in line with IDPC's data minimization rule.

To comply with these policies, an enforcing mechanism should understand how to detect and react to the possible violations of each case. For example, enforcing *Policy 1* requires a differentiation between users of the hospital so that a doctor approving a drug dispensation cannot be the same as the one who prescribed the drugs to hospital patients. *Policies 2 and 3* require maintaining the duration of a shift, ways to check authentication for a nurse and for a physician, and managing assignments of permissions to roles. *Policy 4* requires a way to distinguish emergency calls from patients, access to patient data, and tracking action sequencing.

5.3 The Problem

Policy 1 above concerns internal business processes of the hospital, in particular the *drug administration process in BPEL*. Figure 5.1 illustrates the activities involved in the drug administration process, from the request to treat a patient, to drug prescribing, retrieval of the drug from storage, and drug dispensation. We will use this process in a slightly changed form in the remainder of this work. The process is modeled as a WS-BPEL process that orchestrates the Web services connected by an ESB. The motivation of examining the application using an ESB in the first place has been given in the previous chapter.

For every drug treatment request for a hospital patient, a new process instance is created. With respect to Policy 1, a mechanism should check that the *Prescribe* and

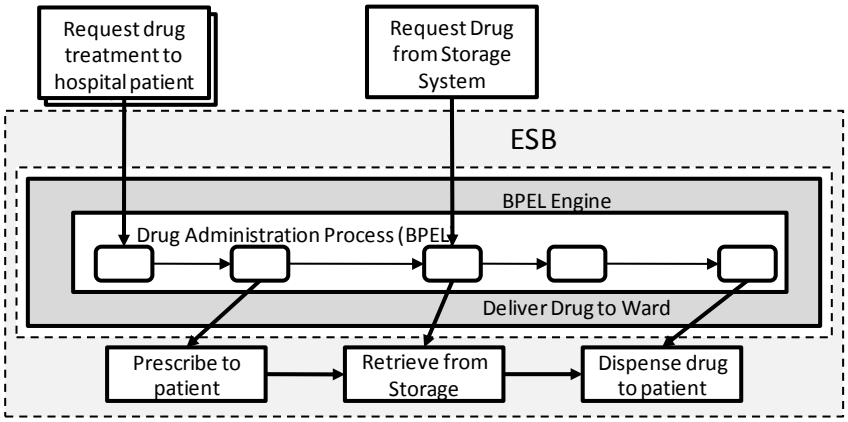


Figure 5.1: The hospital drug administration process.

Dispense activities are performed by the same doctor. If the doctor is found to be the same, the policy would be violated. This check is performed by the BPEL engine.

Enforcing SoD constraints at BPEL level is, to our knowledge, the only existing approach to enforce SoD in service-oriented applications. The engine can inhibit BPEL events performed by actors whose roles are understood by the business process. However, the BPEL engine cannot natively manage role assignments. The engine needs to contact an external entity – a third-party role manager – to check the role/permission assignment for the actor on whose behalf the *Prescribe* and *Dispense* operations have been invoked. Contacting this third-party is prone to attacks or bugs since this third-party is located outside of the BPEL engine e.g., the communication flow between them might be tampered with, or delayed. Therefore, this BPEL-only implementation falls short in that the execution of logical invocations of services cannot be controlled by the BPEL engine down to the message level; also, the BPEL process does not distinguish between service invocations emitted or served by the same host, and cannot manipulate service location information. Location information can be needed in cases like Policy 3. Therefore, BPEL cannot check that *least privilege* holds for the messaging layer (or any lower layer that it relies on). BPEL policy enforcement is secure as long as all the events related to BPEL communication generated in the system are secure.

Can SoD constraints be enforced also by xESB? As it is, and like BPEL, xESB lacks the knowledge to distinguish organisational roles in the role hierarchy (e.g., doctor with the permission to prescribe drugs or doctor with the permission to dispense drugs). Nevertheless, xESB has access to message flows and to location information as well. Location information can be useful especially when enforcing Policy 3, which

separates what permissions are allowed for a role depending on the location within the hospital. Reasoning with location information and relaying third party communication flows needed when enforcing SoD are part of xESB's capabilities. Equally, the power to ensure least privilege for higher levels, by managing message metadata securely, is also part of xESB's capabilities. These observations have led us to propose the first approach to combine ESB with BPEL engine features for better guarantees that a policy is really being enforced.

Based on a study of the different enforcement strengths of ESB and BPEL, this chapter presents an enforcement model that discovers enforcement capabilities advertised at both process engine and message-level. BPEL policy enforcement can be made more secure with message-level policy enforcement as offered by xESB; and conversely, xESB mechanisms can observe violations of message-level policies that can be treated by triggering higher-level processes. Since for management reasons it is advisable to have all policies in one language and in one place (since this makes deployment and review much easier), our model has a central view. It decides where to enforce which policy constraints, and how to map these constraints to the existing capabilities at the right level. For instance to enforce the SoD constraint above, our mechanism can combine the BPEL-level check with ESB's capability to resolve the second operation to an endpoint that is surely operated by a different doctor; this would be a way to reinforce a high-level rule with low-level mechanisms, when the assumption is that each doctor should use a dedicated device with a known address or address range (e.g., a PDA, or a tablet).

The following section analyses in more detail the limitations of performing policy enforcement only at the level of business processes, or only at the ESB level. Focusing on the SoD example of Policy 1 does not restrict generality for the enforcement of policies 2-4, that require similar BPEL *and* xESB capabilities.

5.4 Enforcement Capabilities of xESB vs. BPEL

As explained in Section 4.4, we use the term *enforcement life cycle* to mean a set of policy enforcement actions grouped in three steps: the first step is *to detect* the events relevant to the policy; the second step is *to decide* whether the detected events constitute a violation of the policy; the last step is *to react* as specified by the policy. Drawing this line helps to describe the shortcomings of current approaches.

5.4.1 BPEL Engine Enforcement Capabilities

To detect policy violations, a generic BPEL engine can employ three main categories of *standard capabilities*: observation, event triggering, and event aggregation.

Observation of events. BPEL engines can observe events occurring during the execution of a process. The execution history of all process instances is saved in the audit trail, and the data can be later used for an analysis if a certain condition has become true.

Triggering of events. BPEL engines are capable of triggering events at state changes during the execution of a BPEL process. Events that occur during the execution of a BPEL process are saved in the audit trail. The audit trail can be used to monitor the execution of all running instances of all deployed process models on a BPEL engine. A process instance can also be started when a message or event has arrived at the BPEL engine.

Aggregation of events. In a BPEL engine, event aggregation is implicitly done by an internal component called the “process navigator". The navigator is responsible for the execution of the activities defined in a BPEL process. One feature of the navigator is the initialisation of a new activity if the required previous activities have completed or faulted. The navigator of a BPEL engine is notified by an event that the preceding activity has ended. Such events can also be made visible to the outside for complex event aggregation, e.g., by emitting an event containing the aggregated information of several events.

Functionalities covering the reaction step of the enforcement life cycle include the **suspension** and **termination** of process instances. These enforcement abilities are part of the standard abilities of the BPEL engine we use in our prototype. To further support enforcement, Khalaf et al. enriched the BPEL engine with additional reaction abilities: functionality to block and unblock processes, to insert, delete, and modify activities, and to modify variables [120].

Termination and suspension of process instances. A BPEL engine can terminate and suspend running process instances. This means it is possible to stop a process instance that behaves in a way that violates policies. Terminated process instances cannot be reactivated; suspended process instances can be resumed.

Block or un-block process instances. The execution of running BPEL processes can be blocked when a certain event occurs. This is the difference to the suspension of a process instance described before. To unblock a process instance another

event needs to occur. This can be, for example, an unblock message coming from a policy decision component.

Insertion, deletion, and modification of activities of a running process instance. It is possible to insert new activities into running process instances. For long running process instances this is a means to implement new policies, i.e. policies that were not present when the process instance was initiated. The changes performed onto process instances can be mapped to the underlying process model. This procedure has been previously called *instance migration* [199].

Modification of variables and activities. Variables and activities of running process instances can be modified to lead the process execution to a direction that avoids the violation of policies.

The BPEL engine inherently keeps track of the state of the process instances running on it. In case of an emergency shut down most BPEL engines persist the states of all running processes on disk so that they can be resumed after a restart. BPEL internal fault and compensation handling is not always visible to the outside, thus it must be signalled and enforced within the BPEL engine. State keeping helps to make BPEL-level enforcement decisions, but an enforcement decision maker on the BPEL engine does not come off-the-shelf.

5.4.2 ESB Enforcement Capabilities

In this section we show enforcement aspects that are typical for an ESB and cannot happen on a BPEL engine. Following the enforcement life cycle concept, detection in the ESB happens specifically on message flows. It encompasses endpoint resolution, observation, triggering of events, blocking of a message flow and transformation of messages. This functionality is provided by default in standard ESB platforms, but to our knowledge it has not been used before for security enforcement.

Endpoint resolution. Endpoint resolution is one of the two main functions of the ESB. Resolving endpoints means matching a virtual endpoint to a physical endpoint in a service registry. The ESB uses its own protocols for this matching. As specified in the JBI standard, endpoint resolution can be static or dynamic.

Observation. The ESB mediates all messages flowing through. These message flows can be logged for later analysis.

Triggering of messages. ESBs are capable of emitting events, usually when a certain message arrives. Hence when a certain message is received, the system is injected

with a generated event. At message-level, both the message and the event can be either a service invocation, a service response, or a fault raised in the application.

Blocking message flows. ESBs offer the possibility to react synchronously to incoming or outgoing messages. This means that when a certain message has been received or is about to be dispatched, the ESB can perform an action before routing the message to its destination.

Transformation of messages. ESBs are capable of aggregating, splitting, filtering, and processing the messages that they route from one endpoint to another. Processing of messages on certain criteria is covered generically by enterprise integration patterns. Standard patterns like the message splitter, message aggregator, and message filter are already implemented in a number of ESB platforms.

The additional enforcement functionalities that were suggested in Chapter 4 and implemented with xESB cover the reaction step of the ESB enforcement life cycle:

Modification of a message. The ESB controls the messages between the source endpoint and destination endpoint. This implies that it can modify parts of the message, be it the message metadata (source, destination, security headers, etc.) or the message payload. Usually the ESB can discern between payload and routing information. The payload is not always accessible, e.g., when encrypted.

Rerouting messages. Rerouting messages can be done on the fly by sending a message not to its intended destination, but to another endpoint. Rerouting can serve multiple enforcement functions, e.g., block the initial destination endpoint from receiving a message or to retain the message for a limited amount of time.

Inserting messages. ESB traffic can be duplicated among endpoints, and the ESB can insert messages into existing flows.

While the BPEL engine handles logical flows between logical endpoints, the ESB handles correspondence with the real endpoints. The ESB features native endpoint name resolution, message processing, message reliability and delivery mechanisms. For instance, the ESB has access to service metadata such as: position information, interface information, intra-service protocols, user tags, and ranking or price of a service. Endpoint resolution means the ability to choose another endpoint for a message depending on various parameters. Non-functional runtime aspects like traffic load and trust can influence the flow of messages between endpoints. The message flow can be controlled by the ESB at the infrastructure level. Similar to the case of the BPEL engine, the ESB does not come with decision making support with respect to any kind of constraints.

Table 5.1: Comparison of new Enforcement Capabilities of ESB and BPEL engine

New Enforcement Capability	BPEL engine	ESB
Modification of Messages	-	×
Rerouting Messages	-	×
Insertion of Messages	-	×
Block / Unblock Process	×	-
Termination / Suspension of Process Instances	×	-
Insertion, Deletion, and Modification of Activities	×	-
Modification of Process Variables	×	-

Table 5.1 shows a comparison of the new enforcement capabilities brought by xESB as per Chapter 4 and the BPEL engine. It summarises our efforts to provide two middleware components with extended support for enforcement in an SOA. The BPEL engine and the ESB have two distinct feature sets in terms of enforcement and combining these capabilities can lead to powerful fine-grained policy enforcement.

5.5 Solution Design

Security policies need to describe what should happen when the system tries to perform a transition from one state to another. In this section we present a model of the information needed to specify a security policy.

5.5.1 Policy Model

With this model it is possible to describe enforcement actions to be performed by the BPEL engine or by the ESB or by both. We use this policy model in Sections 5.5.2 and 5.6 to show the interaction between an ESB and a BPEL engine during the execution of an enforcement action. Figure 5.2 shows the model specifying the content of the enforcement policy. It consists of two parts:

The detective part. Detection specifies the critical state to be observed and how this state can be detected. The *scope* of the detective part refer to the SOA components that need to be observed in order to detect the critical state. For example, for the SoD policy above (Policy 1), the scope is the *prescribe* and *dispense* operations, and the critical state is when the latter operation is about to occur. For Policy 2, the scope relates to all the actions that are permitted to the Head Nurse, and the critical state starts during the 12-hour-shift, after the first nurse who's already performed one action that is only permitted for the Head Nurse. A similar

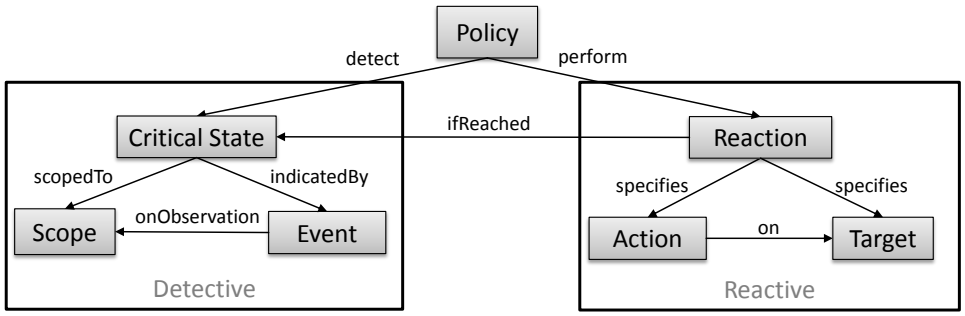


Figure 5.2: The content of a policy as seen by the suggested model.

reasoning holds for Policies 3 and 4 to deduce the critical state and the scope.

The reactive part. The reactive part specifies what actions should be taken if the critical state is reached and what are the targets, i.e., the elements of the event(s) that triggered the critical state. The policy enforcement mechanism would perform an enforcement action onto the targets. For the enforcement of Policy 1, a reaction can be the blocking of the request to perform the *dispense* action. The target is the BPEL invocation to perform the drug dispensation action. For Policies 2, (and similarly for Policies 3, and 4), the reaction is to block the invocations to actions permitted to a Head Nurse but performed by a nurse that is different from the one who performed the first Head-Nurse only action. Blocking the occurrence go an event is only a simple reaction. A more complex reaction would be to trigger a set of business processes that alert the administrator, record a special log event, and block the invocation about to occur.

The policy enforcement mechanism makes decisions that refer to where and how to perform the detective actions, and where the reactions have to be executed. The decision is made based on:

- the policy specification, that refers to two elements:
  - the observational scope;
  - the targets on which the reactive actions need to be performed;
- on the capabilities of enforcement components, described in a service catalogue:
  - at ESB level;
  - at BPEL level.

In this way, detective and reactive parts can be performed by different enforcement components. Currently, the decision if the policy is enforced on a BPEL engine or on an ESB is made at design time (described below). Overall, the policy must have both

a detective part, to specify what to observe in the specified scope, and a reactive part, to specify what actions to perform on what targets.

This separation specifies what capabilities can actually be used to enforce a certain policy, from all the existing policy enforcement capabilities. A BPEL engine would not be able to observe communications between services if this communication is not part of a business process running on this engine. Similarly, an ESB would not be able to control internal events and data flow of a business process. The ESB enforcement actions target *messages* and are scoped to the observation of the messages that flow between the services that use this ESB. BPEL engine actions, on the other hand, target *process*, *process instance*, *process activity*, *process activity instance*. Thus, if a policy specifies:

Every instance of a business process must perform activity $Drug - Dispense$.

In this case, because the policy refers explicitly to the keyword *process*, the target is a business process; consequently, the detection of the critical state should go to the BPEL engine. If the policy says:

All messages between endpoints $S_1, ..., S_n$ must be anonymised.

or

Allow action requests only when performed by a doctor in the hospital.

then, because of the *message* and keywords, the observation scope and the action target is a *message* (that also carries some location information, otherwise difficult to obtain by the BPEL process) and clearly belongs to the ESB. However, service invocations can be controlled by both ESB and BPEL: on the BPEL level by controlling *invoke* activities that invoke a service, and on the ESB level by controlling the invocation message.

To demonstrate the differences and similarities between ESB and BPEL level enforcements in our case, consider the first example regulation specified in Section 5.2: *The same doctor cannot approve both a drug prescription and the dispensation of that drug to a patient*. The critical state of an enforcing mechanism for this policy can be specified as *The doctor to have approved a drug prescription wants to dispense the drug to a patient* with the corresponding reactive part of *Block drug dispensation by this doctor*. To detect the critical state, two services S_1 performing *drug prescription* and S_2 performing *drug dispensation* need to be observed. If these services are invoked from the same business process BP_1, then the observation can be mapped to the observation of the corresponding *invoke* activities of BP_1 and thus can be done on the BPEL engine level. The reactive part in this example is specified by action *block* on target S_2, which can

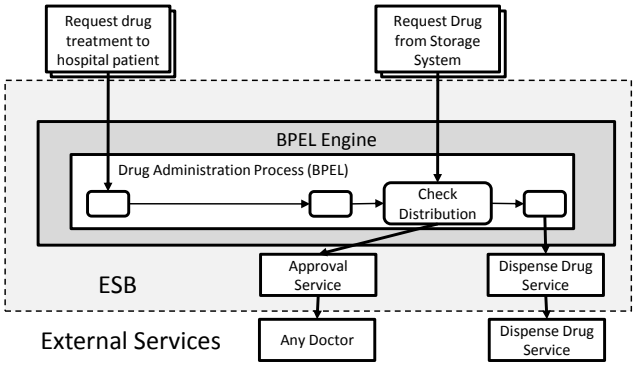


Figure 5.3: Original implementation of the drug administration process

also be mapped to the blocking of the corresponding *invoke* activity on BPEL level. However, if usage of S_1 and S_2 is not coordinated by a single business process, for example when drug prescription and drug distribution are parts of different business processes, then additional correlation of the service invocations is required. In this case the ESB can carry out the necessary message correlation to detect a critical state and block messages to the service S_2.

In general, when the policy controls the usage of services S_1, ..., S_n, both ESB and BPEL can be used for its enforcement. In this case, the decision of which to use must be made by the designer based on the current system implementation and with the goal to minimise distribution of the policy enforcement. For example, when these services are invoked from multiple points but all invocations flow through one ESB, this ESB can intercept requests from different processes and services at one centralised point. A BPEL engine in this case would need an extra policy for each BPEL engine that runs processes that invoke these services, and thus would have multiple enforcement points. Similarly, if a set of services that need to be observed are invoked from a single business process, then their usage can be controlled at one centralised point on the BPEL engine level.

5.5.2 Combination of Enforcement Capabilities

xESB complements the process engine enforcement because it performs complete mediation, endpoint resolution and leverages communication disparities. These disparities are often unavoidable because providers do not reveal the implementation of their services. To the usually stateless controls offered by the ESB, the BPEL engine provides a central point of coordination and global state management. xESB capabilities

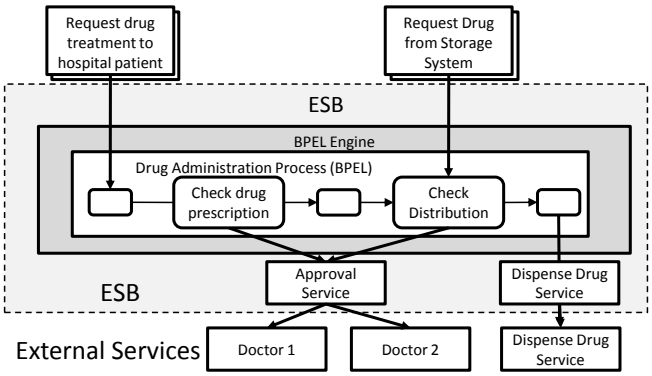


Figure 5.4: Modified implementation of the drug administration process. Also shows ESB responsible for choosing the doctor to approve the requests.

fit better with a cross-domain stateless enforcement, while the process engine better serves enforcing complex actions to achieve adherence to a common policy.

As described in sections 5.4.1 and 5.4.2, we have extended an ESB and a BPEL engine with added functionality for enforcement. Both components have been extended independently to meet enforcement requirements in their respective domain. In order to make best use of enforcement functionalities of both ESBs and BPEL engines we need to combine them. In the following we show how the combination of both enforcement capabilities works in a use case scenario.

Let us assume the hospital in our example has a BPEL process for managing the drug administration process shown in Figure 5.3. This process is out-dated: it does not implement Policy 1 as stated in Section 5.2 because it only has one check activity instead of the required two. There is also no policy deployed in the ESB ensuring the adherence to Policy 1. The activity labelled *Check Distribution* is responsible to check if a drug sample can be dispensed as is. The ESB forwards any approval request to any doctor being currently in charge for approvals of prescriptions and dispensations.

Figure 5.4 shows the newly implemented drug administration process as per Policy 1. To implement Policy 1, a new activity labelled *Check drug prescription* has been inserted into the process model. Already running instances have been changed on the fly by using the activity insertion enforcement mechanism of the BPEL engine. The Approval service in the ESB has not been changed. No new endpoint has been introduced for the new check activity. Instead, a new enforcement policy has been added to xESB stating that two approval requests from the same process instance are routed to distinct doctors. Here we use the underlying and realistic assumption that every doctor is using the system from a physically different endpoint, so that the ESB

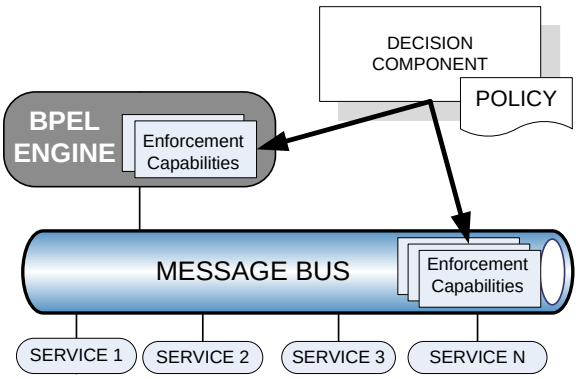


Figure 5.5: The architecture implementing the enforcement life cycle. The arrows to the right indicate that the decision component uses the interfaces provided by the ESB and BPEL engine for detection and reaction to violations.

can differentiate between doctors' device endpoints. When the same endpoint is operated by two different doctors for two different actions, the ESB cannot go beyond ESB endpoint information, which is normally not associated with user data. With external information, xESB can route to distinct doctor endpoints on the same machine.

5.5.3 Solution Architecture

Enforcement actions affecting the control flow of a business process are meant to be executed on a BPEL engine. It is possible to influence the control flow of a workflow from the outside by modifying input messages. To do this, deep knowledge of the workflow is needed in order to be able to do the right modifications to the input data. Therefore, we propose to execute enforcement actions that affect the control flow of a workflow directly on the BPEL engine. On the other hand enforcement actions that have to do with message routing or endpoint resolution of Web services are better to be executed on the ESB. We envision an architecture that joins the capabilities of the message bus with those of the BPEL engine, as shown in Figure 5.5. There can be one or more BPEL engines deployed to the bus to orchestrate the execution of a business process. The enforcement capabilities of these engines, as well as that of the ESB, are made available to a higher-level enforcement component that we call the *decision component*. As mentioned earlier, neither the message bus nor the BPEL engine offer any explicit support for enforcement decision making. Our decision component will discover the ESB and BPEL detection capabilities (as described in Section 5.5), and will make a decision of how to react to a possible violation of a policy. This reaction can be either at the BPEL, at the ESB level, or at both levels. Our example emphasises the case when the BPEL process is 'helped' by some ESB capabilities, but the opposite

is also possible, when a message-level violation can trigger a reaction that involves orchestrating several services to prevent, sandbox, or compensate for the detected violation. For the moment, the way to decide between these two cases is by checking an explicit flag in the policy, stating that its target it located at one level or the other. Thus, we rely on the policy writer to specify what part of the policy is linked to the ESB and what part to the BPEL instance. Keeping the decision component away from either the ESB or the BPEL engine makes the decision component independent of the limitations of either layer. In this way, the enforcement assurance of the composition will incorporate both types of message-level and BPEL engine capabilities.

5.6 Implementation Considerations

In this section we show how the prototypes of xESB and a BPEL engine implement the enforcement life cycle. This implementation has been carried out within the MASTER European project and has been showcased in a conference demonstration [42].

The implementation combines xESB with an instrumented BPEL engine. First, with xESB we hooked into service bus message interception, and message modification mechanisms. As mentioned in Chapter 4, ServiceMix offers an interface for intercepting all messages at the level of the Normalised Message Router. We employ this interface for *detection*: we direct all intercepted messages to an interface that belongs to the Decision Component shown in Figure 5.5. Before being directed to this component, messages can be filtered based on message metadata like message destination, message source, and user data. As *reaction*, we added a set of Web services to the message bus that can perform the following enforcement actions: blocking, modifying, delaying and inserting a message into the message flow. These trusted Web service endpoints can be invoked before or after a message is dispatched.

Second, the Apache ODE BPEL[1] engine was extended with enforcement capabilities to block and unblock a running BPEL process. The concepts of the prototype are based on two previous works: the first approach [120] proposes an architecture to extend ODE to emit all events occurring when a process is executed. The BPEL engine is enhanced with a component to extract the internal events (*detection*). The other approach [136] describes a management framework for BPEL based on Web service Resource Framework (WS-RF) [165]. Here, WS-RF was used to expose every BPEL process deployed on the Apache ODE BPEL engine as a resource (*detection*). This resource can be queried and changed. The changes are mapped back to the referred BPEL process (*reaction*).

[1] `http://ode.apache.org`

Web service interfaces link enforcement on the Apache ODE BPEL engine with enforcement on the modified message bus. All SOAP messages produced by the ODE BPEL engine and possible application services are routed through the ESB. The ESB can access the BPEL engine capabilities, and hence ask for orchestration of enforcement actions or events occurring at the BPEL level. Conversely, the BPEL engine will access ESB messages and ask to perform modification actions at message-level.

The Decision Component shown in Figure 5.5 is for now manual, in the sense that it waits for human intervention. The administrator can select what actions to perform from the set of all possible reaction ways described by a policy.

How does this implementation enforce the policies in Section 5.2? The ESB detects all incoming requests to the *CheckDrugPrescription* and the *CheckDistribution* services. All incoming requests to the former are let to pass unaltered, but a copy of the *CheckDrugPrescription* requests is sent to the Decision Component every time. The Decision Component will record in a “doctors not to distribute drugs" list the doctor-id (policy 1) or IP/location data (policy 3) of the doctor to whom this request is addressed, as well as the drug sample id (for policy 1). All incoming requests to the *CheckDistribution* service are blocked by xESB waiting for an enforcement decision. For each of them, the Decision Component will check if the drug sample id is the same as in a previous *CheckDrugPrescription* request. If it is, the Decision Component will invoke an enforcement operation in order to send the same dispensation request again to a different doctor to approve it. With the underlying assumption that each doctor has a different IP address, this modification of IP addresses can be done by xESB. With a ‘was security processed’ flag set, the new request would not be again blocked by the ESB, and will be forwarded to a right doctor for approval. The decision of the Decision Component is for now performed based on a list from human administrator declaring the doctors who cannot invoke the same Web service operations. Policies 2, 4, and other policies can be enforced similarly.

5.7 Related Work

In SOA enforcement, there has been no approach, yet that covers the enforcement life cycle in its entirety: (1) observation of events, (2) decision of right countermeasures, and (3) execution of a reaction. Moser et al. introduce the VieDAME4BPEL framework for monitoring BPEL processes and enforcing the replacement of partner services [157]. An interception and adaption layer allows to intercept outbound messages, monitor messages, apply message transformations, and replace services. Monitoring inbound messages and state changes is not discussed. This is a basic re-

quirement to react to possible faulty behaviour of a BPEL process. Another approach supporting the dynamic change of workflows is $ADEPT_{flex}$ [198]. $ADEPT_{flex}$ defines a set of change operations which can be used to adapt the structure of a running workflow. Conducting automatic changes to the running workflow is not intended by the solution.

In [227] Tsai et al. propose an event driven framework to enforce policies in an SOA environment. In this work a BPEL engine is instrumented to emit events for certain state changes of a running BPEL process. These events are then aggregated by a so called global policy engine and forwarded to a local policy engine. This policy engine is responsible for executing the necessary enforcement action. There is no explanation on what enforcement actions are possible and how these actions are executed.

Research in SOA enforcement that considers message-level enforcement has a lot of generic models [221, 137, 86]. Neither of these solutions discusses how the enforcement at the message bus layer actually happens, so that it can be linked with the orchestration level. More relevant to our viewpoint are the message-level enforcement frameworks that are scoped to SOA governance [146, 151]. These approaches hint to the centralization of policy management, when policies pertain to different abstraction layers. None of the solutions above separate between message-level and BPEL engine-level enforcement capabilities. To our knowledge, our work is the first to propose combining SOA components in the stages of the enforcement life cycle.

5.8 Summary

This chapter introduced an analysis over the enforcement capabilities of BPEL and ESB, along with a new approach and model for ensuring stronger policy enforcement in SOA applications. We made the case that enforcement only done by a business process engine cannot inhibit lower-level communication, that can pose a security risk. Also, we showed that ESB-only enforcement cannot usually comprehend the message payload rather than the meta-data, and hence is not able to react to violations in a way that is more complex than just invoking a service, e.g., orchestrating several service invocations as a complex reaction to a violation.

Our approach extends and combines BPEL and ESB capabilities on the framework of a SOA policy model. It is an architecture by which BPEL enforcement can use ESB abilities for policy enforcement, and vice versa. This is done by exposing these capabilities via enforcement interfaces; reacting to violations across these two levels is done using the enforcement capabilities that are known from a central control point.

Chapter 6

Assessing the Implementation of Policy Enforcement

What does not destroy me, makes me stronger.
Friedrich Nietzsche, *Twilight of the Idols*, 1888

Measurement is critical to software security. Only by quantizing our approach and its impact can we answer questions such as: How secure is my software? Am I better off now than I was before? Am I making an impact on the problem? How can I estimate and transfer risk?
Gary McGraw, *Software Security: Building Security In*, 2006, Chapter 2, p.47

6.1 Introduction

For hospitals and other healthcare institutions, it is essential to know if their applications are compliant with existing healthcare IT security regulations. Checking compliance to regulations involves checking that the security policies derived from regulations are properly enforced. This need is especially stringent when healthcare applications are using software components from various providers. Even though these components can be certified, certification does not guarantee that the borrowed software has no unwanted side-effects and poses no security threats.

Up to now, compliance assessment has been the task of either an IT auditor (e.g., the hospital in our healthcare case study is subject to healthcare audits as described in Section 3.1), or of internal isolated analyses like a-posteriori analysis [58]. Auditing checks whether individual criteria are met in the overall application code. Some examples of such criteria can be found in the ISO/IEC 27001 standard, e.g., whether the access to sensitive information is controlled, whether the management is aware of risks due to security incidents, or whether the security requirements have been

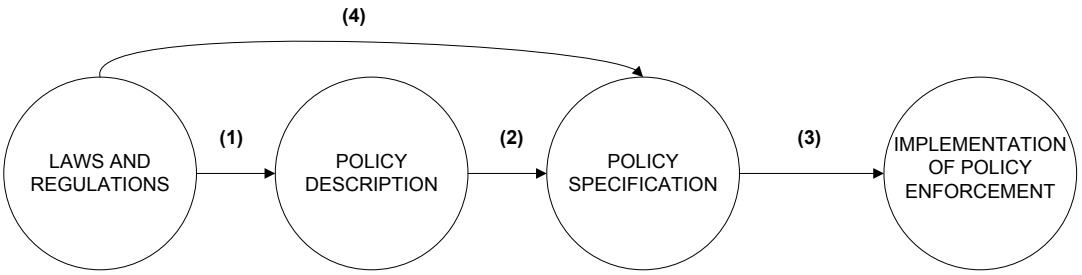


Figure 6.1: Steps in getting from laws and regulations, to policies to be enforced by the security system.

correctly identified. Such criteria are common for a range of programs; their presence is evaluated with a mere yes/no tick, and it is not impossible that an application that undergoes a successful audit actually has deeper security issues. The latter, a-posteriori analysis, is specific to a particular application; it investigates misbehaviours that have occurred before the program was stopped, but does not act on them other than detecting violations, identifying the perpetrators and making them accountable.

Waiting for the yearly audit to know if the application really enforces security regulations is not practical. This is because it is difficult to 1) identify all sources of faults detected by audit in the application, 2) treat them with minimal disruption, 3) verify that the treatment was effective, and 4) treat the treatment. Rather, the management would find it useful to know of problems sooner than once a year and several times for a long-running application. It is obvious that the sooner a security problem is discovered, the easier it is to treat it before it escalates. Also, having periodic compliance 'heartbeats' would help management see not only that *there are* security issues, but also if the alleviation strategies to correct those issues are effective. Hence, having a *runtime* assessment of how well the existing security infrastructure enforces a policy would not only ensure a successful audit, but demonstrate a safer application. For the clients, a safer application is a more trustworthy application.

6.1.1 Where to Look for Misbehaviours?

When is policy enforcement incorrect, and where should we look for it? To answer these questions we need to know how security constraints are born. Security constraints are the result of a refinement process shown in Figure 6.1. The diagram shows the common stages to get from a requirement of a law or regulation, to a policy that can be enforced by security components. These stages refine the security requirements from natural language (laws and regulations, and policy description),

may pass through an intermediate step of a policy specification in a policy language (either a formal language like process algebra, or less formal languages like XACML), and finally are enforced, with the help of enforcement mechanisms available in the system (the atomic mechanisms that can be used for security policy enforcement are called *security blocks*). How correct a policy is enforced depends on how correct the transitions in Figure 6.1 are implemented. These transitions can be incorrect because they are the responsibility of different people:

- usually, the *security officer* refines laws and regulations to a policy description,
- a *policy creator* derives the policy specification from the policy description, and
- a *security developer* implements the software that enforces a policy created by a policy creator.

Hence, problems can occur from two main directions:

Wrong/incomplete transition from natural language, to policy specification. First, security constraints can be misunderstood by the security officer. Everybody who has had to understand legal regulations knows the difficulties of law and regulation parlance. Then, even if the constraints are understood properly, the security policy to be enforced might be incomplete or inadequately specified. For instance, if patient data is to be anonymised in all flows of a healthcare application, then these communication flows should include database operation requests *and* flows to external entities, and they should be expressed in the policy language, even though in natural language they are implicit.

Wrong transition from specification to the enforcement of a policy. Even if the security policy is properly specified, enforcing it can incur errors or inaccuracies. For instance, when the user session does not expire after 10 minutes but instead after 10 hours, or when a business process simply 'forgets' to authenticate a user. These issues can happen when the security developer makes mistakes, either on purpose or accidentally.

The correctness of a complete transition chain composed of a step that derives a policy description from law text (labeled *(1)*), a formal policy specification out of a policy description (labeled *(2)*) and finally enforces a policy given in a formal specification (labeled *(3)*), has been studied within the European project MASTER[1]. Other researchers have focused on the correctness of the pass from regulations to policy specification (labeled *(4)*), either specifically for some healthcare regulations [50] or

[1]Managing Assurance, Security and Trust for Web Services (MASTER) - FP-7 Project `http://www.master-fp7.eu/`

legal business rules [223]. There is also work on the correctness of the transition from a policy specification in a formal language, to the policy enforcement stage (step *(3)*) [29]. While we are aware of these formal works and their importance in the process of auditing, our approach lies on the side of practical assessment *at runtime*, when there is no prior formal specification of which a machine-readable policy is derived. We assume we have a complete and correct policy specification in a non-formal policy language, and we want to assess how well a set of practical security mechanisms enforces a given security policy. Apart from its obvious usefulness, another motivation for this approach is that up to now nobody has checked if the security features actually implemented respect the desired security properties, be them practical or theoretical. This assessment is especially needed when no formal policy specification exists on which to evaluate the theoretical properties of a correct policy enforcement mechanism: Bauer et al. [140]'s soundness ("all bad events are eventually made good"), transparency ("compliant events pass through unchanged"), preciseness (both sound and transparent), etc.

6.1.2 Contribution

To our knowledge, there are no practical solutions to assess how well a policy specification has been enforced with one or more security blocks, when the program is running. As of now, there is just policy-agnostic testing of individual components in isolation, and that is done mostly in-house for a company's own application components. Runtime assessment would be useful for the management or stakeholders of the healthcare application, so that they know what management strategy to follow in order to adapt to changing risk. Also, enforcement assessment at runtime would be useful to security administrators to react timely and adequately to misbehaviours (that have occurred or are about to occur). This can be done without stopping the program, hence without financial losses.

With this motivation, we want to build a mechanism to assess how well a running security control enforces a security policy. To do so, it is easier to start by checking the effects rather than investigating the causes of misbehaviour. Also, it is easier to label an effect as undesirable than to write policies that prevent it from happening. Such unwanted effects are policy violations, even if that policy exists only implicitly and has never been explicitly written down. Having a quantitative indication about the damage and impact of such violations can help a security administrator from two points of view: first, a quantification of system state can be compared to other states in the past; second, it would give a hint on where to intervene to improve existing enforcement mechanisms.

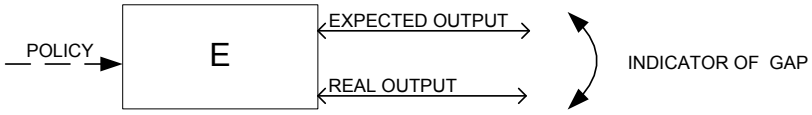


Figure 6.2: The gap between the expected behaviour of an enforcer E, and its actual behaviour.

This chapter presents an approach and solution to assess how correctly policy enforcement mechanisms work. We introduce the novel notion of *enforcement indicators*. Enforcement indicators are a specialised type of security metrics that, for a given application and policy, show the deviation between the way the policy is enacted, and the policy's desired effect (see Figure 6.2). A second novelty in our approach is the feedback system for enforcement, that correlates enforcement indicators with *enforcement corrections*. These are actions performed by the system that aim to compensate an abnormal enforcement indicator value. In other words, whenever the value of an indicator drops below or rises above a predefined value specific to the application, an appropriate correction is performed so as to compensate the deviation measured by the indicator. Apart from presenting a model of how to incorporate indicators into existing controls, this chapter presents a first prototype of indicators for SOA enforcement, that we show to be able to express any computable indicator expression.

6.2 Requirements from the Case Study

In our example, the hospital keeps data records of all its patients, and is required to periodically report some of the data to the Ministry of Health for both statistical and reimbursement reasons. The Ministry of Health requires several categories of data – doctors' activity in terms of patient cases that were handled, surgery data, hospitalised cases, drug administered etc. Such reports are produced in the hospital application by several different IT services, and they all relate to patient data. In the light of privacy regulations like [63], HIPAA-1 and IDPC1, the reports must anonymise all sensitive patient data, but at the same time, some patient data must remain in the record so as an audit of the hospital system could track individual activities related to patients without their private data being disclosed. In order to have a flexible way of implementing anonymisation services for the variety of IT services, and to connect easily with its contractors, it was decided that the hospital infrastructure would use a message bus (as in [13]) and implement the system in a phased manner.

In year one, the hospital decided to implement the anonymisation process internally. The security developers plan to develop their own privacy enforcing process such that all data that is expected to exit the hospital system, from different report-

ing services, will go through the same point where the sensitive parts are inspected and filtered out. For a yearly compliance audit on data security, the hospital needs to know how well their patient data anonymisation works and whether any sensitive information has been leaked.

In year two, the hospital signs a contract with a tele-assistance company whose purpose is to offer medical assistance by phone to outpatients. This tele-assistance company would also connect to the hospital infrastructure using the message bus mentioned earlier. The staff contacting patients by phone need to have partial access to patient records (possibly full patient records in special cases, e.g., when recommending a MRI scan, previous reconstructive surgery data will be needed to know of metal parts in the patient's body). For reasons of patient privacy, the tele-assistance staff should not know full details about doctors or the surgeries that the respective patient underwent. The tele-assistance company uses an ePrescription software by which tele-assistance physicians send prescriptions to the patient's email address instantly. The patient cases handled remotely in this manner still need to be reported back to the hospital so that it can assess the correctness of the data that reached the tele-assistance employee, the prescription details and the data reaching the Ministry. For its second year audit, the hospital needs to know how correctly the tele-assistance company reported the data to the Ministry of Health when either a hospital-implemented anynomization service is used, or when an internal anonymisation service is used by the tele-assistance company to send the data to the Ministry. Since some tele-assistance services use the hospital infrastructure, the hospital can check which data flows to the Ministry of Health, and take action if any privacy violation is detected.

To go more in depth, we consider a policy that states that healthcare communication to a certain endpoint E must have the patient name field anonymised. This policy is expressed in first-order logic with strings and equivalence relations like this:

$$\forall m \in M : dest(m, E) \Rightarrow (pid(m) = "xxxxx")) \tag{6.1}$$

The relation above says that for all messages m in a set M of messages detected by a security mechanism, if the destination of m is endpoint E, then the field that identifies the patient information $pid(m)$ in the message will have its content replaced with the string $xxxxx$. To enforce this policy, we assume that the hospital's security infrastructure is able to inspect and control all information flows among all endpoints (also to a certain service endpoint called E). The second assumption is that anonymisation means replacing the patient name data field by a fixed amount of characters (e.g., five x's). It follows that the implementation of an enforcement mechanism for such a policy can deviate in several ways:

1. some messages are anonymised correctly, others are not at all anonymised and pass unchanged;
2. some messages are anonymised correctly, others are anonymised incorrectly e.g., just 3 characters are replaced by x's;
3. some messages are anonymised incorrectly, others are not anonymised at all;
4. some messages are anonymised correctly, but for others, faults or errors occur (e.g., SOAP faults that can disrupt the flow of message processing) that deflect the data to other endpoints, hence the faults do not contain anonymised data;
5. frequent faults or errors, with no messages being output at all;
6. incorrect anonymisation for every message;
7. for every received message, more than one message is sent out (e.g., extra traffic generated by duplicating the inbound communication to a malicious endpoint); there are two sub-cases here: the extra messages are anonymised correctly when they should be, or they are not;
8. all messages pass unchanged, with no anonymisation;
9. none of the messages are anonymised to the fixed length field.

Depending on the application, such policy deviations can impact the system in different ways. If it is essential that some anonymisation happens indeed – or in other words, if the security mechanisms that enforce this policy, as an ensemble, provide soundness. Hence cases 1, 3, 4, 5, and 8 are critical, as the sensitive data is altered in some way before reaching endpoint E; alternatively, cases 2, 6, 7 and 9 are misbehaviours that are less critical than the others. Either way, cases 1 to 9 listed above can be ordered by severity in different ways depending on the application policy requirements.

6.3 Assumptions

In order to build an assessment framework, we consider that the hospital manages a service-oriented application that uses middleware technologies such as the ESB to integrate their partners' software components. With networks like NHIN this has become a reality and the advantages of this choice are:

1. *Ease of SOA integration*. As described in Chapter 2, the ESB is a state-of-the-art technology used in industry for integrating SOA applications.

2. *Cross-service, cross-domain policies*. Policies like the anonymisation requirement presented above spread across the hospital domain, the tele-assistance domain, the government domain. Such policies constrain communication among services (e.g., no more than one report request from any governmental organisation service per day), or the data transferred between services and domains (e.g., anonymisation of sensitive information, or service request history).

3. *Cross-service controls are needed to monitor adherence to cross-service policies*. If such policies would be enforced on every machine to host a hospital service, the effort to deploy the necessary enforcement mechanisms, as well as to acquire the data, would depend on individual platforms and operating systems. Also, it would make the management of such controls very difficult if the policies change.

We assume that in order to enforce security policies across multiple services (that make up one enterprise application), there is a set of policy enforcement mechanisms of which some are at message-level. Previously, Chapter 5 has presented how BPEL controls are not capable of monitoring lower-than-BPEL communication flows, and that these flows can actually be relevant for security policy enforcement. Of course, the efficacy of message-level mechanisms depends on how much of the actual messages they can process. Encryption can impede such processing, since in our scenario it protects against the unauthorised release of medical data. Yet in our scenario the hospital is simultaneously an actor and an infrastructure provider; this means that to a certain extent it should have control over the data of its patients, and also over how that data is released. For this reason, we consider in this thesis that the hospital can decrypt message payloads, and we do not look into the related cryptographic details.

We also assume that security policies are specified by a *policy creator*. The *security developer* enforces security policies by using some pre-existing enforcement blocks. Both creator and developer are familiar with the semantics of the security policies, but not also with the implementation details of the enforcing mechanisms.

Another assumption is that logging is done across the entire distributed application. This is a given in today's distributed world, where logs are kept at all times, for reasons of accountability and disambiguation, as done in CODA [205]. These logs can be collected either on each endpoint or can be centralised for all component endpoints together. The inputs and outputs of the enforcement controls are always recorded in current systems, so it is natural to assume that these records contain all traces needed to ascertain system or user (mis)behaviour.

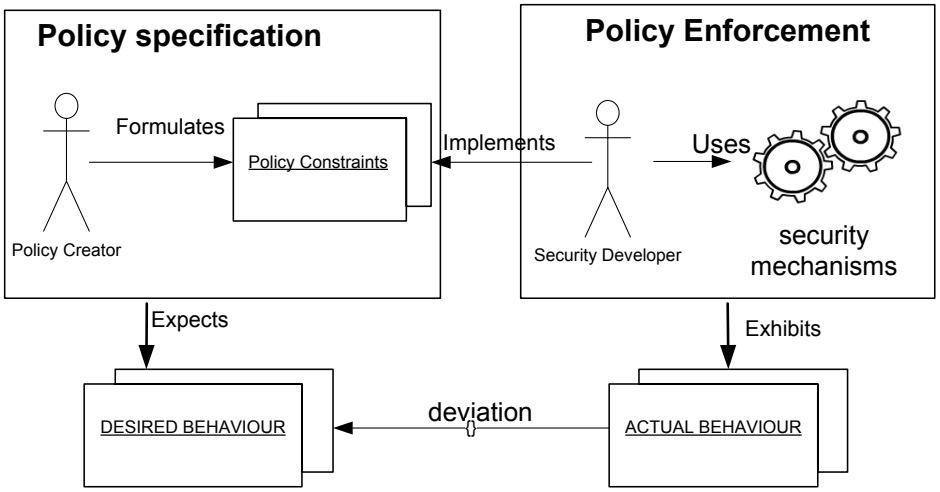


Figure 6.3: The enforcement process as a result of using enforcement mechanisms to implement constraints. There might be a deviation between the behaviour wanted in the specification and the behaviour allowed by the enforcement process.

6.4 Approach to Assess Policy Enforcement

In this chapter we concentrate on how to assess policy compliance rather than on how to actually enact it. Our focus is to measure the difference between the *desired* behaviour as expressed by the policy creator, and the *actual* behaviour of the enforcement mechanisms, that were put together and configured by the security developer. This difference, or *deviation*, is shown in Figure 6.3. The actual behaviour of the enforcement mechanisms of the policy enforcement implemented by the security developer; as explained before, the deviation between expected and actual behaviour can occur from reasons such as the absence of proper enforcement mechanisms to implement the wanted behaviour, or the developer putting together the enforcement mechanisms in a wrong way. For enforcement indicators to be meaningful to the application administrator, the next subsection details the requirements for enforcement indicators at the SOA message-level.

6.4.1 Requirements of an Enforcement Indicator

As enforcement indicator is a specialised software metric. In Web and business analysis, the term “indicator" is seen by experts as a metric with a meaning to the management [2]. In this dissertation, we want to take this term into security research. Yet, in a reality full of metrics, our enforcement indicator needs to be both *meaningful* and

[2]For example, Anil Batra and Dennis R. Mortensen describe some characteristics of key performance indicators on their blogs and slides at `http://optimizationtoday.com/web-analytics/online-kpis-back-to-basics/`.

concrete. In opposition with the rather conceptual and vague metrics in IT security, we want to actually model, analyse and implement a security indicator in practice.

To build such an indicator, we need to know what its properties are and what they depend on. We start off from the two main aspects: measure and meaningfulness. First, since this indicator is a software measurement, it should have two main properties of any measurement [128] – *accuracy* and *precision*. Second, for an indicator to be meaningful it has to aggregate data gathered on a set of criteria so that it can present some meaning to a higher abstraction level. It follows that it should have a more SOA-specific property – *granularity*. These first three properties are explained below:

Accuracy. In general, accuracy is defined as:

> "The accuracy of a measurement is the degree to which a measurement reflects reality" [128, p.30].

In this sense, we want an enforcement indicator to correctly identify all real deviations, and ideally also to correctly identify their features (e.g., types of faults and their context). For instance, anonymisation constraints only apply to data that must be anomymised in the first place; an indicator that takes into consideration also the messages that are to destined to endpoint E and/or do not have any field with patient data, is not accurate. Accuracy should be balanced with performance requirements, since our enforcement assessment approach works at runtime. Accuracy depends on the amount of data analysed at a time, the deviation description, and the frequency of the captured deviations (the more frequent deviations, the less time to analyse each of them accurately).

Precision. In its turn, precision refers to the variation among different measurements of the same property. Precision is a property of how to compute the indicators, rather than how to define them. Common sources of variation include external factors influencing the measurement, and the existence of multiple unsynchronised entities that perform the same measurement. Intuitively, an indicator computed by a sole entity, at a sole location, is most likely precise, when the external factors influencing the computation (e.g., load on the host machine) are kept to a minimum.

Granularity. Granularity generally refers to the number of distinct atomic parts of a composite entity (e.g., the granularity of a Web service refers to the operations, or methods, that it provides). For us, a security indicator is a composite measure and hence its granularity refers to the number of distinct atomic measurements that have to be done to compute it at one time. *Granularity* belongs to the defini-

tion of an indicator, based on defining one or more atomic misbehaviour events. An enforcement indicator is fine-grained if it maps one-to-one certain deviation type (e.g., looks just for case 9 of Section 6.2); conversely, an indicator is coarsely-grained if it aggregates several categories of misbehaviours (e.g., cases 1, 2, 3, 9 together, or an aggregation of both anonymisation and encryption deviations). There is a tradeoff between the granularity of an indicator, and the effort to define it down to its atomic measurement criteria.

Precision, from the features above, comes with the architecture described in Section 6.5, and the from the way to perform measurement. Accuracy comes with the method used to correctly identify relevant events, in our case misbehaviours. But what are misbehaviours? In order to derive what misbehaviours are, we need to define the patterns of normal, or legitimate, behaviours.

To define normal vs. abnormal mechanism behaviour, we employ a cause-effect relation model [228]. From this perspective, an enforcing mechanism is a 'grey box' whose behaviour consists of a trace of output events (or *effects*) and a trace of initial events that triggered the outputs (*causes*). Correlating the inputs and their corresponding outputs is a hard problem in a black-box model, that is, for a system in which nothing is known about the event flow among its component (security) mechanisms. We assume we have some information about the chaining of what we call *security blocks* – the component mechanism of the security enforcing system. Hence, if we know what outputs correspond to what inputs – because we would have, for instance, a partial ordering relation such as time and a unique identification of each event – we can conduct a complex analysis in order to determine if the actual behaviour deviates from the expected behaviour (given a policy specification to be enforced).

The 'grey box' model implies reasoning about the correlation about the 'right' effects for a given cause, and about the number of causes related to the number of effects. In more detail – an enforcement mechanism that functions correctly should produce all necessary effects for a given cause. For instance, every time the enforcer receives a message that contains patient data, the corresponding log will contain a record of that message having the patient data replaced with zeros or x-s. In addition, a correct enforcement mechanism should produce *only* desired effects in the presence of triggers. For instance, the only effect of receiving a message with patient data, is the same message with the patient data anonymised (and a log entry). Other disallowed effects can be calling other services or performing operations not required by the policy (e.g., case 7 in the example section). On a different note, we want our grey-box to be *robust* – that is, a good enforcer mechanism to cope with errors that appear at runtime due to malicious/malformed input. From this point of view, if a

causing event triggers an error, then the policy should be somehow enforced on the data included in the fault. For instance, patient data included in the payload of a message that generated a SOAP fault should not be released as is, but anonymised.

For message-level policies, i.e., those that impose constraints on messages exchanged across the application infrastructure, the properties above can be further refined into desired or undesired enforcement behaviour. These patterns can be observed from the logged communication across services. Hence, for a message-level mechanism, depending on the correlation between causes and effects, their distribution, and their cardinality, we identify several properties of message sequences that can constitute disallowed policy behaviour. These properties are bound to an observation window, called *scope*. A later literature review revealed that they have similarly been proposed at the service orchestration level [1]:

(R1) absence/existence over a scope means a message/flow must not trigger a message/flow over that scope. For forbidden SOAP faults, this maps to case 4.

(R2) universality over a scope means a message/flow must happen for every time frame over the specified scope. We are considering time frames in order to be able to perform an online analysis. This maps to cases 9 and 6 above.

(R3) bounded existence over a scope means a message/flow must trigger the wanted message/flow for a bounded number of times over the specified scope.

(R4) precedence over a scope means a message/flow must trigger a message/flow only if preceded by another event over the specified scope. In our anonymisation example, there cannot be a log entry without a sensitive message.

(R5) response over a scope means that a message/flow must trigger a message/flow that will have to be followed by another one over the specified scope.

(R6) chain precedence over a scope means a message/flow must trigger a message only if preceded by a certain chain of other messages/flows over the specified scope.

(R7) chain response over a scope means a message/flow must trigger an event that will have to be followed by a certain chain of other messages/flows over the specified scope.

When multiple policies are enforced, it may be that seemingly correct enforcement effects for one policy, are in fact triggered by other policies. Overlapping enforcement events of multiple policies is outside the scope of this discussion; we consider that as far as the correct effect is triggered by a cause, the enforcement is correct.

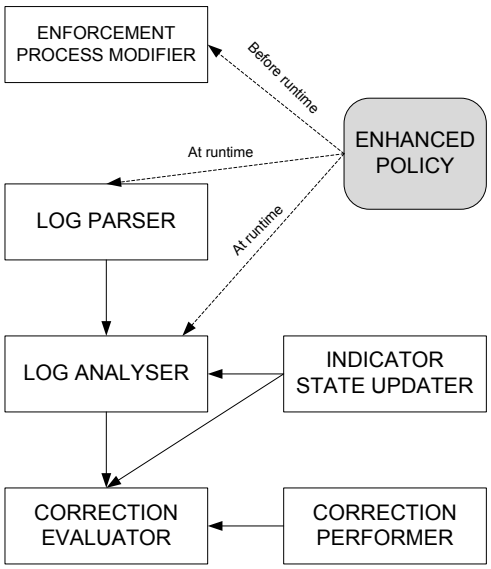


Figure 6.4: The architecture of our framework.

6.5 Design of an Indicator Framework

We have designed the assessing of how correct the enforcement of a policy is, by means of a process that involves several steps:

1. specifying violations and associated indicators in connection with the policy;
2. instrumenting the enforcement implementation to obtain the traces that capture the relevant violations;
3. calculating the indicator value;
4. performing the corrections as a system response to surpassing an application-dependent threshold value for the indicator.

The architecture of our solution is given in Figure 6.4: at first there must be a policy that has been enhanced with extra data about how to compute and use the indicators. Also, the security blocks used for the enforcement of that policy should be modified before runtime in such a way as to produce frequent log traces; at runtime, the Log Parser and the Log Analyser components use the information about what violations to look for and how to compute enforcement indicators. The task of the Log Parser is to scan the execution logs in search of violations; the Log Analyser matches the violations specified in the enhanced policy with those filtered by the Log Parser, computes the indicator based on a mathematical formula specified in the enhanced policy, and sends updates to the Indicator State Updater. The Log Analyser is connected with a

Correction Evaluator, that, based on the type of the current violation and the deviation of the overall indicator with respect to its threshold, decides what correction should be performed, and delegates it to a Correction Performer module. The Performer can be a Web service that can perform the correction decided by the Correction Evaluator.

6.5.1 Specifying Indicators

An indicator is very closely linked with a policy: the indicator assesses how well the enforcement mechanisms used to enforce that policy achieve this purpose. It is realistic to consider that policy creators have sufficient knowledge of the application constraints so that they can specify the essential unwanted behaviour. For this reason, the specification of the indicator can be included in the policy file. This inclusion can happen as the policy is written, or can be afterwards merged with an already written policy, as long as there is a component to extract the indicator data and interpret it.

The elements of an enforcement indicator specification are shown in Listing 6.1. In itself, the indicator is a mathematical expression over a set of parameters that are policy deviations; the weights (or coefficients) of the parameters are included in the expression. Whether this expression is a linear combination of its parameters, or constitutes a more complicated mathematical relation, is an aspect that depends on the application. An example can be a weighted sum, as follows:

$$I = \sum_{i=1}^{n} w_i v_i \ , \qquad w_i \in [0..1] \tag{6.2}$$

The indicator expression considers a number of violation occurrences (in the formula above, there are n violations defined); each violation has a type and occurrence parameters (e.g., an event that occurs all the time, sometimes, or never over a scope), and these occurrence parameters are quantified with a weight. In formula above, the occurrence count v_i is multiplied with a weight w_i between 0 and 1, and the indicator is a weighted sum of these elements. The formula above is just an example, and the expression can be anything from a fraction, to an integral. The assumption is that the policy creator or a security expert can, for the given application, express this indicator formula as a mathematical function of a set of violations. We are aware that similar indicator expressions, albeit with a completely different purpose, are already possible with Key Performance Indicators (KPIs) in tools such as WebSphere Business Monitor[3] or Symantec's IT Analytics Solution[4].

The enforcement assessment system can compute violation occurrences at runtime,

[3]WebSphere Business Monitor, `http://www-01.ibm.com/software/integration/business-monitor/`.
[4]Symantec IT Analytics Solution, `http://www.symantec.com/docs/HOWTO48370`

```
<Indicator>
  <expr>mathematical expression here</expr>
  <parameters>
    <violation>
      <violationType/>
      <violationParameters/>
    </violation>
  </parameters>
 <threshold>xs:double value here</threshold>
 <correctionType>xs:integer value here</correctionType>
</Indicator>
```

Listing 6.1: Generic XML specification for an indicator.

and with the indication parameters thus instantiated, can check if the indicator value exceeds a threshold; when it does, a handler of a certain correction type is invoked.

6.5.2 Log Analysis

Once the system has been instrumented to log messages that are relevant for computing indicators (hence to gather the deviations as soon as they happen), we need to process the log records. We choose to do the analysis in real-time, as the system is running, in order to infer runtime indicator values and to use that advantage to react to the potential deviation in time.

As it would be inefficient to analyse the logs every time a new log entry is added, we use a sliding-window technique. In it a parameter t is defined such that only events that happened later than this t are considered for analysis. In effect, a time window of duration t is slid over the log entries, and the window is advanced with the arrival of each new log entry.

It is also possible to save state from one computation of the indicator to the next. In other words, when we compute the new value of some indicator, we can have access to the old value. This allows in effect the computation of indicators that depend on events outside of the sliding window, and in fact arbitrarily far back in time, should that be desired. Therefore the imposition of a sliding window is not a practical obstacle to computing indicators.

When events are logged at different points in the system, for example centrally in the message-level middleware, or at the service endpoints, it is possible that log entries are produced out of causal order, causing an effect to appear in some logs earlier than its cause. This must obviously be taken into account when computing indicators. Equal care must be taken when analyzing log files that have been produced on different computers with different local clocks that may have changing offsets and

drift. Synchronizing the machines involved using protocols such as NTP[5] should reduce the occurrence of this issue.

The next problem to tackle is to find those events that constitute (or are part of) a policy violation. As we have argued, it is easier to say that a certain outcome is undesired than it is to specify the system so that these undesirable outcomes are impossible by design: perfect accuracy, that is, unambiguous identification of policy-violating events, can be attained only with a correct formal specification of the desired behaviour of the system, and what log event sequences it can generate. Since we assume that we do not possess specifications of the analysed processes, we have no choice but to use methods that will necessarily produce false positives (events classified as policy-violating when they are in fact not) and false negatives (events classified as not violating a policy when in fact they are).

In our approach, we use techniques from full-text searching. The advantage of this is that we get a measure of the distance between the pattern and the match, which supports us in making a decision on how heavily to weigh the match when updating an indicator, or whether to consider the match at all. The disadvantage is of course that there will be false positives and false negatives mentioned above.

6.5.3 Instrumenting the Enforcement Process

Out-of-the-box policy enforcing mechanisms – or enforcement blocks – can range from mechanisms that check sender details for sender authentication, to mechanisms that examine message validity, to mechanisms that analyse the message payload for anonymisation. We suppose we have no knowledge of the internals of these mechanisms, so we can infer little about what is the real cause of a policy violation. In the anonymisation example, a message can be deemed as a violation when its payload was not anonymised, or when the message was malformed, or when the message sender could not be authenticated. With multiple causes for an effect, it would make a difference if we know some details about the chaining of enforcement mechanisms. For instance, we could know beforehand that there are three enforcement blocks: one for message validation, the other for authentication the message sender, and the other for the message anonymisation. These mechanisms, arranged in some order, act on independent features of a message. Knowing the in-s and out-s of (some of) these chained mechanisms allows to point more accurately to that mechanism where the violation occurred. This approach is novel, and is a far better alternative to security testing because of several reasons:

[5] `http://www.ntp.org/`

1. it is an assessment in the real business context (in vivo), while security testing is done 'in vitro';
2. it checks the occurrence of events that are damaging to the application overall, rather than standard generic security tests;
3. since it is done as the system is running, it can be used to check the system evolution, hence it is *meaningful* to the management; security testing is not;
4. with some information about the chaining of security enforcement blocks, it can help detect 'areas' that are sources of violations; security testing is usually 'blindfolded' to this respect.

A possible reaction to a violation is to enable extra logging at the location range where the violation was caught. Specific branches or sequences of mechanisms in the implementation of the policy enforcement can thus be surrounded by instructions that produce finer-grained log entries about incoming and outgoing events (in this case, messages). This added information could refer to message payload (to check the incoming payload against the outgoing one), message metadata (to check where messages are directed to), or message payload hash (to check if the payload has been modified at all without knowing its contents). Apart from supplying finer-grained information about events inside the enforcement process without knowing its exact internals, this approach has the advantage of imposing the causal order over the logged events. That is, if mechanism M_1 comes before mechanism M_2 in the enforcement implementation, log entries of M_1 will always come before those of M_2, hence when the logging is done in one file, the output of M_2 will always be logged after its cause (the input to M_1).

6.5.4 From Indicators to Corrections

After the relevant deviations have been identified, the Indicator State Updater shown in Figure 6.4 computes a new value of the specified indicator, and compares it with the stated application-dependent threshold value. Corrective actions must come into play when the value of the indicator goes below or above the threshold. In the anonymisation example, for instance, a drop below 10% unanonymised messages (from a desired value of 100%) can mean that the mechanisms that perform anonymisation (rather than those that authenticate message sources or check message legitimacy) are not working correctly. Possible corrections, in this case, could be alerting the system administrator (usually the default is the cause of the violation is not known) or replacing the faulty mechanism with another one (or redirecting all next messages to a

trusted processor), among others.

An interesting associated problem is how to decide what is causing the indicator fluctuations, based on the indicator value and the events logged. This problem relies on *correlating* several elements:

- indicator value and most recent events: a sudden change in the indicator value, compared to values in the past, can reveal possibly severe deviations to have happened recently or earlier in time. If they are individual messages, they are easier to pinpoint than message flows that may have started long in the past;
- indicator fluctuations in time: can reveal if certain deviations happen repetitively at application runtime; with that clue, an administrator can investigate the causes of such deviations;
- indicator fluctuations and corrections: can reveal if the performed corrective actions are beneficent, i.e. have the desired effect over the system runtime.

Going more in depth on these directions can be of tremendous importance for discovering the causes of misbehaviours of security mechanisms. However, since they go deep in the process of root cause analysis, they lay beyond the scope of this thesis.

6.6 Implementation

From the framework described above, we have implemented the indicators components, and we have investigated how to implement corrections. We used an out-of-the-box middleware tool – Vordel XML gateway[6]. This is an XML gateway for SOA that offers processing of messages on the enterprise infrastructure by means of message-level policies. Such policies can be constructed by chaining prebuilt mechanisms in the desired process flow; the configuration of such mechanisms can be adjusted by the policy enforcement developer. For instance, Figure 6.5 shows the concrete Vordel implementation of the anonymisation policy, as seen by the developer. The implementation consists of several types of prebuilt mechanisms that have been configured to implement together an anonymisation policy:

- the "Look for" nodes of this process are called filters i.e. decision blocks that examine an incoming XML message on the gateway and decide whether it matches the criterion of the filter or not (in this case, the presence of a node called $sam : c$ or $sam : a$);

[6] http://www.vordel.com/

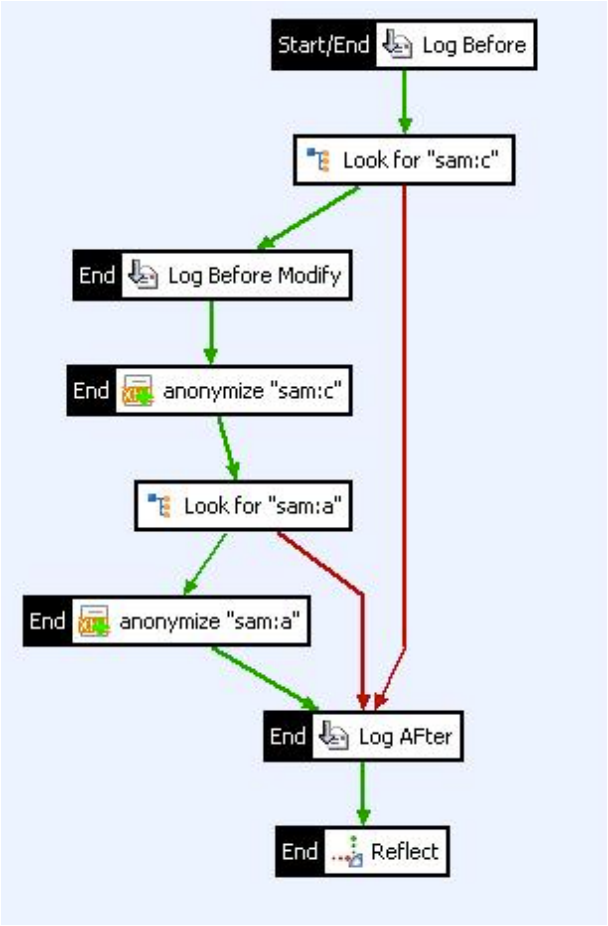


Figure 6.5: Screenshot in the Vordel policy editor of the implementation of a policy that anonymises XML nodes of an incoming message. There are four kinds of enforcement blocks: three identical blocks that just record an entry in the Vordel log – “Log Before", “Log Before Modify" and “Log After"; two “LookFor" blocks that check the existence of an XML element in the message; two “anonymise" blocks that replace the value of a node with another value; the “reflect" block is a default block that marks the end of the gateway processing. The red arrows (directly between “Look for sam:c" and “Log After", and “Look for sam:a" and “Log After" respectively) indicate the event flow if the condition evaluates to false, the green arrows (all other arrows) indicate the flow on the true branch.

```
<AuditEntry timeStamp="1275583457352"
  messageID="Id-00000128feb2e848-00000000002b452b-1520"
  level="Success">
  <clientIPAddr>127.0.0.1</clientIPAddr>
  <filterName>Log Before</filterName>
  <filterType>LogMessagePayloadFilter</filterType>
  <logText>'LogBefore' logged msg payload</logText>
</AuditEntry>
<payload>
  <![CDATA[
   --- HTTP method data here
   --- soap envelope here ]]>
</payload>
```

Listing 6.2: Log entry envelope in the log file of the Vordel gateway.

- the anonymise blocks are message processors that internally replace a part of an XML message, with a given one (in this case, we replaced the value of the *sam* : *a* or *sam* : *c* elements, with "xxxx");
- the log blocks write their incoming messages to a Vordel log file.

Since the condition that a filter checks against a message is boolean, a filter can have a green branch (condition evaluates to true) and a red branch (condition evaluates to false). In this way, all incoming traffic that the gateway intercepts passes sequentially through this box-and-arrows policy. Because of this graphical policy language, as well as the existence of pre-built mechanisms for a relatively wide range of security functions (authentication, message processing, etc.), we chose the Vordel message gateway for our prototype in order to generate logs on implemented policies.

6.6.1 Indicator Prototype

We generated the log records of the anonymisation process shown in Figure 6.5 by setting up a custom Web service, and sending SOAP traffic to its endpoint. This traffic would be diverse so that for each message intercepted by the gateway, the policy would apply differently (e.g., messages with or without the *sam* : *c*, or *sam* : *a* nodes).

The gateway logs events using its own XML schema: a log record contains log event metadata and log event payload, which is in fact the transferred SOAP message. The log event metadata (hereby called envelope) contains information about the block that generated the message, the success of the operation, timestamp and message sender and receiver endpoints, as shown in Listing 6.2. In order to process the gateway logs, we opted for the XQuery query language[7]. It was a natural choice given that

[7] http://www.w3.org/TR/xquery/

the logs are kept in XML format, and also that XQuery is known to be an efficient query language on such data. XQuery is a query language on XML data; it is similar to MySQL on databases, and has a FLWOR syntax: it processes the input based on **f**or, **l**et, **w**here, **o**rder and **r**eturn operations. Also, in order to identify the wanted violation patterns from the log records, we chose the Apache Lucene[8] API for string matching. A provider of the features of both these technologies is Berkeley Nux[9], an open-source Java toolkit for XML processing, that we used for our prototype.

Log processing. Our prototype scans the Vordel logs as an XML stream and extracts the occurring violations in two stages: first, a filtering component – the Log Parser – performs a search of potential violations (similar to filtering indicator related events) over the current time window; these potential violations are kept in store with all their metadata. As of now, the filter looks for messages with tags $sam : a$ and $sam : c$, using the query shown in Listing 6.3. The XQuery query is stored in a String that is then executed by the Nux API with a $query.execute()$ invocation; the query declares the SOAP namespace for the SOAP envelope data; then declares another namespace that depends on the schema of the Web service being invoked, and lastly searches with the envelope for a certain node. From the indicator specification in the policy we know what nodes to filter from the XML namespace and node name.

Approximate text matching. On the events identified before, a second component independent of the filter – the Log Analyser – conducts a match of the records with the violations specified in the policy. This match can be on the current time window, or against all previous events; if the match succeeds, the indicator value for the anonymisation policy is updated. The input of this component also comes from the indicator specification, i.e. what should the patient data node text not consist of, for anonymisation. Hence we needed to look for node text that is different from or that approximately matches "xxxx". How to search for exact or approximate matches? Apache Lucene is a cross-platform text search engine that helped us for this task. One of the features of Lucene is that of wrapping a query with a text match functionality, whereby the match of two text parameters is done based on a score; the score measures string distance and is a value between 0 and 1, where 0 means 'not matching' and 1 means 'exact match'. The internal algorithm for this metric is that of the Levenshtein distance between two strings – the minimum number of edits (character delete, insert or substitute) to transform one string into another. Based on Lucene, as integrated in the Nux API, the XQuery query becomes only slightly more complicated, as shown in Listing 6.4. The $lucene : match()$ call will invoke the approximate string

[8] `http://lucene.apache.org/`
[9] `http://acs.lbl.gov/software/nux/`

```
String q = new String(
  "declare namespace soapenv =
   \"http://schemas.xmlsoap.org/soap/envelope/\";"
   + "declare namespace " + rc.getTagXmlns()
   + "=\"" + rc.getXmlns() + "\";"
   + "/soapenv:Envelope/soapenv:Body/"
   + rc.getRelevantTag());
 Nodes res=new XQuery(q,null).execute(doc,null,null)
     .toNodes();
```

Listing 6.3: Java code that uses the Nux API to execute an XQuery. The results are pushed in a resulting list of nodes.

```
"/soapenv:Envelope/soapenv:Body/"+tagPatientData
  + "[lucene:match(.," + anonymousValue() + ")>"
  + similarityScore + "]"
```

Listing 6.4: Adding the similarity score to the query.

match on the value of the patient data node in the SOAP envelope, with the anonymisation value, based on a given similarity score. By default the score is 0, meaning any approximations of the value "xxxx" are considered violations and hence added to the indicator value. But if, for instance, it is enough that "xxxx" be "xxx" followed by one more character, Lucene supports wildcard matching: $xxx?$ for any character after the first three; $xxxx*$ for any number of characters after "xxxx", or generically $xxxx \sim 0.8$ for a fuzzy match with a similarity score of 0.8.

Computing indicators. As explained above, using Apache Lucene and the expressive power of XQuery, we can extract all occurrences of unwanted node text where there is an approximation of the anonymised. How to count these occurrences? Counting these approximations is made by default by XQuery, and the obtained number is given to our mathematical formula evaluator[10]. This formula evaluator takes the expression of the indicator formula, and replaces the parameters with the counts of the matches returned by the query over the log. If the expression of an indicator has more than one parameter, there is no need to perform a new query: the original Log Analyser query can be enriched with new elements to search for in the same pass.

6.6.2 Obtaining More Logging

In line with the argument in Section 6.5.3, relying on the default Vordel logging feature to record just incoming and outgoing traffic from the point of view of the policy flow in Figure 6.5 is not enough. If the implementation of the anonymisation policy is not correct (even if the building blocks are in themselves 'safe'), then we would need

[10]We used a free mathematical expression evaluator library available at `http://lts.online.fr/dev/java/`

any logs internal to the policy to help us locate a deviation source more accurately. Such extra logging records need to contain data about what block or chain of blocks generated them, and can be obtained with filters or chains of blocks that have logging enabled. Note here that we don't want to rely on the policy enforcer developer to add the log invocations, since our assumption is that this developer is not reliable[11]. Also, default logging enabled for all blocks to be used in the policy implementation may be useful for a detailed analysis but may be excessive.

We observed that Vordel filters and block chains can be "instrumented" with log statements, simply by creating custom filters and blocks that wrap existing 'bare' ones with a default log block. Adding a default log block, in this way, means that the accompanying block or filter is prone to deviations. Obviously, this instrumentation should be done offline, before the policy implementation.

6.6.3 Possible Corrective Actions

The concrete correction that can be performed in reaction to the indicator deviation depends on two aspects:

1. the capabilities of each assessed system, that is - how well the sequence of enforcement blocks can be instrumented;
2. the type of indicator/correction being expressed;

Corrective actions aim to patch the misbehaviour in a way that is most appropriate to the type of misbehaviour. Thus, in essence the main idea when using corrective actions is that of *type flags* to correlate the corrective actions with the indicators and changes in their values. This flag is associated with the property that an indicator is measuring: correctness of enforcement (i.e. wanted effects happen when certain triggers occur), or robustness (i.e. how the enforcement process deals with faults or errors). In our work, we considered five types of indicators and related corrections:

Parameters. If the indicator is being calculated for a policy where parameters can be changed, and that have not been respected for correct enforcement, then the corrective actions will be used to adjust these parameters. For instance, it is possible that the parameter "3" when enforcing the policy "After 3 failed login attempts, block the account and alert the administrator" was not complied with and instead of 3, there were 5 login failed attempts before the account was blocked.

Security blocks. If the indicator is used to measure the correct functioning of a particular security block, the correction can be to either insert certain enforcement

[11] The developer is likely to overlook the logging blocks in Figure 6.5.

components before or after it, or to modify or replace the block with another one. For instance if there are several authentication violations for a user authentication block known only to perform authentication, then a solution would be to replace that block with another one performing the same function.

Chains of security blocks. If the indicator is used for measuring a chain of blocks, then the corrective action would be to use another chain instead. Similar as the case above for a chain of security blocks that is replaceable with another chain of security blocks, for instance a message authorisation chain that has two security blocks: a user role authorisation block and a message content validation block.

User/usage. If the indicator is used to identify a misbehaving user or the disallowed usage that can be linked to a certain endpoint, then the corrective action considered would be to ban the user or limit that activity. For instance, if there are more than 100 message requests from the same user (or endpoint) per minute, then that user can be suspected of initiating a Denial of Service attack and should be banned or redirected to another service provider or endpoint.

Default. A generic indicator that is not captured by the above four types will lead to a default corrective action, e.g., a system specific alarm, like ''inform the administrator", and wait for solution, or "invoke an emergency service".

For a flexible corrective system, it can be beneficial not to have corrective actions always specified in the policy. The reason is that it could not be known clearly beforehand if/where that corrective functionality is available at runtime. This approach – of letting the system instantiate a corrective action – allows the system to address certain scenarios impossible to handle otherwise. For example, consider a scenario where the policy writer suggests a concrete correction C, but the action cannot be performed because C was a Web service that has been undeployed. Knowing the type of the deviation and the corrective action required allows the system to choose another corrective action in the place of C. The improvement is to let the policy writer suggest a generic class of corrections C' associated with a property of a violation (type of violation, or location of violation). For example, a class of corrections is a "chain of block or filters" that perform "authentication with Kerberos" for an authentication violation located between "start-block" and "end-block". This provides a more flexible correction process: the type was specified explicitly but without a concrete endpoint instance of a Kerberos authenticator.

To a large extent the decision of whether to specify the corrective actions in the policy depends on the type of actions. For example, in some cases, the policy writer may not know exactly what to do when the indicator value changes, e.g., drops below

```
1 declare variable $tickStart as xs:long external;
2 declare variable $tickStop as xs:long external;
3 for $soapmsg in /relevantVordelLog/logRecord
4   where $soapmsg/timestamp/text() >= $tickStart and
5   $soapmsg/timestamp/text()  <= $tickStop
6 return $soapmsg
```

Listing 6.5: An XQuery that checks the scope of the analysed records.

30%. In such cases, he can instruct the system to correct the problem by using a block of filters that does a certain type of action, e.g., call a method, which could internally analyse the situation and take more targeted action. This approach works with single filters, or blocks of filters, or alarms when the policy writer doesn't care about exactly what happens where. However, in the cases where the corrective actions are parameter adjustments, this approach does not work and the corrective actions are specified in the policy as a piece of code that is interpreted and executed centrally.

6.7 Validation

If we want to validate our approach for assessing policy enforcement, we have to answer the following questions: can XQuery express the violation patterns that we need for computing indicators? Is a specific violation always caught? Is a corrective action correctly taken?

To answer the first question, whether XQuery can express the patterns that an enforcement indicator is based on, we want to see if this language can express requirements **R1**-**R7** in Section 6.4. First of all, all these requirements are over a scope – either the current time window or the whole history. XQuery has already a construct that can check the scope of the records being analysed: *where*. To designate the time window, this construct can be used as in Listing 6.5.

If the analysis is over all previous log records, the *where* construct can lack. The number of results returned by the previous query is the number of log events within the current time window. Adding an extra condition to the *where* construct as shown previously in Listing 6.4, would return the number of incorrectly anonymised messages with patient data. Comparing the former record count with the latter gives requirements **R1** - whether there is or not a deviation in the scope (latter count equals or greater than zero); **R2** – whether a deviation is universal over the scope (latter count equals former count); and **R3** – whether the deviation is bounded within the scope (latter count is within a predefined limit).

Precedence (**R4**) and response over a scope (**R5**) can also be addressed from within

```
1 declare variable $time as xs:double external;
2 for $b in /relevantVordelLog/logRecord,
3     $a in /relevantVordelLog/logRecord
4  where $b/timestamp/text() - $a/timestamp.text() > $time
5  return
6   <seqViolation>
7       {$b/logPayload}
8       {$a/logPayload}
9   </seqViolation>
```

Listing 6.6: An XQuery to show the precedence or response violations.

XQuery. Precedence imposes an order over a set of events in the log, most likely conditioned by a timestamp or a correlation relation. For instance, a condition like "event B cannot occur after event A if later than 10 minutes from A" can be modelled with an XQuery join query (similar to join queries in databases). The query below returns all precedence or response violations (replacing '>' with '<') is shown in Listing 6.6.

A query like this does express the desired condition, but since it does not give more details about the correspondence between the two events, several A events might be associated with one B event. Hence, a further relation between events B and A can more accurately correlate the 'right' A events with the 'right' B events. A similar argument can be done for requirements **R6** and **R7**, by increasing the number of join branches. For these reasons, we can conclude that indicators specified as combinations of the requirements expressed in Section 6.4, can be expressed in XQuery. As a matter of fact, previous work [119] has shown that XQuery is a Turing-complete language, hence we can make the claim that any computable indicator on policy enforcement can be expressed as a query in this language.

A related observation would be that the use of XQuery relies entirely on having logs in well-formed XML. This assumption holds for the middleware that we are aware of, since the log format is often customisable to XML (e.g., Vordel gateway, Mule ESB, Servicemix ESB, Tivoli). We do not find this to be a limitation for tools that do not produce logging in XML, since it is feasible to format the data into to custom XML by means of an additional log processing step.

Computing indicators relies not only on an expressive language but also on correct violation identification. The violation may not always be easily specifiable: while for an anonymisation value, it is easy to say that sensitive data is not anonymised if it does not match exactly with a string value, for a failed login attempt for instance, the effect can manifest itself as either a SOAP fault (with parameters) or as an non-fault message whose payload contains a server error (with parameters). Hence the violation specified just by its wrapping object will not be detected, unless the search

query contains elements that uniquely identify the misbehaviour as it occurs. It is likely that a violation need be identified by several string parameters, that will have to be checked via string matching against the log records. In such cases, care must be taken so as to implement the fuzzy string match correctly. It can be that the matching score of some parameters differs from others', or that some parameters may return positive matches while others might not exist in the log records.

Whether a corrective action is correctly taken is a trickier task to evaluate. It can be difficult, for instance, to know if the real cause of the misbehaviour is due to the implementation of the enforcement process, of a mechanism within the process, or a problem in the specification of the policy or of the violation. For instance, let us assume that after three login attempts that fail, the system has to send an alarm, but for some reason, it doesn't. In such case it is hard to detect if this is a misbehaviour of the system – in fact waiting for a fourth failed login attempt, or a wrongly defined "failed login attempt". In our approach, we assume that the deviations are straightforward to define and detect, and they are already known beforehand in terms of their category. Discovering the deviations without knowing how they map to existing deviation categories pertains to root-cause analysis, which is outside the scope of this discussion. Root cause analysis can be used as a separate offline process that can give feedback to the policy writer to help him decide on how new deviations map to existing (or not-yet-existing) categories.

6.8 Discussion

There are several other aspects worth noticing about our assess-and-correct approach. As mentioned before, the policy creator may not be aware of all types of problems and their associated corrective actions. From this point of view, our framework can observe *known* unwanted deviations and help improve the way security mechanisms enforce policies. Also, corrections rely on suggestions from the policy creator. If the corrective feature is not implemented in the system, then the demanded correction will not happen.

What type of policies can our approach assess? Security indicators can assess the enforcement of policies whose effects satisfy two conditions: 1) are observable to the assessing mechanism, and 2) were logged. Observability means the visibility that the assessing mechanism has over enforcement events. In our healthcare scenario, the hospital owns the messaging infrastructure, and hence can reliably access the infrastructure logs because it owns the assessment mechanism and the logs. When the system logs are owned by another party than the assessor, the assessment problem

is complicated by the need of the log owner to filter out private details. A related aspect is what can actually be logged that is useful for enforcement assessment. Our method can assess access control policies on the access and usage of application-level resources: Web services and communication among Web services (e.g., SOAP message attachments). Actions such as deletion of a file on a machine, or flow of data extracted from a file within the distributed system – cannot be normally traced with an XML gateway or cannot be detected from network communication logs.

Deploying our framework does not require large modifications to the existing security mechanisms, but enablers for indicator computation and corrections. From the elements presented in Figure 6.4, the Log Analyser, Indicator State Updater and Correction Evaluator blocks are single centralised components of the assessment system. Different correction performers can be deployed as a trusted Web services, and have to be known to the Correction Evaluator. The Log Parser can be deployed either centrally if the logs are kept in a centralised repository, or can be distributed to several endpoints that would send their results to a centralised log analyser. The Enforcement Process Modifier assumes the only modification to an existing enforcement process, but it would be transparent for both the security developer, and the application user. As explained before, the modification can be to log all security mechanism activity by default, or the activity of certain security blocks.

6.9 Related Work

Policy compliance elicited by analyzing execution logs is not new. Aalst et al. approach the problem by comparing the logs of a chosen system node against an abstract model [1]. Uncompliant message flows are judged in terms of appropriateness (how well the model describes the observed behaviour) and fitness (how the observed behaviour complies with model) for one business process instance whose model is given in BPEL. We argue that these two criteria are hard to grasp for a system maintainer. The problem of how to define compliance with a policy is also approached formally by Etalle et al. [58]; they suggest how to log relevant events at system runtime and how to query them to detect uncompliance, but this evaluation is done a-posteriori. Log auditing at runtime is also formalised by Roger and Larrecq [201], and by Tripakis [225]. While the former reports on some very preliminary experiments, the latter combines an online diagnoser – that gives data about how correct the system is functioning – with an offline tester – that gives data about observed system faults. Nevertheless, both approaches are theoretical and do not consider how to dynamically adapt the system to the discovered misbehaviours. Similarly, the work

of Serban and McMillin [212] is theoretical in that it combines security assertions with event histories, but is only concerned with whether the runtime of the application is compliant or not with a set of specifications. Our work, on the other hand, combines quantitative assessment of enforcement with corrective actions; this is by contrast with related work, where either policy enforcement assessment, or only corrections, are approached. Existing works do not approach the problem from the point of view of the occurrence of meaningful misbehaviours, nor do they differentiate between policy specification and its implementation. Our viewpoint is that if misbehaviours occur, the implementation of the policy, rather than the policy itself, should change. Moreover, our approach is practical in that it offers a concrete implementation of the enforcement indicators at message level, along with concrete suggestions for implementing the correction framework onto the XML gateway.

Assessing a system's security level by means of metrics is yet a controversial issue. Shostack and Stewart promote actual measurements in addition to considering recommended best practices, since they can observe what is actually taking place in a system [214]. That metrics need be contextualised is the consequence of works like [159], suggesting there can be no metrics that can correlate with defect density in software projects, or other useful numeric criteria; also, Jacquinth gives a number of features that a useful security metric [113], but the suggested examples lack context specificity. At the other extreme, there are opinions like Bellovin's [25], who argues that software compliance is not continuous and it may be that a percentage of security may not mean anything to the administrator. Our viewpoint, on the other hand, is that measurement is needed to evaluate how correct the enforcement infrastructure works compared to the security developer's goal. We see this correctness function as being continuous, depending on time and on various events that need or need not happen in the given context.

6.10 Summary

Assessing the enforcement of a policy is essential to enterprise management, security administrators and developers alike. The enterprise management learns how much or how little the application complies with its regulations, based on how well deployed mechanisms enforce policies; security administrators and developers obtain concrete evidence about the occurrence of violations and learn how to better address them. To assess policy enforcement, in this chapter we have proposed the notion of runtime indicators and corrections for policy enforcement, at the message level middleware in an SOA application. Enforcement indicators can be used to track the misbehaviours

in the implementation of the enforcement of a policy at runtime. They are not flags of whether the enforcement is correct or not, but rather a quantitative measure, e.g., they signal how many violations of type V there are (at a certain location) when enforcing policy P.

We propose to link indicators with corrective actions for the detected misbehaviour. Our feedback system is not self-learning, since corrections are based on suggestions from the administrator, but is adaptive, since corrections apply on the fly to the enforcement mechanisms and their efficiency is evaluated in the next round. Corrections can range from raising a system alarm, to replacing an endpoint known to perform a security function, with another endpoint for the same function. How to choose semantically equivalent components and use them for security enforcement is out of the scope of this thesis. However, plugging or replacing security components involves establishing communication connections that, if not configured properly, can damage rather than improve the value of the enforcement indicators. The next chapter explores configuration issues on the links among security components.

Chapter 7

Configuration Management

7.1 Introduction

With systems like that of the Nationwide Health Information Network Exchange (NHIN)[1], bound to exchange health information across over 30 organisations, policy enforcement becomes difficult. This is because there are multiple patients, users (e.g., healthcare personnel, patients) and constraints across more than one system. When patient or user data, profiles and action histories are spread across several domains, it is difficult to make this data available safely and consistently across the system. Maintaining this data is essential for correct security decisions, but is complicated by different domains introducing specific data access constraints that can be conflicting.

As of now, security links are piggybacked on normal application connections among application components. This means that security data that needs to be exchanged and processed across domains is transported on the links that the application has already established for application purposes. Dedicated security channels exist with very specific security entities such as certification authorities, and are much less frequent than the connections among authorisation servers, security data sinks, etc. The consequence is that performance constraints over application connections impact security links. In this chapter, we focus on the impact of three performance-enhancing techniques – caching, data retrieval and data correlation for security data – on the correctness of the security policy enforcement. In other words, even if policy enforcement has been designed and implemented correctly, the way the security data is handled by a system that has to show best performance, can severely undermine the system's security.

Caching can impair policy enforcement. Commonly, user certificates or sessions, authorisation decisions or patient data are cached so that they are quickly retrieved.

[1]NHIN `http://healthit.hhs.gov/portal/server.pt?open=512&mode=2&cached=true&objID=1142`

For instance, for a hospitalised patient being treated in the Radiology ward, the non-radiological data in the patient's health record are not likely to change frequently, hence they are cached on a server of the Radiology ward; in this way, the data retrieval would be fast when changes need to be persisted into the patient's record. However, when this data changes faster than it is cached, the decision of authenticating or authorizing actions may no longer be correct, in which case revenue or human lives can be at stake. For instance, there can be a hospital policy forbidding an MRI scan to a patient in the first hours after been administered a drug. If the patient is administered a certain drug in any other department, and by the time the request for a MRI scan arrives the cache would not have been refreshed (this is likely, since refresh rates are usually set independent of the application - e.g., at night and not necessary every night), then the MRI scan request would be authorised. But the decision would have been incorrect, since it relied on stale cache data.

Attribute retrieval can also affect policy enforcement. Attributes refer to all data needed for authorisation decisions (e.g. user profile, history, system state, user state). Retrieval can be done directly or using a mediator, in which case it runs the risk of staleness. Also, if attributes are semantically related, their retrieval should be the same way (we call it *correlation*). Attributes are not always retrieved directly, for two reasons: (1) it is costly to retrieve them every time, and (2) the data is private. For instance, a hospital has multiple wards (or departments) that operate on patient data, that have their own authorisation systems, and that are bound to internal or hospital security policies. To authorise a healthcare action (e.g., a surgery) onto a patient in ward A, the hospital system might need to check data from a legal department (e.g., patient's or patient's family surgery consent). If the data in other wards isn't retrieved in its freshest state to the system in ward A, or if too much data from other wards (e.g., maybe in other hospitals) is given to ward A, then data that is not immediately needed for authorizing an action is leaked and can be misused.

When security enforcement and performance collide, such vulnerabilities occur in different systems tuned for performance. The Common Vulnerabilities and Exposures database [154] reveals numerous entries related to cache and configuration management. For instance, OpenSSL suffers from race conditions when caching is enabled or when access rights for cache modification are wrong (CVE-2010-4180, CVE-2010-3864, CVE-2008-7270). Similar problems of access restrictions to cached content are with IBM's DB2 or IBM's FileNet Content Manager (CVE-2010-3475, CVE-2009-1953), as well as ISC's BIND (CVE-2010-0290, CVE-2010-0218). Exploiting such issues is reported to lead to buffer overflows, downgrade to unwanted ciphers, cache poisoning, data leakages, or even bypassing authentication or authorisation. Protocol implemen-

tations like OpenSSL and BIND, or servers like IBM's and RSA Access Manager, are examples of technologies that need to scale fast, but cannot scale *fast and safely*. This is a real problem with expanding hospital systems, that are outsourcing services to other institutions, and integrating services in super-networks like NHIN. Such super-networks need to be both fast and secure, primarily for the well-being of their patients.

This chapter argues that the techniques to improve a distributed application's performance must be tailored to the application's security needs. In our view, caching, retrieval and correlation of enforcement attributes require a management layer that intersects both the application and its deployment environment. To our knowledge, these aspects have not yet been approached in security enforcement. In this chapter we describe a method and framework for adjusting at runtime the security subsystem *configurations*, i.e., the ways to connect components and tune connection parameters. This adjustment requires to find ways to connect security components so that their connections satisfy a set of constraints, specified by domain experts or administrators (e.g., hospital security architects or security domain administrators) and included as annotations in the XACML security policy. The system configuration that allows for both security and performance needs is *dynamic*. Constraints can change because the tolerance for either performance overheads or for inaccurate enforcement decisions can vary; security domains can vary in dimension; infrastructure topologies can change. Since varying runtime constraints imply re-evaluation rounds, we want to automate the reconfiguration of the authorisation subsystem. Thus, our contributions are:

1. We show the need to consider security and performance restrictions together, rather than separately. We focus on attribute retrieval, caching, and correlation. To our knowledge, we are the first to look at caching from the perspective of the impact of stale attributes over the authorisation verdict when enforcing a policy.

2. We present a method to dynamically compute the correct configurations of policy enforcement services, by transforming system constraints into a logic formula solved with a constraint solver.

3. We describe the first middleware tool to perform adaptive system reconfiguration on security constraints. Having split computation in two phases, preliminary results show that the heavier computation phase takes 2.5 seconds to compute a solution for over 1000 authorisation components.

The contributions presented in this chapter have been previously published [79].

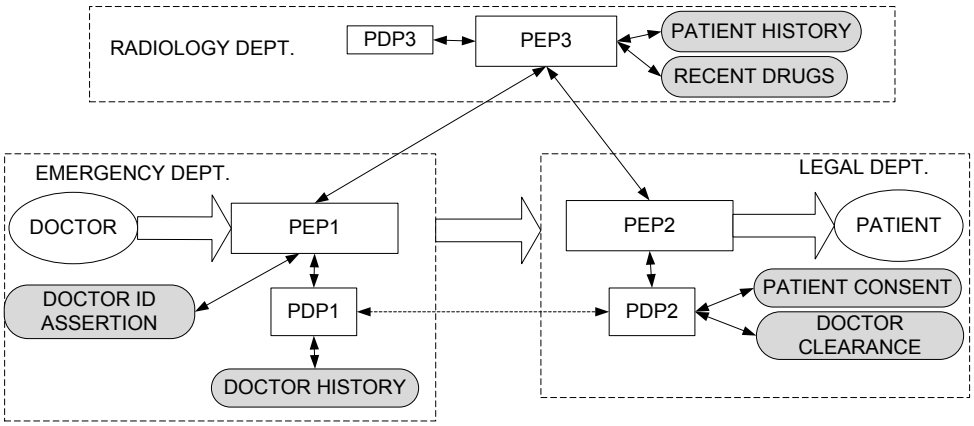


Figure 7.1: Cross-department authorisation for a Doctor in EmergencyDept. from the patient representative in LegalDept., with hospitalisation information from RadiologyDept. The greyed boxes are the surgery authorisation attributes: the doctor's id assertion, doctor's surgery history, patient's consent and clearance, and patient's radiological history and recently administered drugs.

7.2 Requirements from the Case Study

In what follows, we deem *security subsystem* the ensemble of components that perform authentication or authorisation enforcement – a set of PIPs, PEPs, and PDPs. We make the realistic assumption that, for large enterprises in healthcare, the security subsystem is divided into security domains (i.e., groups of components under similar security policies). User credentials, profile and state are spread across different domains; access patterns to such data can vary, and domains need to authenticate to other domains when user data is retrieved.

We target policies whose evaluation require scattered security data, e.g. the policy:

> (**P1**): "*An authenticated doctor cannot perform a surgery on a patient unless he has a clean history record, has the patient's consent and the patient has not been administered any unsafe medication in the last 12 hours*."

The doctor's history, the patient's consent, identity tokens, and patient's record and medication history – this data is kept at various locations where it may change at runtime, with an impact over allowing or disallowing further user actions (e.g., surgery authorisation). If doctor and patient attributes are not updated correctly or if fresh values are not retrieved in time, surgeries will be prevented for all cases, or patients who were administered unsafe medication might be allowed to undergo surgery even if they shouldn't. Figure 7.1 shows a multi-department system where in order for a doctor in domain EmergencyDept. to be allowed to operate on a patient, the surgery request must pass through the security subsystem of the RadiologyDept., gather some

data about the patient history, and pass through the security subsystem of LegalDept., which will require some data on the patient's consent to the surgery, and to the doctor.

For us, the problem of attribute management as shown above can be solved by finding a tradeoff between security correctness and performance needs. We want to compute and recompute such balance at runtime and adapt the security subsystem accordingly. We do not want to change the entities of the security subsystem, but the connections between them that involve retrieval and caching of security attributes.

7.3 The Configuration of Attributes

Attributes are application-level assets, and have value in the enforcement process (that is aware of application semantics). In our example, doctor and patient history, doctor id and clearance, patient consent, and patient drug medication are all attributes; in Shibboleth [105], the LDAP class *eduPerson* defines name, surname, affiliation, principal name, targetId as usual identity attributes.

How to obtain attributes is specified by *configuration (meta)data*. For instance in Shibboleth, identity and service providers publish information about themselves: descriptors of provision, authentication authority and attribute authority, etc. Such metadata influences the enforcement process, and can cover:

- visibility (e.g., is the identity assertion server reachable? is the patient history public or confidential across certain departments in different hospitals?),
- location or origin restrictions (e.g., a surgeon's action history can be absolute or per institution),
- access pattern of security subsystem to contact third-party (e.g., if the patient history or health record can be encrypted, can be backed up or logged),
- connection parameters (e.g., how often should patient or doctor history data be cached or refreshed? who should retrieve it?).

Our point is that this configuration level, including attribute metadata, complements the enforcement process and can influence it. We examine three aspects: (1) push/pull models for attribute retrieval, (2) caching of attributes, and (3) attributes to be handled in the same way (coined 'correlation').

7.3.1 Attribute Retrieval

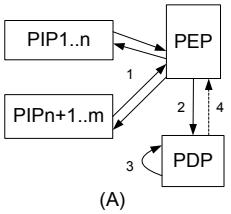
In our healthcare example in Figure 7.1, *attribute push* from PEP1 and PEP3 to PDP2 means that some attributes – 'doctor's identity assertion' , 'doctor's history' and the

patient information from RadiologyDept. – are pushed to the LegalDept. and then to PDP2, which will make the final authorisation decision. The 'doctor history', 'patient history' or 'patient's recent drugs' can also be *pulled* by PDP2 at the moment when it needs such data.

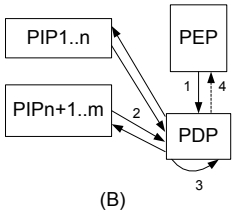
From the point of view of performance, both attribute push and attribute pull can be problematic. In the attribute pull case, an excessive load on PDP2 can make it less responsive for other surgery requests. Performance is linked with security: low PDP2 performance can be made into a Denial of Service attack by saturating PDP2 with requests that require intensive background work. The case of all attributes being retrieved by PEP1 and given to PDP2 is also delicate: for example, if PEP1 knows what PDP2 requires exactly by 'doctor history', it means that PEP1 will do the computation instead of PDP2. This scheme can put too much load on PEP1, that can have a negative effect on PEP1's fast interception of further events.

From the point of view of trust and intrusiveness, the push scheme is problematic. In Figure 7.1, PDP2 must trust what PEP1 released as doctor history (e.g., is it relevant to this surgery and patient? is it more than that?), and this potentially sensitive data would travel between PEP1 and PDP2. Making this decision triggers several risks, of which: PEP1 might release more data than necessary data; taking the decision might be delayed; if the policy changes (e.g., from "clean surgery record", to "at least one successful surgery") then PEP1's logic should be updated. The security subsystem should decide if exporting the computation of the relevant past surgeries to PEP1 costs more than the privacy of such local data to the surgeon's domain. In either case, PEP1 and the communication channel should be trusted not to tamper with the data, and the policy should be fixed.

To determine when to use the push or the pull scheme, we have analysed a number of possible policy enforcement configurations and several existing scenarios and their solutions – PERMIS [35, 36], Shibboleth [105], and Axiomatics [17]. In particular, PERMIS looks at role-based authorisation in multiple Grid domains. There are two kinds of regulations on subject credentials in PERMIS: policies on what credentials can be attached to the authorisation request leaving from EmergencyDept. (policies enforced by PDP1 in Figure 7.1) and can reach LegalDept.; and policies on what credentials can be trusted by LegalDept. as the destination of an authorisation request from EmergencyDept. Validating a buyer credential in LegalDept. – done by PDP2 – involves a chain of trust among issuers that have different rights to issue attributes for different sets of subjects. On a similar note, Shibboleth [105] enforces attribute release policies whereby once a user has successfully authenticated to a remote website, certain user attributes can be provided on demand via back-channels by requesting



(A)



(B)

Push scheme	Pull scheme
stateless attributes	state attributes
environment attributes	behaviour history
not sensitive user data	sensitive user data
request details	obligations
destination details	domain decisions
stateless context	stateful context

Figure 7.2: Diagrams A and B show the classic attribute push and pull models. The table to the right shows how different attributes fit the push or pull scheme better.

Web sites while other attributes cannot. These examples confirm that user or system attributes are sensitive, and should not always be pushed to the PDP.

Based on the security considerations above, we separate the applicability of the push and pull schemes based on types of attributes (see Figure 7.2, right). The push scheme is more appropriate for those attributes that can be safely collected by the PEP or can only be retrieved by the PEP rather than the PDP (e.g., message annotations whose semantics is known by the PEP: IP address of a surgeon's machine, description of a token service, location and country of a certificate server). A similar treatment may be for attributes unlikely to change frequently: user identity, group membership, security clearances or roles, authentication tokens, or any constant data about system resources or the environment that the PEP can retrieve easily. Conversely, the PDP directly interacts with PIPs when it needs specific application logic and related state (e.g., patient and surgery data, medical obligations), history of a patient, or data that only the PDP is allowed to access (e.g., recent drugs administered).

7.3.2 Attribute Caching

Caching in policy enforcement can be of three types: of (1) attributes, (2) decision, and (3) policy [245]. The PDP may cache policies that change little in terms of content, but caching attributes with different change rates is difficult. For instance, updating doctor surgery history depends on how often the doctor performs surgeries. This history can be updated frequently, or can remain unchanged if the user does not perform any surgeries because none are requested. Conversely, a surgeon's "rating" [2] can be a measure of how many successful surgeries he or she has performed. This rating can change at once, when the surgeon performs unsuccessfully. In short, these examples show that some attributes tend to change unexpectedly. Unlike such attributes,

[2]We have not seen the notion of a doctor's rating based on medical success to be implemented in practice, but it would certainly be viable in a private medical system.

enforcement decisions and policies are more static. Yet for similar reasons as for attributes, the PDP or the PEP might maintain a cache of the decisions already made. We hereafter focus on attribute caching, but our framework can apply to decision and policy caches too.

Table 7.1: Different caching concerns in enforcement.

Scheme	**What to cache**	**Where to cache**	**How to invalidate**
Attribute caching	attributes	PEP or PDP	time limit
Decision caching	policy decisions	PDP	explicit
Policy caching	entire policy	PEP	explicit

Since cached attributes are attributes too, the push and pull scheme applies to caches as well as to attribute retrieving. New values of the attributes needed to make a policy decision can be either (1) pushed to the entity that needs them, be it the PEP or the PDP, or (2) pulled by the same entity that needs them. There are two cases that combine both caching and attribute retrieving issues:

1. *push to PEP cache, push to PDP*: a PEP pushes attributes to the PDP, and whenever an attribute is updated, the PEP cache is notified and updated. The scheme relies on an external entity to notify the PEP of attribute changes. Inaccurate notifications can compromise decision correctness.

2. *pull to PEP cache, push to PDP*: a PEP that pushes attributes to the PDP, and periodically queries attributes for fresh values (pull). This case does not use a third-party but puts more load on the PEP. Also, the PEP should have poll times that depend on the change rate of the attributes to be cached.

The cases when the PDP pulls attributes by itself and stores them are similar in that cached values need to be refreshed at a rate decided either by a (trusted) third-party or at the rate at which the PDP can query the data sources. From this last point of view – the polling rate of the PDP – cache management has the notion of cache invalidation policy, whereby the cached values have a validity time; when they become invalid, the manager will retrieve fresh copies. Table 7.1 shows how to invalidate caches depending on what security aspect is cached.

7.3.3 Attribute Correlation

In an interview with Gunnar Peterson (IEEE Security & Privacy Journal on Building Security), Gerry Gebel (president of Axiomatics Americas) pointed out that there can be different freshness values for related attributes [77]. The example given is that of

a user with multiple roles in an organisation; whenever the user requests access to a system resource, the PDP needs to retrieve the current role of the user; some of the user's roles may have been cached (in the PDP, PEP or PIP) hence there might be different freshness values to be maintained for several role attributes. In our healthcare scenario in Figure 7.1, it can be that more than one patient attribute changes from the moment a surgeon knows about it, and before the surgery is decided on. In order for the security subsystem to allow the surgeon to proceed with a surgery, it must gather surgeon information, surgery restrictions, *and* patient information. If at this stage, the system can access the freshest patient history, but not the last medication administered to the patient, then the surgery might be allowed even if it should not proceed.

We generalise such examples to the idea that correlated attribute need a common refresh rate for all the attributes in the group (e.g., all role attributes from an LDAP server, username and password attributes, source IP and port attributes, role and mutually exclusive role list, etc). Some attribute correlations are application-dependent (e.g., mutually exclusive roles, like doctor who approves the surgery vs. the doctor who requests the surgery), others are application-independent (e.g., username and password for single sign-on). Hence, bundling attributes that should be used together is a must when enforcement decisions need to be accurate. With PEPs or PDPs likely to use overlapping attribute sets to enforce a cross-domain policy, synchronization over common attributes is essential for attribute consistency.

Having a control over the attributes used in policy enforcement implies the need for a management layer that we call 'configuration layer'. We continue by describing our solution to the issues above.

7.4 Solution Design

We consider the setting of a distributed application on top of which there is a security subsystem in charge of enacting several security policies. This subsystem consists of multiple PEPs, PIPs and PDPs. In the state of the art so far, the connections between these components are fixed once a policy is deployed. In our view, they should change at runtime, since this way the system can *adapt* to varying security or performance constraints. How often these changes are incorporated into the security subsystem depends on which of security or performance is paramount, and on the disruption incurred when actually changing the connections among security components.

We will use the term *wiring* for enabling a configuration on the concrete infrastructure (realizing the connections among PIPs, PEPs and PDPs in order to support

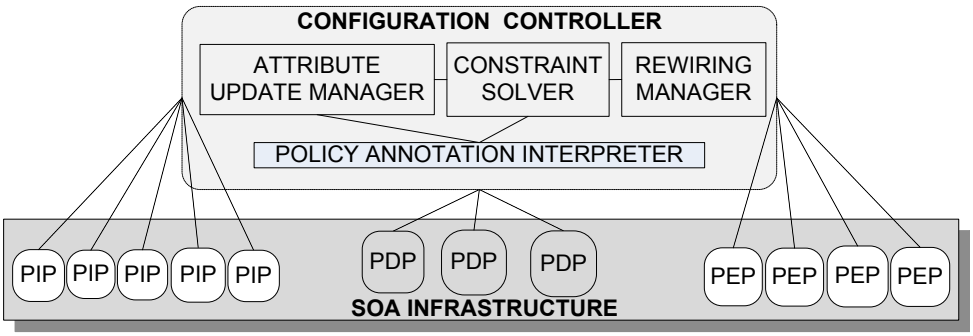


Figure 7.3: The architecture of our solution.

the attribute management features explained above). We want to rewire the different authorisation components with hints from the application domain and knowledge of the enforced policies. We see such hints supplied with the XACML policy, since it is likely that the policy writer, aware of some application domain details, can formulate requirements for the treatment of attributes. We also assume that security components have standard features: a cache storage space (hence also cache size and cache refresh policies), permissions over enforcement attributes, or other properties (domain, location, etc).

The architecture of our solution is presented in Figure 7.3. Instead of having a set of hardwired connections between the PEPs, PIPs and PDP, we envision a configuration layer on top of the SOA infrastructure. This layer enables the dynamic rewiring according to varying runtime conditions of the application. Responsible of this task is a Configuration Controller (CC) consisting of four components: a Policy Annotation Interpreter (PAI), an Attribute Update Manager (AUM), a Constraint Solver (CS) and a Rewiring Manager (RM).

The **Policy Annotation Interpreter (PAI)** extracts the policy annotations relevant for the configuration of the security subsystem.

The **Attribute Update Manager (AUM)** monitors the value changes of security attributes and notifies or directly propagates the new values to PEPs and PDPs or to their respective caches. This component also manages attribute synchronization and consistency. It should be connected to all PIPs needed to enforce a policy. The purpose of the AUM is to propagate attribute changes to the security subsystem. The environment, user properties or security relevant state might change at runtime, so the security subsystem needs to be notified.

The **Constraint Solver (CS)** is the component that finds solutions to satisfy the connection constraints of PEPs, PIPs and PDPs. Such constraints cover attribute retrieval,

```
<xs:element name="AttrProps" type="att-xacml:AttrPropsType"/>
<xs:complexType name="AttrPropsType">
   <xs:attribute name="AttrId" type="xs:anyURI" use="required">
   <xs:sequence>
     <xs:element ref="att-xacml:Providers" minOccurs="1" maxOccurs="unbounded">
     <xs:element name="AttrNotCached" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrCacheable" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrPushable" minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="att-xacml:CorrelAttr" minOccurs="0" maxOccurs="unbounded">
   </xs:sequence>
</xs:complexType>
```

Figure 7.4: Attribute meta-data elements in the `att-xacml` schema.

caching and correlation, and are specified in the security policy. The CS processes the policy, searches the solution space, and selects the configurations of the security subsystem that do not violate the constraints.

The **Rewiring Manager (RM)** enacts the solutions found by the CS. The RM resides at the middleware layer and is dependent on the SOA infrastructure. Rewiring can be done by the ESB itself. How the ESB can rewire its components is not within the scope of this chapter, since it was addressed before [87].

With this approach in mind, we continue by examining types of runtime constraints, how they can be specified, and how the CS can handle them. There are several assumptions for our running system: first, all PIPs are considered to provide fresh information to the other security components. Then, PEPs can cache PIP data, and PDPs can cache enforcement decisions and policies.

7.4.1 Annotating XACML Policies

For surgery authorisation policies like P1, our approach is qualitative. We aim to ensure compliance with quality considerations such as: how to pick a provider if there are multiple providers of surgeon or patient data, or how to retrieve data (e.g., if the histories or medication attributes have to be fresh), etc. This additional data (where an attribute can be retrieved from, if it can be stale or pushable) belongs in the authorisation policy. The reason is that the policy writer usually has the correct idea over attribute usage and invalidation with respect to the correctness of the policy decision; the policy writer knows if the user token does not change a lot so that it can be pushed to the PDP, or if the buyer/seller feedback rating is critical and volatile so should not be cached.

The XACML 3.0 syntax does not natively support attributes about subject, envi-

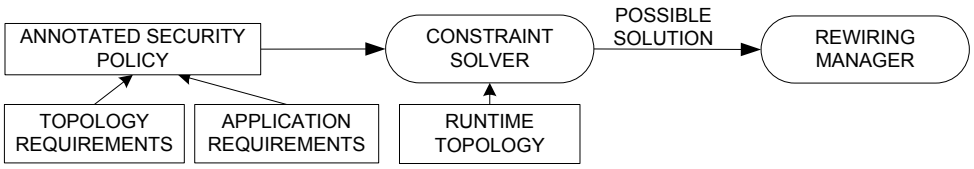


Figure 7.5: Model for configuration and rewiring in our approach.

ronment or resource attributes. We propose to enrich the default XACML syntax with annotations specific to the following aspects: (1) what PIP/PEP provides an attribute, (2,3) if an attribute can be cached and where, (4) if an attribute is pushable or pullable and where, (5) the correlation among different attributes. Until a XACML profile bridges this gap, a solution to this problem is to specify attribute metadata as an element in an enhanced schema we call `att-xacml` and part of which is shown in Figure 7.4. The `AttrPropsType` type consists of the properties of attributes that we are interested in: `attribute-id`, and a series of elements to indicate on what PEPs or PDPs they can be pushed, cached or not cached, what PIPs provide them, and correlated attributes. We assume that these features, attached to each attribute in a policy, are interpreted by an extension of the XACML engine. The PAI processes the semantics of these attributes for the CS.

7.4.2 Satisfying Configuration Constraints

The restrictions on attributes that were specified in a XACML-friendly syntax in Section 7.4.1 will reach the Constraint Solver (CS). The CS has to match these constraints against its runtime view over the authorisation subsystem, which we call *runtime topology*. The result is a number of solutions that satisfy all constraints; we call these solutions *configuration solutions*, or *configurations*. If the CS finds no solution, then the constraints are not satisfiable and the security subsystem remains unchanged. The CS is shown in Figure 7.5. Satisfying configuration constraints is complicated by the existence of different and interdependent constraint types. It is not in the scope of this chapter to analyse the impact of each of these constraints over the resulting configurations. Yet, for now, we acknowledge that reconfiguration depends on how often and what triggers the CS to re-evaluate its previously found configurations and issue new ones. Here we see two possibilities: the re-evaluation can be per policy, or per policy subpart. In the first case, the trigger of the reconfiguration is a new policy version, and the changes required for the new configuration might be scarce and far apart in the topology. In the second case, the trigger of a reconfiguration is a change in the runtime topology that relates to an attribute provider or consumer referred to by a

security policy; here, the reconfiguration changes are likely to be closer together in the topology. It is hard to say which one happens more often, since this is application dependent. A tradeoff must be made between a system that reconfigures itself either too frequently, or never.

7.4.3 (Re)Wiring

Rewiring of PEPs, PDPs, and PIPs refers to changing the connections among these components. These connections are established at the SOA *middleware* level, that enables component intercommunication. xESB can be the manager of the security subsystem. The main reason is that, by design, the ESB controls the deployed security components and their connections; when runtime conditions change (e.g., components appear or disappear, security policies are updated), xESB can modify message routing on the fly (e.g., validate credentials before they reach an authorisation server), trigger attribute queries (e.g., check the feedback rating or surgeon history attribute every minute and not every hour) or change attribute propagation to security domains (e.g., what entities are entitled to receive patient history updates). In particular, the dynamic authorisation middleware in [87] applies the dynamic dispatch system of a generic ESB to enable run-time rewiring of authorisation components across services. Each authorisation component (PEP, PDP, PIP and PAP) is enriched with an authorisation composition contract, and a single administration point allows the wiring and rewiring of authorisation components. Using lifecycle and dependency management, the architecture guarantees that authorisation wirings are consistent (i.e., all required and provided contract interfaces match), and that they remain consistent as the rewiring happens. Since rewiring was addressed before, here we focus on the constraint solving problem.

7.5 Implementation

From the components shown in Figure 7.5, we concentrated on the constraint solver component. While aspects about the RM and the AUM have been approached in previous work, runtime constraint solving is the most challenging aspect of our solution. We assume the CS receives runtime data about the components of the security subsystem, and attribute constraints from the deployed policies (Section 7.3), and produces configuration solutions that satisfy the constraints. We have prototyped the constraint solving in Java with the SAT4J[3] SAT solver.

[3] `http://www.sat4j.org/`

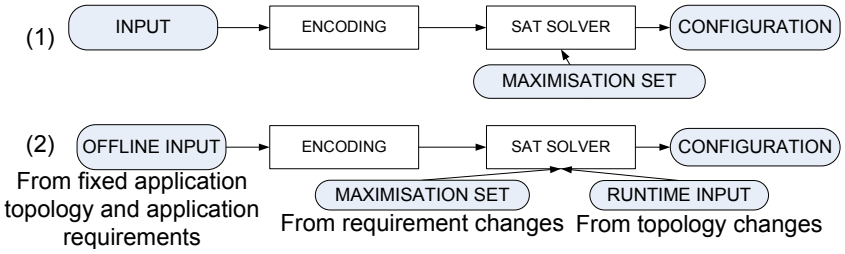


Figure 7.6: Two SAT solver methods to solve constraints (inputs and outputs shown darker)

7.5.1 Constraint Solving with a SAT Solver

We used a SAT solver because we wanted to obtain configuration solutions for a wide range and number of constraints (Section 7.5.2 explains why it is difficult to do this without a SAT solver). Given a propositional formula in conjunctive normal form (CNF) describing both the problem and its constraints, a SAT solver is designed to find an assignment of the propositional variables that satisfy the formula. The solver can also address partial maximum (PMAX) SAT problems: given a set of clauses S (a 'maximization set'), find assignments that maximise the number of satisfied clauses in S. This approach naturally maps to the configuration problem that administrators are faced with. For a security-aware application, satisfying all security constraints is paramount, while performance constraints should be maximised if possible. The reciprocal is treated similarly.

Fig. 7.6 shows two ways to use a SAT solver in our problem: in the first case, an encoded input is fed to the solver to find a valid configuration. If some part of the input changes at runtime, the whole input will be encoded again, and the solver will recompute all solutions from scratch. This method requires a massive effort to re-encode the entire problem, even if only a small part of it has changed. We employed an alternative incremental approach. The SAT solver receives an initial set of clauses, and if a subset of these clauses change at runtime, its internal mechanisms recompute only the subpart of the problem that has changed. A maximisation set can be provided in both cases. Next, we describe the encoding, offline and runtime input in our approach.

Offline Input. The administrator sets up the general settings. The set $PIPs$ of all possible PIPs, the set $PEPs$ of PEPs, and the set $PDPs$ of PDPs, say which attributes can be provided by each component. Specifically, for each $pip \in PIPs$, $pep \in PEPs$, and $pdp \in PDPs$, the following data is provided:

- $\texttt{provide}(pip, a)$ iff PIP pip can provide attribute a;

- $\texttt{provide}(pep, a)$ iff PEP pep can provide attribute a;
- $\texttt{needs}(pdp, a)$ iff PDP pdp needs attribute a;
- $\texttt{pull}(a, pdp)$ iff PDP pdp needs attribute a directly from some PIP;
- $\texttt{correlation}(pdp)$ iff PDP pdp either pulls or pushes all attributes;
- $\texttt{not\_cacheable}(a, pdp)$ iff PDP pdp needs freshness for attribute a;
- $\texttt{cached}(a, pep)$ iff PEP pep can tolerate a stale attribute a.

These predicates become propositional variables for the solver, along with:

- $\texttt{arch}(pdp, pip, a)$ means that a is pulled directly by pdp from pip.
- $\texttt{arch}(pdp, pep, a)$ means that a is pushed from pep to pdp.
- $\texttt{arch}(pep, pip, a)$ means that a is pushed from pip to pep.

The solver assigns the truth values to the $\texttt{arch}()$ variables; these give the active connections of each component, and each set of assignments is a configuration.

We also use the notation $PIPs_a \subset PIPs$ (and $PEPs_a \subset PEPs$) as the set of PIPs (PEPs) such that $\texttt{provide}(pip, a)$ ($\texttt{provide}(pep, a)$, resp.). Similarly, $PDPs_a \subset PIPs$ is the set of PDPs such that $\texttt{needs}(pdp, a)$ is true. In the attribute retrieval model, we consider the following formulae:

$$\texttt{pull}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pip, a) | pip \in PIPs_a\}$$
$$\texttt{push}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pep, a) | pep \in PEPs_a\}$$

saying that an attribute a is pulled by (pushed to) the PDP pdp if and only if there is an arch between pdp and a PIP pip (PEP pep, resp.). Thus, if the pdp asks for a, and $\texttt{pull}(a, pdp)$ is true, then a must be retrieved directly from a pip.

We use the notation $\oplus\{v_1, v_2, \dots\}$ meaning that exactly one of $v_1, v_2, \dots$ is true, and $\ominus\{v_1, v_2, \dots\}$ meaning that at most one of $v_1, v_2, \dots$ is true. Encoding the possible paths and constraints means a conjunction of the following formulae:

- For each a required by pdp, exactly one connection must exist between pdp and either a PIP providing a or a PEP providing a. This is expressed by the following formula, for each $\texttt{needs}(pdp, a)$ in the Offline Input:

$$\texttt{needs}(pdp, a) \supset \oplus\{\texttt{arch}(pdp, x, a) | x \in PIPs_a \cup PEPs_a\} \tag{7.1}$$

  Moreover, in case there is an arch between pdp and pep, providing a, then there must be also a connection between pep and a pip providing a (formula (7.2)).

Otherwise (formula (7.3)) no arch between pep and pip providing a is necessary. For each $\texttt{needs}(pdp, a)$ and $pep \in PEPs_a$:

$$\texttt{arch}(pdp, pep, a) \supset \oplus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \tag{7.2}$$

$$\neg\,\texttt{arch}(pdp, pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \tag{7.3}$$

- If a pip does not provide a then no connection providing a can exist between pip and other components. For each $\texttt{provide}(pip, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\}$$
$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pep \in PEPs_a\}$$

  Similarly for the PEPs: for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$
$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

- To show the advantage of this approach, we consider another constraint: we suppose that each component can provide (e.g. for performance reasons) an attribute only to a finite number of components (in this case, one). To model that, for each $\texttt{provide}(pip, a)$ in the Offline Input, we consider the following:

$$\texttt{provide}(pip, a) \supset \ominus\{(\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\} \tag{7.4}$$

$$\texttt{provide}(pip, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pep \in PEPs_a\} \tag{7.5}$$

  Formula (7.4) (same for (7.5)) says that if pip provides a then there must be at most one connection between pip and the PDPs (PEPs, resp.) requesting a.

  It is similar for the PEPs, so for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$
$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

- Without data showing that PDPs cache attributes, we considered only PEPs to do that. This choice does not limit generality. We had to encode the constraint: if pdp asks for a, and $\texttt{not\_cacheable}(a, pdp)$ is true, then a cannot be retrieved from a pep that cached a. So for each $\texttt{not\_cacheable}(a, pdp)$ in the Offline Input, we have the conjunction:

$$\texttt{not\_cacheable}(a, pdp) \supset \bigwedge\{\neg(\texttt{cached}(a, pep) \wedge \texttt{arch}(pdp, pep, a)) | pep \in PEPs_a\}$$

- The attribute correlation constraint can be described as follows: if pdp has correlated attributes, then all its attributes must be either pulled or pushed. For each $\texttt{needs}(pdp, a)$ in the Offline Input, this can be modeled as follows:

$$\texttt{correlation}(pdp) \supset (\texttt{pull}(pdp) \vee \texttt{push}(pdp))$$

  where: $\texttt{pull}(pdp) \equiv \bigwedge\{\texttt{pull}(a, pdp)\}$, and $\texttt{push}(pdp) \equiv \bigwedge\{\texttt{push}(a, pdp)\}$

Notice that by defining PMAX-SAT problems it is also possible to impose constraints like: if possible, preference should be given to attribute provisioning via indirect paths, i.e., from PIP to PEP to PDP, over direct paths, i.e., from PIP to PDP. This can be obtained by maximising the number of truth values for the variables corresponding to the arches from PEPs to PDPs.

In satisfiability problems, coding the at-most-one operator $\ominus$ is often problematic. If there are $n > 1$ variables, an improved SAT encoding of $\ominus\{v_1, \ldots, v_n\}$ is the logarithmic bitwise encoding in [73]. This operator's CNF encoding requires $\lceil \log_2(n-1) \rceil$ auxiliary variables and $n\lceil \log_2(n-1) \rceil$ clauses. With this encoding, the CNF formula corresponding to (7.1) has a complexity (in terms of number of clauses) of

$$O(|PDPs| \cdot \mathrm{N_{a\_pdp}} \cdot (\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}}) \cdot \log_2(\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}}),$$

where $\mathrm{N_{a\_pdp}}$ is the average number of attributes requested by each PDP, and $\mathrm{N_{pips\_a}}$ ($\mathrm{N_{peps\_a}}$) is the average number of PIPs (PEPs, resp.) providing a specific attribute. These constraints generate a large number of clauses. This is why the encoding is computed offline and fed to the solver "on stand-by mode".

Runtime Input. The offline input abstracts a global view of the scenario. The runtime input specifies a current scenario, indicating the security components currently being used, their current set of attributes provided/needed, and any other current constraints. The runtime input is an instantiation of the offline input. For instance, if the Offline Input file contains $\texttt{provide}(\texttt{pip1}, \texttt{a1})$ and $\texttt{provide}(\texttt{pip2}, \texttt{a2})$, but in the current scenario pip1 is not available, then the Runtime Input will contain the following assignment: $\texttt{provide}(\texttt{pip1}, \texttt{a1}) = \texttt{false}$.

7.5.2 Constraint Solving without a SAT Solver

For independent constraints over components, an algorithm to choose configuration solutions may be faster than the SAT solver. Such an algorithm can perform a graph traversal to select the first PIP or PEP that satisfies the requirements of a PDP and some boolean constraints.

Algorithm 1 Pseudo-code for the constraint solving without a SAT solver.

Require: PDPset, PEPset, PIPset, ConstraintSet
Ensure: the first configuration to satisfy all constraints in the ConstraintSet

```
1:  SolutionSet = ∅
2:  for all pdp_i ∈ PDPset do
3:    Attribs = all attributes required by pdp_i
4:    for all a ∈ Attribs do
5:      while there are unvisited PEPs to provide a to pdp_i do
6:        pep_j = the first unvisited PEP to provide a to pdp_i
7:        if edge between pdp_i and pep_j exists and satisfies all ConstraintSet then
8:          SolutionSet = SolutionSet ∪ edge between pdp_i and pep_j
9:          break
10:       end if
11:     end while
12:     if no edge found with a PEP in previous step then
13:       while there are unvisited PIPs to provide a to pdp_i do
14:         pip_k = the first unvisited PIP to provide a to pdp_i
15:         if edge between pdp_i and pip_k exists and satisfies all ConstraintSet then
16:           SolutionSet = SolutionSet ∪ edge between pdp_i and pip_k
17:           break
18:         end if
19:       end while
20:     else
21:       return ∅
22:     end if
23:   end for
24: end for
25: return SolutionSet
```

Algorithm 1 sketches a possible way to obtain one configuration solution that satisfies a set of constraints. For every attribute required by every PDP (lines 4 and 2 respectively), the algorithm looks for an entity that provides this attribute, and if such entity exists, the edge between the entity and the current PDP is added to the SolutionSet. The priority of provision is given to PEPs in this case (line 6), and if no PEP is found, then a PIP is searched (line 13) to satisfy the constraints in the ConstraintSet. The search of the PEP or the PIP that provides the attribute continues until all of them are checked. Also, when the edge between the PDP and a provider exists (e.g., they are physically reachable, both ends support similar encryption schemes), then verifying if the edge satisfies all constraints involves checking that the endpoints of the edge have the same capabilities related to attribute handling.

However, topology traversal as in Algorithm 1 has several shortcomings. First, it

can only return the first solution that satisfies the given constraints, and not several other solutions that an administrator might choose from. It is not easy to adapt it to return all solutions. Second, it considers that the ConstraintSet is static; if the constraints change between the execution of two WHILE loops in lines 5 and 13, or if they change while the WHILE loop in line 7 is still running, then the algorithm like the one above cannot provide a correct solution. Third, the algorithm as above works best in scenarios where existing connections have no impact on choosing further connections. The traversal might become problematic when the topology is dynamic and local connections change global state, that in turn changes other local connections. This can happen, for example, when bandwidth restrictions limit the number of possible connections of a component; or when a doctor cannot perform surgery to the same patient unless after a defined period of time. Designing such an algorithm is not easy, since it is tightly bound to the constraints: whenever constraints appear or disappear, the algorithm needs to be changed. With a SAT solver, the set of constraints can vary without influencing the process of producing a solution, but performance might suffer.

Still, it is worth to design a better algorithm than Algorithm 1 as future work.

7.6 Performance Evaluation

The performance of our prototype depends on several aspects: the size and semantics of the offline input file, the size and semantics of the runtime input, and the internal performance of the SAT solver. Since each SAT solver uses a different solving technique, the performance of our prototype depends on the choice of solver. Apart from this dependency, we considered that the most relevant aspect to test is the encoding of the offline input, that need be recomputed several times along the lifetime of an application. This recomputing can be triggered by components that appear or disappear, or by changes in what components provide or require. The runtime input, on the other hand, is only a (moderately small) subset of the maximal offline one; its impact over the solve time depends on the technique used by each individual solver.

Therefore, we wanted to measure the offline CNF generation and the constraint solving time for a number of configuration topologies, while keeping the runtime input minimal. To obtain different offline inputs, there were several parameters to vary: the number of authorisation components, the number of attributes and constraints on attributes, the distribution of attributes per authorisation component. In choosing which of these parameters to vary and which to keep constant, we considered that the number of attributes is fixed in each scenario, since the administrator should

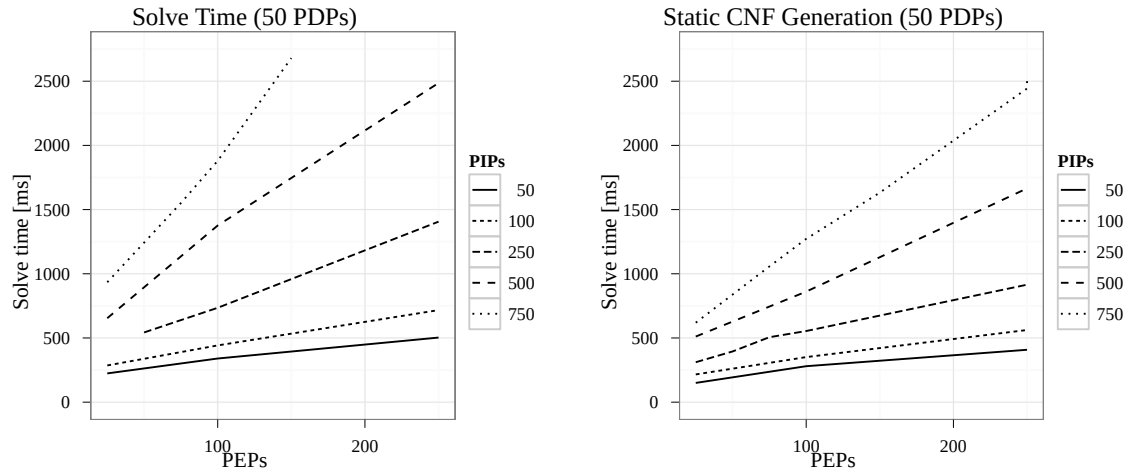


Figure 7.7: The time to solve our constraint problem with 50 PDPs, and varying numbers of PEPs and PIPs (left) and the time to generate the offline CNF clauses (right)

know beforehand the attributes required for all security decisions in a particular application. Hence, we generated over 100 offline input files to describe topologies with varying numbers of PIPs, PDPs, and PEPs, with a fixed number of attributes provided/needed by each. In order to assign attributes to each component, we implemented an algorithm for unbiased random selection as described by Knuth [123, Section 3.4.2].

To test our Java encoding against the SAT4J solver with the aims described above, we configured the JVM to run with 500 to 1000MB of memory, and we used JRE1.5.0. We ran each of the input files against a minimal runtime configuration file of one constraint, with the following parameter changes: 25, 50, 75 PDPs; 50, 100, 250, 500, 750 PIPs; 25, 50, 75, 100, 150, 200, 250 PEPs. The number of attributes was constant to 100, with 3 attributes per component in all cases. We measured the time (in milliseconds as per the Java call *System.currentTimeMillis*()) for the static CNF generation, as well as the time it takes SAT4J to compute the first solution from the already generated encoding (the solve time). The results for 50 PDPs can be seen in Figure 7.7; those for 25 and 75 PDPs are similar. The figures show a linear increase in the offline CNF generation in the number of authorisation components (Figure 7.7, right) with a similar linear increase in solve time (Figure 7.7, left). Figure 7.8 shows the difference between the total time and the static time, as a threshold for the time it takes to solve the runtime constraints.

These preliminary results are very encouraging in that the time taken to compute the constraints from the offline input is very short for a moderate application size (e.g., about 2.5 seconds for 750 PIPs, 250 PEPs and 50 or 75 PDPs), and also that they

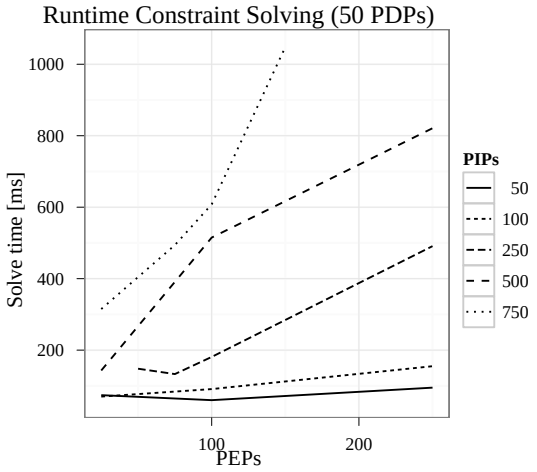


Figure 7.8: The difference between the total time to solve our constraint problem, and the time to generate the offline CNF clauses, with 50 PDPs, and varying numbers of PEPs and PIPs.

grow linearly with the number of components, for a fixed number of attributes. Even though satisfiability is generally known to be an NP-complete problem (and on a general case, it can be expected to obtain an exponential growth in problem complexity and hence performance), the linear relation that we have obtained is justified in that we are using a generic tool to solve a particular instance of a satisfiability problem.

7.7 Related Work

Some policy configuration features come with commercial products: Tivoli Access Manager [102], Axiomatics Policy Server [17], Layer 7 Policy Manager [134], Shibboleth [105] and Globus Toolkit 4 [3]. Tivoli provides central management and enforcement of data security policies across multiple PEPs, with partial policy replication and decision caching. Axiomatics offers central administration of policies, and can manage multiple PDP instances. Shibboleth focuses on federated identity management, allows for session caching and attribute backup, but does not consider attribute freshness. Globus considers resource and policy replication. All these products address different policy management aspects differently, and never consider together cache consistency, synchronisation and security guarantees. The users of these products cannot change such features from a single console, and hence cannot understand how one impacts another.

Two previous works are particularly relevant to our approach. First, Colombo et al. [40] observe that attribute updates may invalidate PDP's policy decisions. The

work only focuses on XACML's expressive limitations, does not consider system reconfigurations and is not supported by any implementation. Second, the work of Goovaerts et al. [87] focuses on matching of provisions in the authorisation infrastructure. The aim is to make the system that enforces a policy seamless and scalable when those components that provide security attributes change or disappear. They do not discuss how attribute provisioning impacts enforcement correctness.

Several other works are related to ours. Policy management is also tackled in Ponder [51] but aspects such as caching consistency and synchronisation are not considered. PERMIS [36] proposes an enforcement model for the Grid that uses centralised context data stores and PDP hierarchies with visibility restrictions over attributes provided across domains. Like us, Ioannidis [106] separates policy enforcement from its management, suggests a multi-layer architecture to enforce locally some part of a global policy, but does not discuss consistency of policy data at different locations and propagation of updates. The thesis of Wei [245] is the first to look at PDP latency in large distributed applications for role based access control. He does authorisation recycling by caching previous decisions, but does not consider cache staleness. Atluri and Gal [15] offer a formal framework for access control based on changeable data, but the main difference is that their work is on changing the authorisation process rather than configuring security components to manage such data appropriately.

7.8 Summary

This chapter considers the impact of properties about authorisation components and their connections, over the correctness of the enforcement process. We have looked at several properties of attribute handling, such as caching, retrieving and correlating related-attributes (i.e., treating them in the same way as one of them). We make the point that attributes are essential for security decisions, and handling them wrongly is a real problem that affects the correctness of policy enforcement.

This chapter concentrates on security attribute management. It proposes an approach to alleviate these configuration problems by managing them together. Our solution to this problems can help administrators in two ways: it can generate system configurations where a set of security constraints need always be satisfied (along with maximising performance constraints), or can check an existing configuration of the system against a given set of (security or performance) constraints. The configuration solutions can be recomputed at runtime and preliminary results show an overhead of a few seconds for a system with one thousand components. To our knowledge, this is the first tool to perform a fully verified authorisation system reconfiguration for a

setting whose security constraints would otherwise be impossible to verify manually. Such a system reconfiguration can be triggered as a correction to an observed security problem. For instance, if an indicator shows that there are too many timeouts on a security block performed by a certain security server, then adjusting the server's connections can trigger a system reconfiguration as described above.

Chapter 8

Related Work in Policy Enforcement

8.1 Introduction

This chapter surveys the methods that exist in practice to enforce security policies at runtime. Parts of this chapter have been published in a technical report [78]. We have analysed the different approaches of enforcement, and for our separation, we introduced two notions: an *enforcement technique* and an *enforcement implementation*. A technique is a general way of solving a problem, and an enforcement technique is a general way of enforcing a set of policies. Techniques that are not necessarily enforcement techniques, but are used in security implementations, are called by other researchers *security mechanisms*. An enforcement implementation instantiates an enforcement technique on a particular setting and target, and with a more particular set of tools.

8.2 Single-Machine Security Policy Enforcement

This section overviews the most important runtime enforcement techniques and their implementations. Since runtime program analysis has been active from early '90s, we have analysed over 100 academic papers published between 1993 and 2010 on techniques and implementations in runtime enforcement. The main criteria for this selection of papers were the earliest references, the most cited papers, and their presence in well-rated conference proceedings (IEEE S&P, USENIX, CCS, SOSP, NDSS, POPL, ACSAC, ASIACCS, PLDI, OSDI, ESORICS) and journals (ACM and IEEE Transactions). Some technical reports and theses are mentioned for completeness.

CRITERIA FOR ENFORCEMENT TECHNIQUES	CRITERIA FOR ENFORCEMENT IMPLEMENTATIONS
T0. OBJECTIVE **T1. ABSTRACTION LEVEL** **T2. LOCALITY** **T3. TYPE** T4. CLASS OF POLICY ENFORCED T5. GUARANTEES	I1. TRUST MODEL I2. POLICY LANGUAGE I3. OVERHEAD

Figure 8.1: The evaluation criteria for enforcement techniques and implementations. The criteria in bold are primary in our separation.

8.2.1 Criteria to Assess Techniques and Implementations

As shown in Fig. 8.1, we have compared enforcement techniques from three points of view: abstraction level, objective or type of policies enforced, and locality. Choosing these criteria was motivated by two viewpoints: (1) guarantees and type of enforcement that a techniques has, and (2) possible limitations or effort to implement each technique. Then, for the implementations of each technique, we looked at the policy language it uses, its trust model, and its performance overhead. Since every implementation measures performance in its own way, comparing overheads between very different implementations is a difficult task; our aim is to give an idea over an order of magnitude rather than precise numbers.

For an enforcement technique we look to evaluate the following characteristics:

T0. Objective The objective refers to what an enforcement technique protects: either the system from malicious targets , or protecting the flow of data that the target manipulates. In both cases, the untrusted program should not perform actions that are considered malicious by the policy, but the difference resides in the effect of such actions: whereas in the first case, the target corrupts the system, in the second case the target leaks data. The objective is tightly related to T4 - Class of policies enforced.

T1. Abstraction level The abstraction level refers to the *locus agendi* – scope, or location– of a technique is essential when analyzing security enforcement from a system point of view. The levels of the software stack where runtime enforcement can be located are shown in Fig. 8.2. There is a lower level – the operating system with system calls and memory management – and the higher level – the platform, runtime, and the application logic.

T2. Locality Locality refers to the size of the part of the program that is being analysed by a technique so that a policy is enforced [237]. For Vanoverberghe and

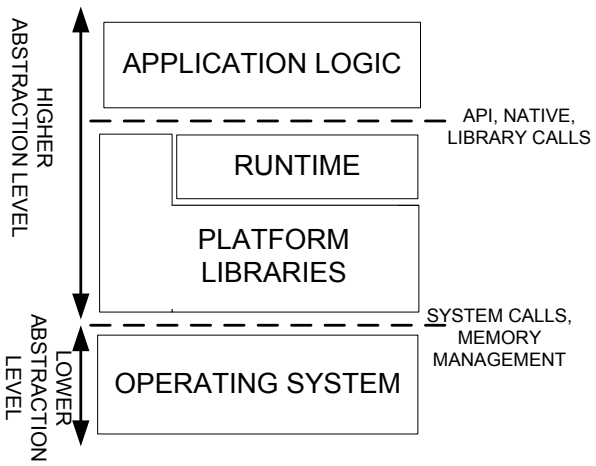


Figure 8.2: Abstraction levels in enforcement.

Piessens, the locality is one event when the technique performs checks around one single event in time; when the techniques examines multiple events, then the locality is a sequence of events.

T3. Type Type refers to the effect of enforcement onto the program's execution. There are implementations that just flag the presence of unwanted events, and others that transform programs such that their execution complies with the policy.

T4. Class of policies enforced The class of enforced policies refers to what type of policy a technique can enforce. Policies have been previously split in properties (safety, renewal, liveness, soundness) and non-properties (information flow) [140]. Since the implementations we found are enforcing either access control or information flow policies, we will refer to these policies.

T5. Guarantees Guarantees refer to the assurance with which the technique enacts constraints. This assurance can refer to the guarantees given by Anderson [6], that cover complete mediation, tamperproofness and verifyability.

Most enforcement techniques use a mechanism called *interception* (or interposition) in order to detect policy relevant events (service or resource requests together and parameters). The relevant event is blocked until a decision and an action are performed, and based on the type of action, we separate between implementations whose security monitor stops executing when malicious behaviour is detected, and implementations whose monitor continues to execute. In the first case, the consequence of the stopping of the monitor can be that the monitored program is halted, or that it continues to run

but throws an exception or updates a security log. We call these implementations *recognizers*. On the other hand, there are also those implementations that transform the malicious target into a compliant one; we call them *sanitizers*. The security monitor, in this case, continues to execute along with the target program, but suppresses or modifies the malicious actions before they happen. It can be argued that recognizers are a subclass of sanitizers where the transformation operations are absent, but the choice to separate them is motivated by a similar distinction between traditional security automata – also known as execution recognizers – and edit automata [140] – also known as execution transformers.

Separating policies in categories, or classes, has been mentioned in a formal context. Schneider showed that not all policies are enforceable and security automata can only enforce safety properties; Hamlen showed there are several classes of enforceable properties (and policies) [95]: some that are statically enforceable, some that are runtime enforceable, some that can be enforced by runtime program rewriting, and some others that are not. Ligatti et al.[142] argued that Schneider's execution monitor, now called *recognizer*, can be more realistically extended into a monitor that can insert and suppress actions in order to correct the target program if it misbehaves. Also, while Bauer, Ligatti and Walker have classified security policies by the computational resources that are available to the execution monitor [22], Fong suggested a classification of policies from the type of information observed by the monitor, more specifically the shallow access history [68]. Fong's work is useful in practice in that it studies the impact of the limited memory onto the model of automata in enforcement.

To our knowledge, apart from these (rather few) characterisations of properties, there has been no further work in classifying types of security policies from a practical perspective. We take on the initial categories sketched by Schneider – *access control policies* and *information flow* policies, and also consider the usage control extension presented in the UCON model [185]. *Access control policies* cover the constraints on a target accessing a system resource. *Usage control policies* augment access constraints with continuous checking, attribute updates as a result of the usage of the accessed resource (i.e., state), as well as with the notions of obligations (i.e., commitments in the future) [185]. The last class of policies is *information flow policies*, that aim to prevent the leakage of data to unwanted entities that can hence reason about program behaviour.

The features of a technique extend over the concrete implementations of the technique: abstraction level, type, locality, guarantees, objective / class of policies enforced. Still, there are some aspects of individual enforcement implementations that give a more concrete idea over the technique they implement: (I1). trust model and components; (I2). policy language; (I3). performance overhead. These aspects are

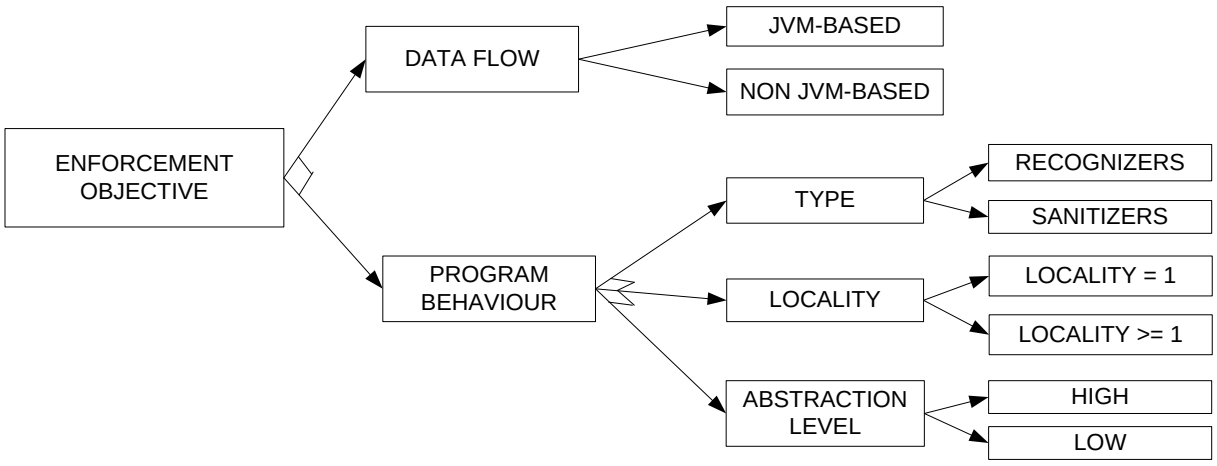


Figure 8.3: Two orthogonal criteria for classifying enforcement depending on policy objective: techniques enforcing policies on data flow, and techniques enforcing policies on program behaviour (with subsequent separation by type, locality and abstraction level).

tightly related to the technique: the trusted components, as well as the overhead, depend on the technique's abstraction level; the policy language is tightly connected with the type of technique, and with the class of policies that need be enforced. Figure 8.1 shows these aspects together.

I1. Trust model. Each implementation is defined by the system entities it trusts, what assets need to be protected and sometimes, where threats come from. To this respect, we will assess existing enforcement techniques on the quantity and manner in which they trust external components.

I2. Policy language. A security mechanism should not only be efficient but also usable by security researchers or developers. In this sense, it is important to assess the ease with which a user can write policies for a particular policy enforcement tool. This aspect involves language expressiveness and ease of use.

I3. Performance overheads. Whenever runtime mechanisms are discussed, their impact on the overall application performance is important. In comparison to static enforcement, runtime enforcement technique implementations bear the risk of burdening the execution of the target every time they are used. It is generally very difficult to measure accurately the performance overhead of an implementation against another simply because the experiments use different assumptions and testbeds.

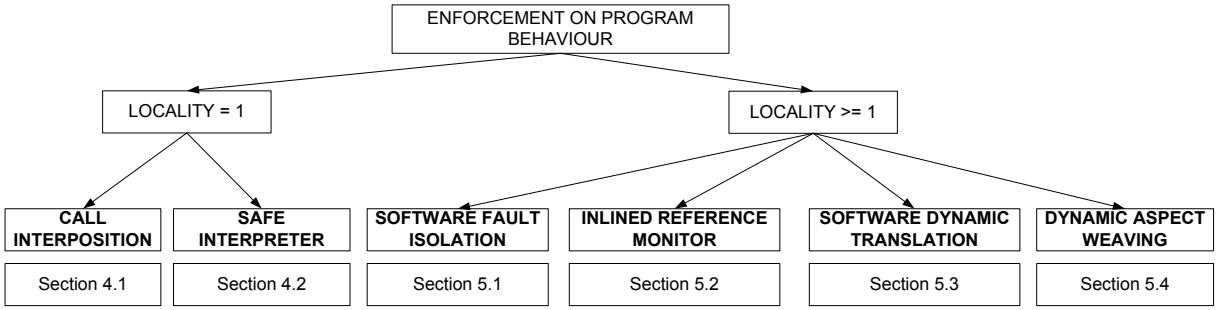


Figure 8.4: Taxonomy for runtime enforcement techniques on program behaviour and locality.

8.2.2 Taxonomy of Enforcement Techniques

There are several orthogonal ways by which techniques can be analysed using the criteria described in the previous section. Figure 8.3 and Figure 8.4 show several ways to evaluate enforcement techniques. Figure 8.3 shows that techniques can be split by their locality, their type and objective. From the point of view of the objective (or class of policy enforced), there are two types of techniques: (1) those that enforce constraints on program behaviour independent of the data that the program handles, and (2) those that enforce constraints on data handled by the program, independent of program flow. The former aim to constrain possibly malicious code from damaging the system, and are associated with *sandboxing* – a generic approach in security that aims to minimise the effects of the untrusted program over the system. These techniques can be further separated by type (either sanitizers or recognizers), by locality (either one of greater or equal to one), and by the coarse abstraction level shown in Figure 8.2 (high and low). The latter kind of techniques focus on data propagation. We see policies on data flow orthogonal to policies on program behaviour. Data flow techniques are usually sanitizers with locality greater or equal to one: information flow is a policy over several executions rather than an individual one and so in order to enforce information flow, a sequence of events is analysed at a time. They can further split into JVM-based approaches and non-JVM approaches.

Figure 8.4 looks closer at enforcers on program behaviour from the point of view of locality[1]. Locality is one for techniques that do interception and interpretation (since these execute per call or per instruction), and at least one for the other techniques. Call interposition is the enforcement technique that explicitly monitors the occurrence of certain calls – specified by the security policy – and either blocks or changes them.

[1]The techniques in Figure 8.4 might have overlapping implementations or similar mechanisms. Our taxonomy is based on general features presented in the initial papers.

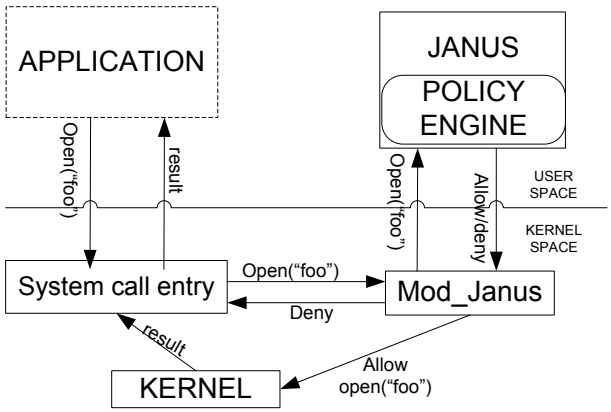


Figure 8.5: The Janus architecture

Safe interpreters are similar in that they execute (or interpret) one instruction or command at a time. Conversely, the techniques with higher locality are related to program rewriting: software fault isolation changes memory addresses in object or assembler code in order to prevent reads, writes, and branches to memory locations outside a policy-specified region (known as a safety domain). Software fault isolation techniques are source language independent. Inline reference monitors (IRMs) usually focus on events closer to the application: method calls and returns, thread creation, and specific security events. IRMs work on intermediate code and hence are specific to runtimes like JVM and .NET. Software dynamic translation (SDT) for security is a more generic technique that translates from compiled to binary code at runtime, sometimes with the help of an interpreter. The added difference to interpreters is that SDT does not focus on the execution as much as on the safe translation of chunks of code into a safe version of executable code. These techniques are presented in more detail in the sections indicated in Figure 8.4.

8.2.3 Call Interposition Techniques

Call interposition techniques exist at the system call level, but also at higher levels. They can be either sanitizers or recognizers. Hereafter we focus on system call techniques for two reasons: first, the bulk of the work in the state of the art concentrates on system call research, and second, a big part of the discussion on enforcement with system call interposition applies for calls at higher levels of abstraction.

System call interposition is motivated by a large class of software attacks to boil down to the system calls made to access the file system, network, or other sensitive resources. Therefore, monitoring relevant system calls makes it possible to detect and counteract a broad range of malicious behaviours. Tracing system calls at runtime

provides a rich amount of data: method call chains, method arguments, return values. The costs of this approach depend on where interception mechanisms reside on the host OS – either kernel-level or user-level mechanisms, and whether they block or alter the call chain.

System call interposition helps enforce access control policies and limit the domain of action for untrusted programs. System call recognizers cover both kernel-level and user-level areas. This has a big impact on two important aspects: portability and performance. Kernel-based sandboxes can become difficult to port, since they depend on a precise kernel structure and content. Policies on intercepting system calls, either for kernel or user-space enforcers, ask for detailed knowledge on OS components, therefore are costly and difficult to develop. Enforcing such policies is also error prone when it comes to replicating OS state, races, symbolic links and related issues [74].

Higher level call interposition has the advantage that is closer to the developer's and user's understanding of the target application. It is much easier to write security constraints at a level closer to the application than to the core system. While the same system call sequence can be exhibited by two applications, a sequence of method calls is more likely to be specific to one target application. The security policy writer needs to have application-specific knowledge of the API or method calls to monitor – and this was not the case for system call level policies that are less dependent of the application that generates the calls, but more dependent on the OS used. Two well-known examples of high-level call interceptor features are Enterprise Java Beans in J2EE [177], and Apache Tomcat [10] servlets. High level call interposition implementations exist in .NET [236] and Java [88].

System call interposition is prone to tampering and bypassability problems: a monitored process cannot have more than one single monitor [65]. If a malicious user creates its own monitoring process and attaches it to any other (as a traced process), then the user would gain full control over the monitoree; this happens because the monitor is not necessarily trusted – it can also be user-defined.

Also, Garfinkel [74] observes that *race conditions* are the most frequent problems to occur with system call interposition. A race condition happens when parameters of a system call can change between the time they are checked by a security tool, and the moment they are retrieved by the OS to perform the system call (Time-of-Check-Time-of-Use or TCTU race). The cause is generally multi-threading or shared memory between threads, so that the shared states between operations performed by system calls can be altered. Worse, Garfinkel also shows that denying system calls may have side effects e.g., security flaws due to privilege dropping, or even undermining program reliability. These problems apply to higher-level call interposition as well.

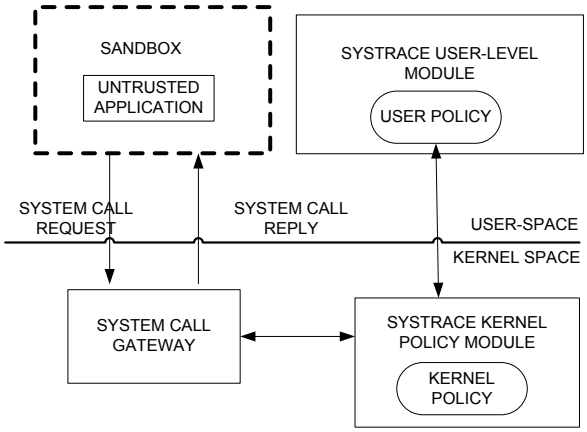


Figure 8.6: The Systrace architecture

System Call Recognizer Implementations

System call recognizer implementations have been embedded in operating systems for a long time. For example, the command `strace`, the `ptrace` system call or the `/proc` virtual file system in Unix are mediator processes that can offer system call recording and filtering.

Another method of system call interception is using *call wrapping*. A wrapper can be used to trap specific calls and attach extra features to them. For instance, COLA [126] works by replacing user-space dynamic library calls with their user-customised versions, but since the aim is functionality (e.g., user notifications when system calls happen) rather than security, COLA trusts the user and is ineffective against programs that use system call instructions directly. Unlike COLA, that does not modify the kernel, approaches like SLIC [84] are more secure for system call interposition in that they are protected from malicious applications.

An example of a user-space system call filter is Janus [85] (see Fig. 8.5). Its focus is to confine untrusted applications by intercepting and filtering malicious system calls performed by ordinary users. Janus has a *mod_janus* kernel module that performs system call interposition, and a *janus* part that, given a policy specified by a user , decides which systems calls to allow. In case of a policy violation, Janus kills the monitored program. All processes spawned by the target are sandboxed as their parents.

A similar idea but implemented at kernel-level is presented in TRON [26]. TRON offers a discretionary access control for a single process by providing protection domains (or sandboxes) for untrusted processes: each process executes in exactly one protection domain that it cannot escape. The enforcement is done by a kernel al-

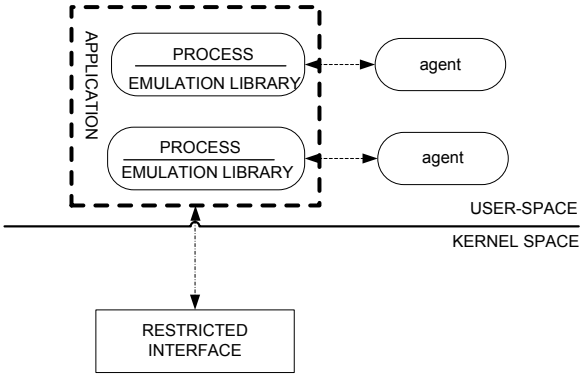


Figure 8.7: The Ostia architecture.

gorithm which examines a domain's access rights to access a specific routine; if the routine is not found in a domain or if the domain does not have the needed access rights for that routine, a violation handler is invoked. The same approach is taken by ChakraVyuha [46] and SubDomain [41]: limiting the privileges of target code over system resources extends into enforcing mandatory access control for all system users.

System Call Sanitizer Implementations

System call sanitizers can change the behaviour of the intercepted system call by modifying the semantics of the call or its parameters. At user-level, system call modifying by wrapping is proposed by Jain and Sekar [111]. They improve the original Janus with Janus2, or J2, by reducing the number of context switches and dealing more flexibly with system calls: their solution also permits parameter modification, apart from simply system call accepting or denying.

We have found mostly hybrid system call implementations, i.e. that include both user and kernel level components. Hybrid architectures run simultaneously both kernel-level and user-level mechanisms of enforcement. They offer the combined advantages of the two approaches: the speed of the kernel-level mechanisms and the portability and ease of development of the user-level ones. Sandboxes at this level have two separate components: an interposition architecture provides access to system call data and enforces policies typically within the kernel, and a policy engine that decides on the resources the sandbox in user-space.

In Systrace [196], for instance, a kernel-level mechanism intercepts the system calls of a monitored process, and informs a user-level monitor about them. Systrace generates policies interactively or, more interestingly, automatically by means of a training session. The kernel policy module interposes when a system call is about to be started or finished, and retrieves information about it. If it cannot decide by itself, it asks a

Criterion	System call implementations
Abstraction level	Operating system
Guarantees	mediation (full if in kernel), tamper-proof
Trusted components	OS, system call wrappers, libraries
Policy class	access control
Policy language	very low level, not user-friendly
Overheads	very low (ms), depends on no.system calls

Table 8.1: Summary on system call interception techniques and implementations.

user-level daemon for a policy decision. This daemon monitors processes and has a more extensive view over the system. The Systrace architecture is shown in Figure 8.6.

Ostia [75] follows a slightly different scheme compared to Systrace and some later Janus implementations (J2 [74]). In order to invoke sensitive system resources, a typical system sends the requests of the sandboxed application to the kernel. Ostia replaces this path with a delegation of responsibility to the *sandbox broker* (see Figure 8.7). This broker mediates resources on behalf of the application, on the terms specified by a user policy. The TCTU race noticed before [74] is solved: while in Systrace-like architectures (approach also known as *filtering architecture*), permission checking is performed by the application sandbox and the access granting comes from the OS, in Ostia the user-level sandbox gets complete control to access resources (approach also known as *brokering architecture*).

Assessment over Call Interposition Implementations

Higher level interposition implementations are not as frequently discussed in the literature as system-call interposition ones. The implications of intercepting higher level or lower level program events are discussed in [236, 54]. Erlingsson and Vanoverberghe mention that a big difference resides in the expressiveness of the policies: system-call level techniques are fine grained and their policies are expressive yet difficult to understand; application-level enforcement is coarser grained but the policies, albeit less expressive, are simpler to read and understand within the practical context of the application.

In terms of overhead, system call enforcement implementations can be fast at OS level. For instance, Janus built on SLIC's interposition mechanisms adds one extra content-switch on each potentially malicious system call, to SLIC's one-time installation cost of 2-8 microseconds plus the time for an extra procedure call. TRON incurs hundreds of microseconds that will depend the number of policies.

In terms of security guarantees and trust model, system call interposition imple-

mentations offer tamperproofness and complete mediation if they are kernel based but not if they are user-level sandboxes. Wrapping in the second case is circumventable – a piece of code could anytime bypass a trapped call by invoking a machine-level instruction – and therefore it must be seconded by some mechanisms that prohibit subversion. The TCB includes the OS and the call wrappers. Table 8.1 summarises system call interception implementations.

8.2.4 Safe Interpreters

A software interpreter introduces a virtual layer mediating interactions between a running program and the CPU. From a security standpoint, the biggest advantage of this approach is that untrusted programs cannot directly reach system resources: all instructions have to pass through the interpreting mechanism which can perform security checks before translating and executing the respective instructions.

Scripting languages come with their own interpreters and are highly portable since the input code is translated in the same way on any platform or environment where the interpreter is installed. For this reason, languages like JavaScript, ActiveX and VBScript are popular browser security research topics. Scripts in these interpreted languages being frequently embedded in Web pages, accessing such a page leads to the download and execution of the script by the interpreter on the user's system. This can be dangerous, hence it is important to keep such execution contained on the client side. In 1998, the notion of *safe interpreters* was proposed, with the primary aim of containing the effects of the execution of untrusted scripts [9, 179]. By Anupam and Mayer, a safe interpreter needs to ensure two things: 1) data security in the sense of confidentiality and integrity, and 2) user data privacy [9]. To achieve that, the interpreter needs to isolate scripts that might execute unsafe commands (e.g., I/O, cd, execute, open), and consequently to enforce access control for the objects in the context of the scripts. Security policies can be, therefore, about object access control, independence of contexts of various scripts, and management of trust between them.

Interpreters enforce access control policies by reacting to program behaviour. The level of abstraction is high since scripts and interpreted languages are programs directly written by human users, and the policies onto such program behaviour are easy to understand by a security developer. The guarantees brought by safe interpreters are tamperproofness and mediation – but there are documented cases when the interpreter can be bypassed [9, 254].

Safe Interpreter Implementations

A well-known scripting language –Tcl– was extended with security features to Safe-Tcl [179]. Safe-Tcl employs a technique by which untrusted programs (called applets) are executed in spaces separated from the trusted system core. The applets cannot interact directly with the calling application (like an kernel space, in an OS analogy), and are kept in a sandbox of the *safe interpreter*. Safe-Tcl has two interpreters: an untrusted one called 'safe', and a trusted one called 'master'. In order to use resources, the applet needs to use 'aliases', which are commands to the master interpreter, that guards system resources and decides to allow or deny alias requests. An alias is thus a mapping between a command in the safe interpreter and a command in the master interpreter. The master interpreter has complete control not only on the states and execution of the safe interpreter, but also on how aliases are called from the applet. To this end, the Safe-Tcl security policy is a set of Tcl scripts implementing the aliases that the policy allows. The applet is completely separated from the policies, and has the power to choose at most one security policy, as an alternative to a policy-less execution when it is only allowed to perform safe commands. Safe-Tcl does not associate the 'right' policy with the 'right' applet: untrusted applications are allowed to choose between policies and this freedom is not necessary even if the policy writer is trusted: an application can choose a less restrictive policy than it should. The Safe-Tcl policies we saw (e.g, limit socket access, the number of temporary files) require blocking of unsafe actions, and hence we derive that Safe-Tcl is a recognizer implementation.

Another approach for access control enforcement with interpreters is that of SecureJS, proposed by Anupam and Mayer [9]. Unlike Safe-Tcl, where there is a master interpreter and restricted children interpreters, SecureJS focuses on the interfaces between the script and the browser, or the script and external entities like Java applets and ActiveX scripts; its focus is to restrict the external interfaces and their methods that involve scripts. It follows that this implementation's threat sources are external browser entities, unsafe scripts, while the browser and interpreter are trusted. Similar to Safe-Tcl, Secure JS concentrates on separating between namespaces with different script objects and restrictions, read-only and writable objects within a namespace, and trust shared between namespaces for object reuse (hence the TCB can include high-level objects). We see SecureJS as a recognizer: it either allows the calls between these entities to happen, aborts the call, or asks the user for permission to allow the call. Anupam and Mayer mention that policies are formulated as access control lists, but SecureJS does not come with performance overhead evaluation since the target is to repel a set of browser attacks.

Other JavaScript enforcement implementations for browser security are CoreScript [254],

Criterion	Safe interpreter implementations
Abstraction level	Application level
Guarantees	tamper-proof, but not non-bypassability
Trusted components	browser, helper modules,interpreter
Policy class	access control
Policy language	scripting languages, or customised
Overheads	from 1 to 25% (peaks at 200%)

Table 8.2: Summary on safe interpreter techniques and implementations.

ConScript [149] and that by Devriese and Piessens [49]. While the first two are built to enforce access control policies, [49] looks at non-interference. CoreScript inserts security checks and warnings in the Java Script code that the interpreter will reliably execute. CoreScript's model is that of a sanitizer since it can also change the semantic of the original code, and expresses its policies as edit automata. Its TCB includes a rewriting module, a policy module, and a special callback module that allows for further security validation and rewriting on runtime-generated scripts. ConScript focuses on policy safety in an adversarial environment by disallowing protected objects to flow to user code, and protecting the integrity of access paths when invoking a function; its purpose is to automatically produce expressive policies to protect a hosting Web page from malicious third-party code and libraries. It is a recognizer implementation since, from the 17 policy examples given, the action is to restrict or limit script functionality so that just allowed behaviour happens, rather than to correct possibly malicious actions.

Assessment over Safe Interpreters

An assessment over interpreters is shown in Table 8.2. These interpreter implementations bring mediation and tamperproofness guarantees, but non-bypassability does not always hold. As far as the security policy language is concerned, interpreters like Safe-Tcl require minute system knowledge ranging from system call API and kernel structures, to proprietary low-level APIs. The policies are expressing events at the same (high) level of abstraction with that of the mechanism enforcing the policy: the language refers to individual calls. The TCB includes the interpreter and various other callback/rewriting modules in the implementation designed to improve on non-bypassability, or to counteract certain attacks.

Safe interpreters perform worse than system call interception because of the interpretation overhead. Interpretation has a much lower performance than system call interception e.g., it raises overhead from 2 to 10% more than for compiled code. This is

the main reason why interpreters are not used in complex applications. Nevertheless, the performance overhead of instrumenting interpreters for browser security seems to be much smaller – ConScript's overhead is around 1%, apart from an initialization overhead of tens of microseconds [149]. For the multi-execution technique in [49], the combination of interpreter and either serial or parallel multi-execution give execution overheads varying between 25 and 200%.

8.2.5 Software Fault Isolation

Software fault isolation (SFI) was introduced by Wahbe et al. [243] and offers low-level code safety. It is a type of software address sandboxing: addresses are changed so that they fall in a specific memory region. When an application is allowed to dynamically load untrusted components, or modules, it is important that these modules do not corrupt the host system, hence the need to isolate such modules within *fault modules*. The SFI model is an encapsulation by which untrusted object code cannot operate on memory addresses different than its own range. All non-CPU system resources are accessed by system calls, so providing wrappers for the system calls to which the untrusted code can be redirected should be enough to contain malicious effects. Small observes in [216] that SFI techniques can be implemented in several ways: in a compiler pass [215], a filter between the compiler and the assembler [216], or as a binary editing tool like [243]. Managing such software-enforced spaces does not require maintaining separate address spaces, because they target a single Unix process. SFI acts at load time rather than runtime. Schneider argues that SFI is not part of the execution monitoring mechanisms because, unlike them, it modifies the target before it is actually run. Nevertheless, the instructions that SFI inserts can implement a security automaton. SFI techniques are focused at enforcing low-level access control of untrusted code to the underlying system.

From a security standpoint, SFI techniques can enforce very fine grained memory access. They also help in what is known as *control flow enforcement*. This notion relates to a known problem when executing third-party code: illegal code transfers. Illegal code transfers refer to situations when, instead of giving control to legitimate code, the system yields control to malicious code. This behaviour is observed with e.g., return-to-libc attacks[2]; to counteract it, a security mechanism has to put restrictions on where the program counter points to before the next instruction is executed.

[2]This is a buffer overflow attack that replaces the return address on the stack with the address of another instruction, usually a call in the `libc` library.

SFI Implementations

The original SFI implementation of Wahbe et al. [243] offers a user-level approach to ensuring code safety and memory protection. First, it loads the untrusted code in a designated memory space. What it monitors is any low-level instruction that jumps or stores an address outside its memory area, for example procedure returns (represented as jumps to registers). A part of the target application's address space is logically separated from the rest of the space and called 'fault-domain'. Software modules placed in different fault domains can only communicate via explicit Remote Procedure Call (RPC) calls. In order not to create any resource conflicts, the OS is modified in order to know of the existence of fault domains and moreover, hardware page tables are also programmed to securely leverage data sharing between fault domains.

There are several other SFI implementations among which we noted Naccio [64], MiSFIT [216], Omniware [145]. Naccio comes in two flavours for the platform library: Naccio for Win32 and Naccio for the JavaVM. Naccio/Win32 automates the process of wrapper writing and modifies the target program too. Its abstraction level is still system calls, but the focus is on wrapping the entire platform interface. Given some resource (resources here include files, threads, network connections) descriptions and some constraints on resource usage, a policy generator automatically creates an abstract policy suited for a particular platform and purpose, together with a modified version of a platform library. Hooks are added to the target application to call the policy-enforcing library and the desired policy file, instead of the original system call entries. Omniware [145] targets mobile code safety; it is a portable virtual machine whose compiler generates portable code for an abstract virtual machine, and then translates it into native fault-isolated code at runtime. In its turn, MiSFIT [216] addresses loads, jumps, and stores, but cannot ensure protection against illegal reads. MiSFIT stands between compiler and assembler; it fault-isolates loads, stores and calls and it processes unsafe instructions by rejecting the program module, or transforming unsafe operations in safe ones. MiSFIT is hence both a recognizer and sanitizer.

For control flow enforcement, DynamoRIO focuses on dynamic code optimization and adds security by proposing the technique of *program shepherding* [122]. Program shepherding restricts execution privileges for untrusted code and also restricts program control flow. It monitors control flow transfers at runtime, by either (1) code origin, (2) instruction class, source or target. Similarly, the RIO dynamic optimizer starts off from an interpreter, and aims to achieve uncircumventable sandboxing by blocking invalid control transfers. Program shepherding is reported to prevent violating code to execute, and hence we classify it as a recognizer implementation. Further

Criteria	SFI implementations
Abstraction	OS and platform
Guarantees	nonbypassability, tamproofness
Trust components	OS, rewriter, compiler
Policy class	access control
Policy language	high-level (Naccio) otherwise very low level
Overheads	low to medium, e.g., 9-45%

Table 8.3: Summary on some SFI techniques and implementations

SFI implementations for control flow enforcement are NativeClient [252], and *control flow integrity*(CFI) [2]. CFI provides fine-grained integrity of the program control flow, by a mixture of static verification (that computes a flow graph), binary rewriting (that inserts runtime checks in the machine-code) and some runtime checks. CFI couples well with IRMs: CFI ensures that runtime execution follows a given control graph, and this is useful for IRM to make sure the extra checks they insert in the program will be surely executed, and the security state updated. Also, CFI ensures safe regions in memory where the state keeping of the IRM can be done. While NativeClient processes x86 code to guarantees that the control flow will target just certain address ranges, CFI can restrict the flow to any address within the control flow graph. In this way, CFI efficiently blocks attacks to control-flow transfer, but cannot protect against attacks complying with the legal control flow graph (e.g., changing arguments of system calls). To solve this problem, XFI [56] builds on CFI and relies on IRM load-time verification to ensure memory access control; hence it guarantees environment integrity. We see these technique implementations as recognizers because of their blocking behaviour towards malicious code.

Assessment over SFI Implementations

Naccio cannot enforce constrained memory accesses and code structure constraints. Its focus is on low-level safety rather than monitoring resource operations, but memory and processor usage constraints cannot be enforced. Naccio is a recognizer since it does not influence the actual behaviour of the program.

SFI implementations ensure integrity of the rewritten code, but can be circumventable: because they do not impose conditions on the flow of the program, the target can find ways of avoiding the checks of the rewriter [2]. The trusted computing base for SFI rewriters usually includes the rewriter mechanism, the compiler and the OS. Table 8.3 summarises some features of SFI implementations.

As far as the security policy language is concerned, most of the implementations of

SFI requires detailed and very technical knowledge of the inner system. The language refers to singular calls like loads and stores (e.g., Naccio) or groups of calls to monitor (e.g., patterns similar to buffer overflow or return-to-libc). Naccio has the problem that if the policy changes, the policy generator needs to be run again to produce an adapted version of the policy-enforcing platform library. The specification language is easy to use, because it is aimed at being accessible and platform independent. Also, Naccio's and Ariel's enforcers are triggered by single specific calls [181], loads and stores. The policies required for SFI instrumentation are very low-level.

The overheads of these implementations are not very big. This is because SFI creates logical fault domains in the context of one address space, and the remote procedure calls (RPC) among these domains are fast. Omniware reports that sandboxed code runs just 9% slower than its non-sandboxed version. MiSFIT states that its read-write-call protection takes from 1.4 to 3.2 times more to execute than unprotected code [216]. The performance analysis of Naccio for JVM shows to perform better than JDK's security manager on short policies (i.e., with 84% rather than JDK's 153%), but for complex ones (e.g. limiting the number of bytes to be written or read from a location) the overheads grow dramatically due to managing resource objects in memory. Also, on SPEC2000 benchmarks, CFI is reported to have a 15% overhead compared to NativeClient's 5% overhead [252], while XFI, adding data sandboxing to CFI, should in theory be slower. Also, CFI is shown to perform faster than Program Shepherding, and on the SPEC2000 benchmarks the overhead can reach 45% [2].

8.2.6 The Inlined Reference Monitor

The Inlined Reference Monitors (IRMs) combine, or *inline*, execution monitoring (defined above) with the untrusted program (Fig. 8.8). They can act either as sanitizers or recognizers. An IRM makes use of a trusted rewriter tool that inserts security code in the target application in order to prevent any access control violations [55]. For that, it needs three types of information [57]:

- the security events are operations declared sensitive by the security policy: API calls on files, system calls, socket communication, etc.
- the security state refers to any kind of context information that the policy logic requires in order to make a decision (for example, some information from the execution history).
- the security update refers to what action the program should take whenever the security event happens, as an update to a security state. The action can be

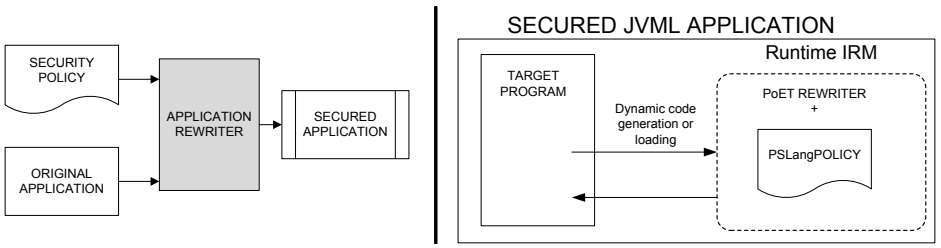


Figure 8.8: The inlining process on the left-hand side. The rewriter takes the application code and a security policy as input, and produces a secured version of the application. On the right-hand side, an implementation of this process with the Policy Enforcement Toolkit IRM.

anything from a policy violation signal to a set of remedial actions (e.g., disable all socket communication).

IRMs can be recognizers and sanitizers as well. The enforcement capabilities of security automata evolved from initially halting the execution of the target to operation suppressing or injecting. Alleviating the draconian behaviour of the original security automata are the edit automata [140] that were introduced later by Ligatti et al. These new automata can truncate, insert and modify, if needed, the execution of the target. Notice here that program rewriters modify programs, while edit automata can modify steps in an execution. Introducing edit automata has led the way to more powerful monitors: edit automata can 'pretend' they allow the target to execute until the monitor eventually accepts it as legal (in other words, a sequence of untrusted actions is suppressed and only if it is proved not to violate any policy, it is reinserted in the program flow). As a result, further series of extensions have been brought to Schneider's preliminary work [95, 208, 141], by linking it to computational complexity and extending it to non-safety policy enforcement. Some notable conclusions are that intuitively, if a policy is enforceable at runtime then it is enforceable by an IRM; most statically enforceable policies are enforceable at execution time; some non-safety properties can also be enforced at runtime, depending on the computational capabilities of the security automaton modeling that execution.

IRM Implementations

One of the first implementations to insert or inline monitoring code in the target program was SASI [54]. Security Automata SFI Implementation (SASI) extends the original SFI to execution monitoring – and to enforcement of any policy that can be expressed as an automaton. SASI embeds the policy enforcer in the executable code (x86 assembly language and Java Virtual Machine Language). The idea is that any memory-access instruction is considered sensitive, so security checks are inserted

right before this instruction is executed on the target machine. These checks are non-circumventable, ensure memory safety and eliminate calls outside of the program. A rewriter module is in charge with merging of the original bytecode with the security automaton of the policy. The inserted code causes the application to halt when the automaton rejects the input, hence SASI is for us a recognizer. Because SASI can describe policies at a lower level than Naccio, it can enforce policies that Naccio cannot: constrained memory accesses, code structure constraints. Conversely, Naccio can enforce policies that SASI cannot, since it modifies the behaviour of the program. Naccio's focus is on low-level safety rather than monitoring resource operations, but memory and processor usage constraints cannot be enforced.

Erlingson and Schneider introduced the IRM [207, 55, 57, 208, 95] but there are important contributions on applying or extending IRMs [236, 217, 22]. Erlingsson emphasised the use of automata with strongly-typed languages like Java and .NET [57, 208, 55]. He observed that the runtime checks performed by a Java 2 JVM consider stack inspection policies [57], while this does not hold for JVMs before Java 2. In other words, more sophisticated security policies can only be run on new JVM versions; conversely, policies not related to stack inspection could not be run on latest JVMs which do just stack inspection. The solution would be to use an inlined reference monitor to perform stack inspection, so that changing security policies would not imply changing the JVM. The suggested solution implements a Java bytecode IRM called Policy Enforcement Toolkit (PoET). The security policies for PoET are specified using a Java-like policy language: Policy Specification Language (PSLang). The integrity of the overall reference monitor is guaranteed by the Java type safety, while the implementation is two-fold: there is a security-passing style IRM and a "lazy IRM". The former uses a variable that stores security information from the runtime stack, and its security triggers are: method calls and returns, thread creation, Java permission checking and privileged blocks. The latter does not consider method calls and returns anymore, but manipulates the Java runtime stack and adds thread creator methods and permission checking in privileged code blocks. The gains of these implementations reside in very good performance and flexibility to do enforcement on any application event, irrespective of the JVM version.

Polymer is an approach that focuses more on policy writing and policy composition [23]. One of its novelties is that policies as Java methods that suggest how to handle trigger actions, and what to do when the suggested actions are followed. The design is similar to Naccio in that there is trusted policy compiler that compiles RMs defined in Polymer into Java bytecode, as well as a bytecode rewriter that instruments the target code according to the monitors. Polymer has the notion of a policy with

Criteria	IRM implementations
Abstraction	application and platform
Guarantees	mediation, integrity, tamproofness
Trust components	rewriter, policy compiler
Policy class	access control
Policy language	security automata or Java-like
Overheads	low to medium, e.g., 0.1-30%

Table 8.4: Summary on some IRM techniques and implementations

other policies as parameters – called *a policy combinator*– and the syntax allows for conjunction combinators, precedence combinators, and selectors to select only one of their subpolicies. Polymer's TCB includes the policy compiler, the bytecode rewriter, and custom JVM class loaders. Polymer is a sanitizer, since it can correct or modify program behaviour.

Assessment over IRM Implementations

As mentioned in Section 8.2.5, IRMs can be circumventable since they do not impose conditions on the flow of the program, so the target can avoid the IRM rewriter [2]. Nevertheless, integrity is guaranteed. For SASI on x86 code, security code integrity is achieved by employing memory protection mechanisms and by forbidding the target code to reference external entities in an uncontrolled way. For SASI on JVM, Polymer, and PoET, memory protection is by default enacted through Java type-safety; security information (e.g., the security states, the calls executed as a response to security triggers, etc) cannot be compromised because JVML forbids accessing code or data that were not loaded by the classloader.

IRMs are getting large and complex, so there is a significant increase to the TCB of the system. The trusted computing base for IRMs implementations usually includes the rewriter mechanism, as well as a certain compiler and binary analysis module. Schneider suggested leaving the IRM out of the TCB and verifying it with a static type checker, which in its turn is lightweight and does not burden the TCB as much as the whole rewriter.

In terms of policy language, security automata are a straightforward formal method to specify policies but they become difficult to grasp for more complicated policies. In SASI, having an automaton describe the policy makes a hard life to a non-expert. Still, SASI's expressiveness allows for enforcement triggers in the form of any instruction chains (e.g., division by zero, stack access properties). Polymer's Java-like syntax makes it easy to understand and use, since policy concerns are modularised

and such modules are reusable; similarly but maybe not as user-friendly as Polymer, Erlingsson's stack inspection approaches have PSLang policies.

The overheads in the case of PoET/PSLang, are encouraging (as good as JVM's internal stack inspection) because the focus in on specific stack inspection primitives rather than arbitrary machine code instructions. Some initial measurements show relative overheads of between 14-60% compared to the JVM-resident stack inspection (even faster in some cases); there numbers were drastically improved in an optimised implementation so that the approach in [57] can be even faster than the JVM stack inspection. SASI's x86 performs almost as well as MiSFIT, with overheads of between 0.1 to 2.6%. For unoptimised Polymer, instrumented Java core libraries incur around 4 ms per method, and a monitored call about 0.6ms overhead.

8.2.7 Software Dynamic Translation

Software dynamic translation (SDT) is the translation from compiled code to binary code at runtime. SDT translates each line of code in one language into machine language instructions and, if using an interpreter, executes them. As mentioned in Section 8.2.2, we separate interpreters (that are eventually used by SDT) from SDT because SDT focuses on the translation of more than one instruction or command at a time, while interpreters focus on the execution of a (translated or not) instruction at a time. Safe interpreters transform the execution of an instruction, while SDT transform the whole program or parts of it. *Virtual machines* are an important example of dynamic translators; they fetch instructions, do a certain translation to the instructions, then prepare the result to be executed. There is a large number of applications of software translation: binary translation (translating binaries from one instruction set to another), dynamic optimisers, debuggers, dynamic profilers.

SDT Implementations

Some SDT implementations that aim for security are Valgrind [162], Strata [211], DynamoRIO [122], and Java [144]. Valgrind is a powerful tool for dynamic binary instrumentation made to shadow in software every register and memory value with another value that says something about it [162]. Valgrind uses dynamic binary recompilation: (1) (re) compiles the target code, (2) disassembles the code into an intermediate form, (3) instruments, or shadows, with analysis code, and (4) converts the result into machine code again. The resulting translated code is cached and rerun if needed. Java is a SDT system where the Java interpreter stands between the bytecode to be executed and system resources. Whether the executable code is interpreted or Just-

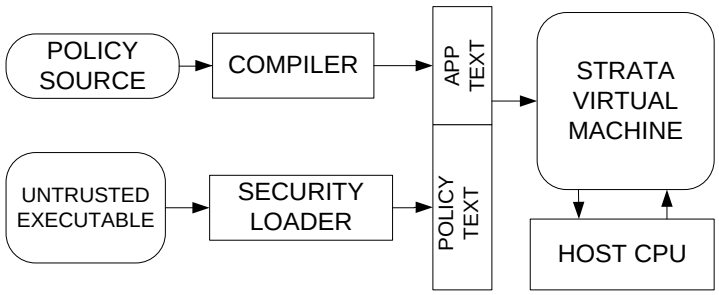


Figure 8.9: How policy code merges with target code in Strata.

In-Time compiled into machine code, the virtual machine is the intermediate layer that ensures portability and extra security. In Java, the interpreter only executes type-checked bytecode and the checks are done both at runtime and compile time[3].

Another approach appears in Strata [211], which supports transparent enforcement of security policies. Strata is a virtual machine that translates executable instructions into safe operations, before executing them; user-defined policies decide whether one instruction is safe and another is not. The policy entry points are usually system calls, and an interpreter mechanism ensures that Strata executes code from within the policy. As shown in Fig. 8.9 [131], the policy code is merged at runtime with application code to correct unsafe behaviour, hence we see Strata as a sanitizer. In Strata, a machine code interpreter can implement a large variety of policies. It does not support a strict policy language: the claim is that any policy language can be used, provided that the developer binds it to Strata constructs. The policies in several papers have a C syntax [211, 210, 101].

Assessment over SDT Implementations

With the Java virtual machine and binary translators, dynamic translation offers a virtualization layer that helps in both security and portability. SDT techniques in security bring strong non-bypassability guarantees e.g., program shepherding guarantees that its sandboxing around any program operation cannot be bypassed, and tamperproofness. In terms of trust model, SDT ensures that as long as the OS, policy writer and interpreter are trusted, the untrusted code cannot bypass the interpreter.

Concerning the security policy language, SDT require minute system knowledge ranging from system call API and kernel structures, to proprietary low-level APIs and security automata descriptions. SDT is approachable for security developers who can write their own call wrappers (which eventually become access control policies).

[3]From this point of view, Java can be listed both as a safe interpreter and an SDT technique.

Criteria	SDT implementations
Abstraction level	runtime and application logic level
Guarantees	nonbypassable, tamperproof
Trusted comp.	OS,interpreter,compiler
Policy class	access control
Policy language	low-level, or custom
Overheads	low to medium e.g., 2-30%

Figure 8.10: Summary on SDT techniques and implementations

SDT implementations show much lower performance than system call interception e.g., from 2 to 10% lower. This is the main reason why software translation is not used in complex applications. For instance, Strata has been shown to bring a performance overhead of 30% which might be unacceptable. As an improvement, the concept of a Just-In-Time compiler was added to the Java implementation, and further research has been done to optimise the software translation process [209]. SDT implementations are assessed in Table 8.10.

8.2.8 Dynamic Aspect Weavers

Aspect-Oriented Programming (AOP) is a software engineering paradigm focused on separation of concerns in software development. Some application requirements (e.g., security) are concerns that transcend the entire application, and adding, changing or removing them requires modifying the whole application. The AOP approach to solve this problem is to weave the initial application with one or more *aspects* or *concerns*. Aspects are high-level objectives and can regard customised system-wide logging, exception handling, performance optimizations, security, etc. Each aspect is associated with the data about where and when to invoke a code block called *advice*.

This procedure implies a number of security-related advantages [238]: a global means of performing security checking (as all the security code is located in one place), separation of the development security concerns, as the developer would not have to worry anymore about security issues along application development, and reusability of security policies across applications. An example is given in [246]: in order to insert authentication features throughout an information management system in a regular OOP way, almost all class interfaces ought to be changed, as well as implementations of existing methods that are considered sensitive.

Aspects can be woven with the application in two ways: *compile-time binding*, or static from our point of view, and *runtime binding*[4]. Static aspect weaving happens

[4]Aspect binding can also be done at load-time, but dynamic aspect binding has received most attention.

when an aspect weaver merges the source file of an aspect with the source code of a class. Once this is done, a regular compiler could compile the resulting code as normal code. An example of a static weaver is AspectJ [121]. Dynamic weaving happens at runtime and enables and disables aspects on the fly. In this case aspects can be created at different moments in time compared to the application, but their weaving together at runtime requires that the application provides some sort of weaving support. Because our work is targeted on runtime security enforcement, we will analyse runtime aspect binding.

Dynamic aspect weaving is a technique at the highest level of abstraction (application level). The advantages gained in this way – expressivity and flexibility in expressing and managing user-level policies – are coupled with weaker guarantees of mediation and non-bypassability, while the TCB is bigger since the weavers have to rely on a number of lower-level mechanisms that would rewrite the code as per the desired aspects.

Dynamic Weavers Implementations

PROSE and Wool [189, 190] are Java-based AOP frameworks for dynamic aspect weaving. The dynamic support for weaving and un-weaving aspects is located within the JVM and is called Java Virtual Machine Aspect Interface (JVMAI). This is a native interface by which the user can manage execution events related to aspect weaving: the user can register requests, get event notifications, and control what happens once a notification is raised. Whenever the currently executing instruction is a joinpoint, the execution is suspended and PROSE calls the crosscut functionality of the aspects associated with that joinpoint. At the cost of interpretation, JVMAI intercepts a wide range of fine-grained events: method calls but also field access, class loading, and breakpoint events. The lifecycle of aspects is independent of the Java application, and aspects can be (un)dynamically plugged.

There are some approaches of weaving of access control concerns, with bytecode rewriting rather than interpretation as above [94, 24, 175]. The SPoX implementation of an aspect-oriented IRM suggests a rewriting algorithm whereby first, a security class is inserted into the untrusted code, then to have all instructions that manipulate certain security-relevant objects rewritten to duplicate its state in the security class; finally, to rewrite all Java method returns to target the rewritten code that contains security checks. SPoX is a recognizer, since the detection of a security violation leads to the program termination. Alternatively, Tom, the approach presented in [175], does Java code rewriting for access control and suggests a method to map the policy environment, the access request identification, and the weaving rules, to the untrusted

program. This process of mapping is an otherwise sensitive task for the Tom compiler. Tom's authorisation decisions are reported to be more flexible than access allow and deny, and hence we see Tom as a sanitizer.

The aspect-based security approaches we have seen are usually formal, base themselves on security automata and mostly focus on the language capabilities of expressing security policies and how these policies can be formalised to deliver more security guarantees. Correctness is an issue: it needs to be proved that the advice, once merged into the code, enforces the specified policy and does not interfere with the property that should be satisfied by the rewritten code [47]. The verification of aspect correctness has been approached with the help of model checking in [127].

Performance enhancements to PROSE are brought by Bockisch et al., who consider another issue of aspect weaving: previous work covered crosscuts statically bound to application code (e.g., a method call). An alternative is dynamic crosscuts - those whose hooks cannot be directly associated with parts of the code (e.g.,"the control flow of variable A contains variable B", or a counter for the invocations of some methods) [31]. The problem is that dynamic aspect weaving, even as it is called dynamic, is usually logically done in a static way: 'dynamic' checks are inserted at all possible joinpoints (e.g., all instructions). This approach leaves no room for really dynamic hooks and for runtime dependencies among these hooks. Bockisch et al. suggest programmatic aspect deployment [150] as a solution to postpone weaving advice. Their architecture leverages on the performance issues of runtime monitoring in PROSE also by making aspect weaving part of the virtual machine execution model.

Assessment over Dynamic Aspect Weavers

AOP techniques present considerable overheads depending on their approach to interception and aspect weaving: reaching joinpoints, context retrieving at jointpoints, retrieving the right advice to call, calling the advice and executing it. A study by Haupt and Mezini reports some research on AOP performance based on the cost of class loading, advice lifecycle management, and various AOP benchmarks [96]. Wool incurs 4 times more time than AspectJ, and is 19% faster than dynamic code translation. The aspect-oriented IRM reported in [94] report a rewriting overhead of between 50 to 80%, but these figures depend on the policy used, the number of dynamic checks and exact weaving method.

Table 8.5 shows a coarse assessment over dynamic aspect weavers. Rewriters that perform aspect weaving perform very well from the point of view of policy language: the aspect language is usually easier to grasp and is dynamic in that aspects can be woven or unwoven dynamically. The downsides are in performance and integrity

Criteria	Dynamic weaving implementations
Abstraction level	application logic level
Guarantees	tamperproof
Trusted components	rewriter, interpreter, compiler, aspects
Policy class	access control
Policy language	automata, very high level
Overheads	high or very high e.g., 50-100%

Table 8.5: Summary on dynamic weaving techniques and implementations

guarantees.

The security issues of applying aspects to an application have been discussed in the state of the art [180, 246, 47]. Palmer emphasises that in an untrusted environment, advice from unknown sources may be malicious [180] and his solution is to use *code contracts*. Sandas and Walker consider the notion of *harmless advice*, that is a block of code (an advice) required to satisfy a non-interference property; the aim is not to alter program invariants, but just use I/O capabilities and start of end certain program computations. De Win et al. bring into discussion the security guarantees of weaving aspects and those of composing security concerns [246]. Addressing this problem, the aspect-oriented in-lined reference monitor [94] brings a policy specification language that merges aspects and security automata; the result is that the rewritten (or interwoven) code is proved to satisfy the desired property.

8.2.9 Enforcement on Data Flow

Policies of the type “no data from this file can be sent via e-mail" cannot be enforced by classic access control mechanisms, because such mechanisms cannot track the way *any* information in a file is processed by some or any application. Information flow control (IFC) tries to bridge this gap; it imposes that unauthorised principals must not gain access to sensitive information – there must be no leakages to domains with lesser clearance. Enforcing IFC policies can be achieved with *dynamic data flow analysis*. This analysis can be performed at several abstraction levels: OS, runtime, library, and application level. Dynamic data flow analysis associates several bits of information with a program object, follows the flow of these bits throughout the program, and if needed modifies these bits according to program statements. No entity other than the trusted monitor is allowed to change value taint value.

There are implementations using taint analysis to ensure that a target application (usually a Web application or a database) will not be subjected to attacks like SQL injections, command injections, cross-site scripting, hidden field tampering, cookie

poisoning, format string attacks, or privilege escalation [38, 92, 163, 251, 43, 39, 139]. Repelling these attacks implies performing data flow control. At runtime this occurs in its three basic steps: (1) *tainting* – data from untrusted sources is marked; (2) *tracking* – all subsequent operations on tainted data are tracked; (3) *asserting* – if marked data is used in illegitimate operations, a violation is asserted. While the approach above is the same for all approaches, what differs is the abstraction level. TaintCheck [163] and Dytan [39] instrument binaries and check jump addresses and memory locations or system call arguments. Approaches like [251, 38, 139, 92] check C or Java methods.

IFC enforcement is, as of now, *mostly static*. It was only recently that dynamic taint analysis has gained ground in handling implicit flows. The state of the art of practical work in this area splits into two branches: JVM approaches and non-JVM approaches. For both types of approaches, the enforcement is realised either by rewriting or translating the application bytecode [253, 235], or by a JVM-wide interception and correction mechanism [37, 160].

JVM-based Dynamic Data Flow Trackers

Implementations based on the JVM aim to add information flow constraints to the Java Virtual Machine [253, 160, 91, 37, 158]. The motivation is to strengthen security controls for a language runtime that serves as a foundation for a large number of applications. Haldar et al. add object mandatory access controls and enforce them dynamically on bytecode [91]. This implies that taints are associated with Java objects, and that the security sensitive events are field reads and writes and method calls. This approach has been deemed rather coarse by subsequent works [37, 253, 160] that deal with a wider range of information flows, multithreading and exception management. The label granularity of these latest approaches is that of pieces of data (derived from files, or network resources or databases) - therefore taints can be associated with object fields, stack contexts and heap contexts [160]. One of the prominent examples in static IFC enforcement is JFlow [158], an enforcer that checks annotations to data flow both statically and dynamically; JFlow does dynamic label binding to variables, and performs dynamic access control checks for each method's access to data and its authority to declassify data values.

Non-JVM dynamic data flow trackers

These approaches target the compiler, runtime libraries or OS [38, 235, 129]. They usually require program source code, hence we do not see them as fully runtime approaches. For instance, some implementations perform static vulnerability analysis

on the code of an application [38, 129]: a special compiler fed with an IFC policy locates the points where the input of the program might generate policy violations. At those specific program locations, the compiler inserts calls to a runtime library and that library manages tag information as the program executes. RIFLE [235] emphasises that, for the user, an unsafe binary is the same as a trusted one, and relies on the policy writer to protect its sensitive information. RIFLE translates the normal application binary into a binary that runs on a special instruction set architecture (a RISC ISA), which is specialised to track both implicit and explicit data flows. A completely dynamic approach is taken by TaintDroid [52], a runtime security solution for Android smartphones that does not require application source code. TaintDroid tracks the way sensitive data is handled across multiple applications, by instrumenting the Dalvik virtual machine interpreter to track data variables contained in messages sent by various applications. All taints are stored in a virtual taint map and checked at runtime when there is a method invocation on sensitive data.

Assessment of Data Flow Tracking Implementations

Most of the dynamic IFC enforcement implementations are able to deter an attacker from manipulating a faulty system by providing improper inputs. However, repelling some attacks (i.e., buffer overflows, code injections) does not necessarily imply highly secured information flows. Le Guernic observes that IFC implementations do not prove soundness and noninterference [90]. Worse, their solutions seldom discuss their degree of mediation. Approaches that modify Java bytecode [92, 253] suffer in that not all method calls are instrumented, which means that IFC applies selectively and thus the whole mechanism could be compromised by targeting native calls.

When it comes to expressing policies, usability of IFC policy languages remains a problem. TaintCheck [163] tracks tainted byte values – those from untrusted sources, or derived from tainted values – and can thus detect dangerous uses of tainted values; it can check if a particular address range or register is tainted. Such a policy is very difficult to specify by a security administrator or user without a deep understanding of the instrumented program. [38] has a similarly low-level policy language. Some JVM approaches [160, 91] follow a Java-like syntax for their policy language or a Polymer [24] syntax. This choice makes it easier for users to write IFC policies.

In terms of performance, the overheads incurred by dynamic IFC implementations are usually important, ranging from around 30% to 200%. TaintCheck uses an x86 emulator to operate on arbitrary binaries at the x86 instruction level, and consequently that induces overheads far over 100%. While a similar performance problem applies to Dytan [39], other approaches employ static analysis on source code [38, 160] or more

Criterion	Data Flow Trackers
Abstraction level	OS, runtime, platform, application
Guarantees	full mediation but not always proven
Policy class	both access and usage control
Trust components	the OS, runtime or platform libraries
Policy language	complex, Java-like syntax in best case
Overheads	very high with minimum overheads around 30%

Table 8.6: Summary on data flow tracking techniques and implementations.

specific optimizations [39, 43]. The obtained performance overheads after employing static analysis range from 7% to 100% [160, 37]. Perhaps one of the best performing is TaintDroid, since it was built for a very constrained execution in terms of resources: TaintDroid reports to be only 27% slower than Android, on an IPC microbenchmark.

The TCB of most IFC frameworks includes the OS or some hardware platforms [43]. In some cases trusting the OS is replaced with trusting a modified JVM middleware, a binary rewriter, an interpreter or some emulation environment such as Valgrind [163]. RIFLE trusts the translation process and the ISA. Table 8.6 gives an overview of the data flow tracking implementations we surveyed.

8.3 Security in Distributed Environments

The work reported in this thesis is most prominently related to building access and usage control enforcement mechanisms fit for SOA. As such, we have separated several main research areas that are connected to this topic: message-level standards, security enforcement in particular distributed systems (Globe, Grid, Web services), usage control architectures for distributed systems, Digital Rights Management (DRM).

8.3.1 Message-Level Security

There is a large number of OASIS and W3C standards for securing Web Services. From the WS-* bundle, security is more relevant with WS-Security, WS-Policy, WS-SecureConversation, and WS-Trust.

WS-Security aims to protect SOAP messages so that they are neither accessed nor tampered with while they are in transit [166]. This standard suggests basic ways in which to protect one or multiple message exchange. The WS-Security specification defines a message security model as a combination of tokens (usernames, binary, XML tokens) and digital signatures to authenticate SOAP mes-

sages. This information is incorporated in the SOAP header by means of a $< wsse : Security >$ header block, that is associated with a message recipient.

WS-Policy is a framework for specifying Web service policies [244]. Policies are composed of assertions and expressions. Assertions have application or domain-specific semantics, they can be issued by different authors, and define what is required of or what a policy subject can do. A policy subject is any entity that can be associated with a policy and can be an endpoint, a message, etc. A policy expression is the XML expression of the policy, and the assertions are connected with policy operators. Policies can contain sensitive data, hence they need be protected against tampering by using other standard procedures like XML Digital Signatures, SSL/TSL, and WS-Security if they are part of a SOAP message. The ways in which a WS-Policy can be associated with a subject is described by WS-PolicyAttachment [241]; there are two ways to do this association: either with WSDL, or with UDDI descriptors.

WS-SecureConversation extends the mechanisms in WS-Security with a framework and syntax that maintain a shared context and support for session key derivation for multiple messages in a Web service conversation [171]. Each security context is associated with a token based on a shared secret, and is designed to be extensible and to have a certain duration. Within each context, different keys can be added/derived so as to maintain key freshness or to associate a different key to a different operation. WS-SecureConversation suggests a key derivation algorithm based on RFC 2246.

WS-Trust extends the mechanisms in WS-Security with a way to handle security tokens in a trusted manner [172]. This specification addresses methods to issue, update and validate such tokens, especially when there are brokered trust relationships among the participants to the exchange. The idea at the foundation of this framework is that a Web service that receives a request from another Web service, should be able to reject serving it if the claims in the request are not trusted. This trust comes from successfully authorizing security tokens in the message request. The WS-Trust specification proposes a security token service framework, that includes ways to request, process and return one or more security tokens, how to renew them and how to validate them.

Other security-related specifications are OASIS's SAML, W3C's XML Signature and XML Encryption.

SAML The Security Assertion Markup Language (SAML) suggests an XML syntax

and semantics for security assertions [167]. These assertions are issued by a special authority and can be of one of three typed: to authenticate the subject, to associate the subject with an attribute, or to reply positively or negatively to an authorisation request for the subject to a specified resource. The assertions can be time-bound, can restrict their further usages in the future, or can be intended for a certain “audience". An assertion that cannot be evaluated, is by default rejected. SAML also describes a syntax for authorisation decision statements; the decision can be *permit*, *deny*, or *indeterminate*. Also, the specification contains several protocols in which SAML information can be bound and exchanged across services.

XML Signature gives an XML syntax and rules to handle digital signatures [242]. XML signatures can be detached from the data they sign, or can be attached to it. In itself, to sign an XML document means to associate a key with the data. The specification describes how to generate and validate the signature.

XML Encryption specifies a method to encrypt data (either XML or not) and represent the result as an XML document [240]. When encrypting XML data, the identity of the elements that were encrypted is hidden; encryption can be of encrypted data (so-called “super-encryption"). Recently, Jager and Somorovsky have shown that XML Encryption can be broken by exploiting a vulnerability that comes with the block cipher mode, the encoding of the text, and the response of a target service [110].

The work reported in this thesis complements the message security features supported by these standards. As mentioned in Section 4.10, this thesis examines security constraints that govern an application, and that hence have a meaning in that application. The constraints that we consider are orthogonal to authorisation and authentication of messages per se, and focus on higher-level conditions among message flows and entities.

8.3.2 Security Enforcement in Globe

Globe [219] is a wide-area distributed system suggested as an alternative to the Internet when it comes to managing distributed objects (e.g., Web documents, Web pages). In an attempt to build a scalable middleware layer with an unique view over its resources, Globe suggests distributed shared objects – independent objects that are physically distributed but have their own policies for migration, replication and security. A comprehensive list of requirements for securing Globe is presented in [135];

these requirements cover aspects such as communication among objects, platform and network security, security associations among objects, trusted caching. Very interesting in this work is the comparison between a centralised and a distributed security coordination among the components of a Globe object: a centralised design is shown to offer more advantages, especially such as complete mediation, least privilege to sensitive data and elimination of duplicate functionality in security protocols. As a matter of fact, the same ideas are with the xESB approach presented in this thesis: as a central and complete mediator, xESB ensures least privilege to sensitive data since its actions have a direct impact over the the higher middleware levels (i.e., BPEL).

A more practical discussion and a concrete security architecture for Globe is given later in [188]. The authors suggest a two-way certificate-based authentication between the client and the Globe object. The authentication process should ensure that 1) the client is protected from possibly malicious code (when installing a local object on its machine) and 2) that the valid request needs only be executed by objects (replicas) that are allowed to execute it. The more interesting step 2 imposes that a client security proxy looks for a Globe object with the rights to execute the invocations, and relies on hints from an untrusted directory service to look that object up. Of course, a simple denial of service attack can be devised as making the repository unresponsive, or as making it reply to such a request with "*no object with these permissions*". Another limitation of this design is that, even if it deals with replication strategies of application-level objects, it does not cope with changing policies. A policy change would trigger changes in the certificate chain among the distributed objects; also, the client permissions are fixed and hardcoded into the security subcomponents of each Globe object. Worse and a bit surprisingly, the Globe architecture does not discuss aspects such as time-of-check-time-of-use for configuration data in a massively distributed scenario. By comparison, the work presented in this thesis does address the latter problems; even if xESB relies on a trusted service directory, an authentication scheme between the service bus and the deployed services would ensure a tighter control over the application functionality.

8.3.3 Enforcing Usage Control and DRM in Distributed Systems

There is a recent body of work in the area of enforcing usage control in distributed systems. However, very few if any of the works we have seen come with any concrete implementations of the models and architectures presented. Pretschner et al. [194] and Katt et al. [118] separate between usage control specification (i.e., policy language) and enforcer mechanisms, as well as the difference between *provisions* – as actions to have been done by *'now'* – and *obligations* – as actions that need be per-

formed in the future. Obligations are constraints on data that can relate to cardinality, time conditions, boolean predicates, technical restrictions, and necessity of data updates. Some obligations are controllable or observable (somebody else performs them, or not). The architecture for SOA usage control enforcement suggested here resembles the one presented in later work [195]: a logical architecture similar to xESB's. Also, in [195] the authors present the difference between enforcement that prevents violations, and enforcement that reacts to violations; this is something that xESB can distinguish and perform at the message level when performing blocking of messages (prevention of violation), as opposed to when it modifies message metadata (reaction to a violation).

A slightly different architectural sketch for usage control enforcement for mobile device systems is presented in [206]. A reference monitor is designed on the client-side, yet by design it has several vulnerabilities: it relies on timers that are not synchronised with a server, the policies are fixed and updates of security-relevant objects are not considered, not to mention management and configuration aspects. Similar client-side enforcement is presented also by Agreiter et al. [28], where the focus is to protect data in a distributed system, and hence policies are attached to the data and sent to the consumer. The authors use trusted computing in order to achieve a trustworthy environment on remote clients, starting from a low level TPM hardware, and going across the operating system, the JVM and the application levels (OSGI, in this case). The limitations of this approach is that only trusted applications can be run, and that these would not produce any policy violations.

DRM enforcement is related to usage control enforcement architectures. Some authors have observed that DRM is always trying to intercept and block disallowed accesses (before they happen) and the data provider's control is always the strongest and most important to enact [195]. A conceptual example of DRM architecture is Erikson's [53], where the author observes that policy enforcement assurance depends on the trust in the platform that enforces the policies on the client side; Michelis et al. give a stratified, distributed view over several state of the art DRM technologies [152].

8.3.4 Security Enforcement in GRID Computing

There have been efforts for authorisation enforcement in GRID systems as well. For example, complex and yet preventive usage control constraints that are typical to GRID sessions can be expressed with POLPA [19]. An architecture for enforcing usage control constraints expressed in POLPA is given in [125], where the PDP is an orchestrator of the checks among various trusted managers (attribute manager, condition managers, obligation manager); equally, the PDP mediates between the

verification modules and the PEPs at various abstraction levels (JVM, Grid services, etc). To an extent, this idea of enforcement actions at different abstraction levels is also present with our BPEL/xESB approach, yet the authors in [125] do not offer any practical implementation, nor any discussions on the management aspects and security assurance of their approach.

Chadwick et al. present PERMIS as an authorisation infrastructure for GRID systems [35]. PERMIS is built to deal with cross-organization security decisions and relies on RBAC/ABAC security policies that need data spread across various systems. The presented scheme offers distributed management of credentials and history-based decision making. PERMIS can handle both push and pull modes for attribute retrieval, but does not discuss policy changes and attribute management for attributes that are not credentials. Nevertheless, PERMIS makes a good case study for configuration management, as we described it in Chapter 7.

Another relevant approach is that of PEI (Policy models, Enforcement models, Implementation models) [255], where the authors approach some management aspects of attribute retrieval (push and pull modes for persistent attributes, and update propagation); their architecture introduces a centralised repository of attributes, that pushes changes all throughout the collaborative system. We do not support such a view, because of several reasons: first, a centralised attribute repository does not completely solve the propagation of attribute updates and its time-of-check-time-of-use problems; second, this would generate large amounts of traffic to push changes to attribute consumers, even if the consumers are not 'interested' in those updates; third, it is not motivated why attribute changes are pushed and not pulled; fourth, it would not scale for a system with a large amount of attributes, like eBay.

Finally, Butt et al. [33] briefly survey security issues in GRID, and the limitations in dealing with them in some systems. The authors emphasise the need for runtime monitoring of GRID processes and that of sandboxing them in the host systems. This need becomes especially stringent as the GRID is subject to attacks such as those reported by Miller et al. in [153]. Miller at al. attack the Condor GRID scheduler when legitimate jobs execute on malicious machines, and where a malicious superuser can take control over the Condor job by modifying the runtime image of the initial job. As in other cases, a way to protect against this kind of attacks is to sandbox remote malicious program activity on 'clean' environments, yet the case where legitimate tasks is run on malicious hosts remain an open problem.

8.3.5 Other Approaches

SeAAS is a recent approach against endpoint-only SOA enforcement [151]. SeAAS suggests security (i.e., authentication, authorisation and non-repudiation checks) as a service that is attached to the ESB and invoked for every call chain that is relevant to a policy in the policy repository. This approach is also suggested in our previous work [176], yet it has the disadvantage that it hardcodes the security flows into the ESB message routing mechanisms, so that the 'weaving' of security checks becomes static: no security infrastructure changes can occur at runtime. This approach is, like xESB, centralised, but it is conceptual and does not investigate what are the concrete enforcement capabilities at message-level.

An interesting approach of distributed policy enforcement is that of 'sticky policies' [156], whereby the data involved in a multi-party enterprise transaction is irreversibly associated with the security policies on this data. The privacy model thus envisaged requires tracing and auditing authorities that would monitor the data traveling in the application. These services and the policy enforcement mechanisms are built on top of a trusted platform. Nevertheless, it is not shown how the trust/tracing authorities cope with the distribution of keys and revocation certificates in an inherently large system. The approach in this thesis is similar in that it uses a common mediator for data tracing and audit, yet we suggest that the policies are separate from the data (otherwise the policy update problem resembles the attribute update problem presented in previous chapters).

Similarly to Zhang et.al's PEI model [255], Tsai et al. [226] suggest a policy infrastructure for SOA that allows for collaborations among security components like PEPs and PDPs, at runtime. The work is in how to combine and coordinate policy enforcement from global to local enforcement entities, when the policies are on business processes.

8.4 Summary

This chapter presented the state of the art in policy enforcement at application runtime. The vast majority of the existing enforcement approaches is focused on single-host scenarios rather than on distributed system policies. A reason for this emphasis is the prevalence of mobile code, and also that, until recently, applications on the market would not be as distributed and large as now. For this reason we describe and compare the single-host approaches in more detail, and also overview some of the most important distributed security monitor approaches.

In single-host systems, enforcement by halting the program when unwanted behaviour is about to happen is too restrictive. Knowing what calls to intercept and process implies knowledge of the target application and also of a policy language to express policy requirements. Since different behaviours of the target might map to the same chains of calls, there are two alternatives: (1) to run the target in a secure environment where each call is monitored – but this approach is expensive – or (2) to rewrite the code of the target so that it will not misbehave. Program rewriting is supported by off-the-shelf tools (e.g., bytecode rewriters and aspect weavers), and delivers strong enforcement guarantees independent of the platform. The problem is, however, that the predominant enforcement model of rewriting is that of security automata. Security automata are not yet fully mature to fit real-world constraints. Albeit supported by theoretical work, implementing security automata cannot yet properly *correct* the program execution when a policy is violated and cannot monitor more than one program implementation at a time. The automaton enforcement models can look at previous program executions (with some memory constraints), but when making decisions it is completely isolated – it cannot receive input from any other entity. Some other limitations of security automata are: they assume the policy is already available as an automaton, which is usually difficult to obtain; their event histories are usually assumed to be infinite [55]; they correct only side-effect-free transitions and cannot deal with information flows, concurrency and user sessions.

With distributed applications, some authors have already noticed that interception of security-relevant application activity becomes more difficult with the need for continuous application monitoring (required by enforcement of usage control); similarly, performing enforcement actions becomes more difficult when minimal application disruption is essential. By comparison, the work reported in this thesis presents the first design and concrete implementation of a centralised security monitor that is fit for runtime cross-service policy enforcement in a distributed system, but also that can be easily extended, managed and assessed.

Chapter 9

Summary and Conclusions

This final chapter presents a summary of the work reported in this thesis, along with our conclusions and some directions for future work.

9.1 Summary

In Chapter 1 we have described the setting of today's distributed applications – a very dynamic area where what sells is the service. Services provide functionality and are used and re-used across organisations. Organizations need to be compliant with security regulations, and hence need to ensure that the services that they lend or rent obey the constraints of each scenario. As a consequence, there are three directions in which organisations need to divide their efforts: building controls security controls that are 'good enough', assessing whether the controls really ensure that security policies are respected, and 'fine-tuning' the controls to influence security assurance. The practical and conceptual problems with the current approaches mandate the approach of this thesis – of a middleware-level tool that allows for custom-building policy enforcement mechanisms, assessing their assurance and changing them for better results.

Chapter 2 describes the background concepts and technologies that are referred to in this thesis. We start from Web services, continue with a brief overview of SOA and SOA layers, out of which we emphasise the tools of the trace in this thesis – the Enterprise Service Bus and BPEL as a technology for business process management. On this background, we describe security regulations that impact the software infrastructure of each organisation, and its contractual relations with others. Although not in the focus of this thesis from the point of view of how to derive, we describe security policies and different access control models. We continue with mechanisms for policy enforcement, and policy languages to express enforceable security constraints. Lastly, we overview some policy-management aspects along with some security metrics.

The case study used in the rest of this dissertation is described in Chapter 3. We describe a case study of a healthcare application that involves the cooperation of several healthcare institutions. Because of shared electronic health records, patients' sensitive data needs at all time be protected against a number of threats, that include data tampering, unauthorised disclosures, or audit. On this background, we describe the actors and their goals in our scenario, and the security constraints that stem from various healthcare regulations, and that must be all obeyed. Healthcare makes a fitting scenario for security policy enforcement because it is a very well-regulated area, yet a financial or an e-commerce scenario would trigger similar discussions.

Chapters 4, 5, 6 and 7 present the main contributions of the work reported in this dissertation, and follow the flow in Figure 9.1. Chapter 4 proposes xESB, the first security mechanism that can enforce message-level security policies that apply to several service endpoints at the same time. xESB is based on a centralised message mediator and can perform a wider range of enforcement actions than any previous security monitor; moreover, it can act at application runtime, without a big performance impact. Since enforcement capabilities are also related with the policy language exploded to specify security constraints, we presented two languages used with xESB. Moreover, Chapter 4 also shows that enforcement actions are flexible and customizable, which makes xESB a very versatile and useful tool for security administrators and developers.

xESB's enforcement strength comes from its ability to process message metadata, but is limited in that it cannot understand message payload in message flows across services in the service-oriented application. This limitation is addressed in Chapter 5, that consolidates the enforcement capabilities of the original xESB by combining it with the strength of the business process security monitor. Chapter 5 analyses and compares the enforcement capabilities at message level and those at business process level. Based on some previous work with BPEL monitors, we describe a central security component that can monitor events and can perform enforcement actions at both abstraction levels. This work makes xESB the first middleware security monitor that acts across service endpoints but also at both message and application layers.

Building a mechanism that enacts security policies is necessary but not sufficient in a setting where provable assurance is critical. Chapter 6 proposes, from this perspective, the first concrete security enforcement indicators. Even though the indicators idea is not new, the work reported in this chapter is the first to implement indicators for security mechanisms rather than for business processes. Based on violations defined in the policy, we have implemented a tool that analyses logs produced by

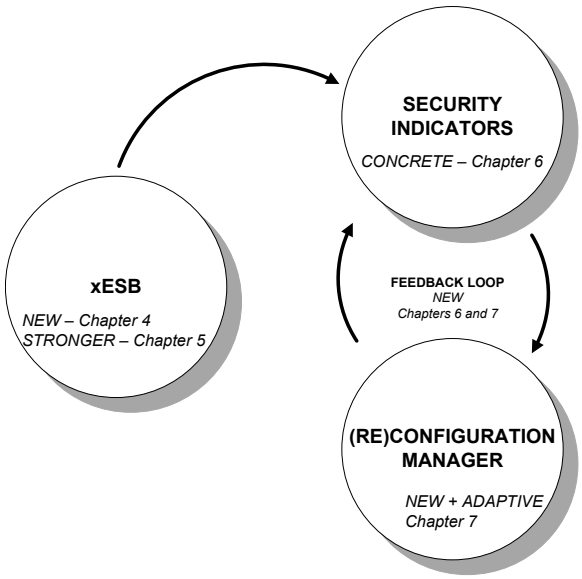


Figure 9.1: The contributions in this thesis and their features.

security enforcers, and decides whether the cumulated deviation from the wanted behaviour surpasses predefined thresholds. In case it does, this deviation will trigger a compensation action of a certain type, that is correlated with the type of deviation.

Even though compensation actions are mostly application-dependent, their types are most probably not: if found not to perform as expected, a security mechanism can be replaced or reconfigured. Reconfiguration means changing runtime parameters. Chapter 7 completes the feedback loop idea by focusing on reconfiguration of security mechanisms from the point of view of how they are connected. We are the first to make the observation that the communication links among the security components in a distributed system should be given special attention. With an RBAC/ABAC model in mind, we focus on security attribute handling. We observe that attributes that are not properly retrieved and updated can impede security assurance (from the point of view of the correctness of the security decision). We design and implement a central entity called Reconfiguration Manager, that manages the constraints of attribute handling and is able to configure the connections between security components at runtime, with acceptable performance overheads for a medium-sized network. The middleware tool thus proposed is adaptive and its (re)evaluation can be triggered as a compensation action when a security indicator surpasses allowed limits.

A comprehensive revision of the state of the art in security policy enforcement is given in Chapter 8. Chapter 8 is split in two parts: the first is a part dedicated to mechanisms for policy enforcement – after the compilation phase – on a single-machine system. We analyse a number of enforcement techniques and implementations, and

offer a taxonomy of approaches with varying advantages. The other part of Chapter 8 is on existing approaches in distributed system security policy enforcement. By analyzing these solutions we derive that insofar there are few practical solutions to enforce policies in a distributed system because of the several abstraction layers, and be because of the problems that are inherent to distribution (synchronization, consistency, etc). Lastly, the present chapter presents a summary of the work reported in this dissertation, conclusions and several future work considerations.

9.2 Conclusions

The work reported in this dissertation achieves its goal to design, develop and implement a comprehensive set of middleware-level tools for security policy enforcement in a SOA system. With security regulations that apply across applications, we made the point that it is better to enforce security policies from a central point across service endpoints, than to embed different security controls for the same policy on each endpoint. To this end, we have proposed xESB, an instrumented ESB that has been incorporated in the prototype of a big European project.

Also, we have drawn the line between low-level (or message-level), and higher-level (business logic) authorisation policies. We show that treating security-relevant events at either business level or message-level is not enough to have a flexible and powerful security mechanism that reacts *meaningfully* to violations. It is difficult to address security misbehaviours observed at message-level; conversely, at business-level, reactions to security events might be meaningful to the application management, but the mechanisms at this level rely on lower-level mechanisms that introduce their own vulnerabilities. The approach in this thesis is to have a centralised mechanism that, with some hints from the application administrator, can coordinate monitoring and reacting to security events both at the level of the infrastructure (message-level) and at the level of the application (business-level).

Apart from proposing a method to combine the capabilities of these levels in one model, we have showed that is possible to assess existing policy enforcement implementations and to react to enforcement deviations at application runtime. We propose not to separate but rather treat together security assessment with a reaction – or a correction – to the overall system state. We advocate that, instead of relying on a yearly audit to discover issues in the security implementation, it is much more effective to discover and address security issues in-house and at runtime. A universal security assessing method that is meaningful for every application cannot exist. However, this dissertation sets the foundation of a customizable framework able to compare actual

outputs of security infrastructures against desired outputs, help direct attention towards possible sources of misbehaviour, and couple misbehaviour types with types of potential corrections.

Also, we show that security and performance are difficult to satisfy together fully, but *should* be addressed together. In the current practice, performance optimizations come first and security is an afterthought. We made the point that this approach impacts the correctness of security decisions, and that there are ways in which to consider the ensemble of constraints that stem from both security and performance needs. We showed that these constraints can be conflicting. The strategy to choose which of them should be satisfied first should not be fixed, because, as it has been previously mentioned, the balance between security and performance needs evolves along with the distributed application. The security components should be reconfigured whenever the strategy of the management changes, but also due to other changes in the system. These reconfiguration reactions can be triggered by an automated management layer, that is above that of the security policy enforcement, and whose presence we have demonstrated by justifying the (re)configuration manager.

To our knowledge, all these aspects have been previously addressed in isolation, because building security mechanisms is done in-house, and consequently so are testing and 'tuning' them to changing conditions. Addressing these aspects together, rather than separate, is the first step towards provably (i.e., by means of evidence) effective, flexible and dynamic security controls.

9.3 Observations and Future Work

We found several directions for future work and open questions along with our work.

To start with, policy evaluation at runtime is difficult. For any application context, plugging in an off-the-shelf policy engine should not be done without some investigation on how to evaluate events against policies. xESB with its original language tackled only the surface of deeper problems such as preselection (or filtering) of policies that are applicable to message flows, or what happens when multiple policies apply at the same time to a message at infrastructure level. Also, in a large policy file a lot of time is spent evaluating conditions in order to find those rules or obligations that apply to a given message. To improve this in xESB, we could implement some form of the Rete algorithm [69], or select rules according to which message parts appear in `where`-clauses. In all, it would be interesting to investigate how existing policy evaluation engines deal with these problems, what heuristics they use, and how those can be customised to the constraints of a given application.

Another set of open issues relate to timing and synchronization. It does indeed make sense to have a `delay` semantics as a way to postpone the enforcement action in case a security decision cannot be made for the moment. However, this action can have side-effects that impact the application. For example, the service waiting for a reply when that reply has been delayed for security reasons, might timeout and affect the operation of a business process. Worse, the artificially-introduced delay cannot be guaranteed for a fixed period of time, simply because in a distributed system event occurrence at service endpoints is not easy to guarantee. Similarly, in the case of a timed obligation (e.g., the user or system should perform a particular action within 3 days) it is an open problem to ensure that the message notification that an action has been performed, arrives sooner or later than the violation notification that marks the absence of that action, for a fixed period of time.

Deploying a policy enforcement architecture onto an existing system is not at all trivial. Off-the-shelf policy decision engines can hardly cope with the evolution of policy languages (e.g., with the recently released XACML3.0) and usually do not cover problematic aspects such as:

- how they should be deployed in relation with other security components;
- how they are able to deal with multiple policies;
- how they deal with policy updates;
- how they deal with policy conflicts.

As mentioned in Chapter 7, in the XACML architecture suggested by OASIS, the PDP does not know what attributes it can retrieve directly (pull) and what attributes it can retrieve indirectly (push) and in what conditions. Moreover, policy updates are a problem when they occur during the enforcement of their older versions; it is not yet clear how to deal with a policy update with minimal disruption to the security subsystem: should the updates be deployed while stopping current evaluations (with a risk of having incomplete security processes), or should they be deployed after all current evaluations have finished (with the risk of having wrong security decisions)?

From the point of view of the cross-level enforcing mechanism presented in Chapter 5, we have envisioned that the decision making component decides whether to delegate the enforcement actions to the ESB or the BPEL levels based on flags contained in the policies. A future step is to investigate the extent to which this process can be automated. It might be possible to create an automated tool that based on keywords in a given policy, can determine whether the policy requires monitoring or performing actions at the message-level or business process level.

From the point of view of security indicators, it is also useful to investigate the impact of policy combination and policy conflicts over the values and expressions of enforcement indicators. An aspect worth investigating in indicator specification is that they make some implicit assumptions, e.g., enforcement effects are independent, and that they can always be mapped to their causes (which might not always be easy). However, in a realistic scenario policies might overlap in their requirements, hence deviation specifications of different policies might overlap or be contradicting. For policy overlaps or even policy conflicts, there can be indicators that measure the gravity of that overlap or conflict. It would be useful to have a methodology to evaluate the correlation between the indicators and the available enforcement corrections, as well as to investigate the efficiency of the corrections performed at runtime.

Evaluating performance overheads in these SOA scenarios is an important body of work in its own right. In the absence of recognized SOA benchmarks[1], even establishing a measure of the baseline performance (i.e. the performance of the system without any security mechanism) is not yet possible. Meaningfully measuring the performance of the application with the added security subsystem depends on the baseline performance, and remains an important and difficult topic because of the large number of testing parameters and their dependencies, that are application or deployment specific. For instance, the complexity of a policy, the number of security components, the attributes and their distribution over organisational domains, etc.

Last but not least, more work should be done to investigate how XACML 3.0 can syntactically accommodate the information that our middleware framework requires. As of now, we assumed that information like attributes about attributes (as in Chapter 7) and the indicator-related data as in Chapter 6, can be supplied along with the XACML policy. It would be possible with no great difficulties to develop an engine that can obtain this extra information in another way, e.g., from a configuration file written by a security-aware application administrator.

[1] However, we are aware of a SPEC systems group, the SPEC SOA committee at `http://www.spec.org/soa/`, that has recently started to develop a new standard benchmark for SOA middleware.

Bibliography

[1] Aalst, W.M.P.v.d., Dumas, M., Ouyang, C., Rozinat, A., Verbeek, E.: Conformance checking of service behavior. ACM Trans. Internet Technol. 8, 13:1–13:30 (May 2008)

[2] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM conference on Computer and communications security. pp. 340–353. CCS '05, ACM, New York, NY, USA (2005)

[3] Globus Alliance: Globus Toolkit 4 API. http://www.globus.org/toolkit/docs/4.2/4.2.1/security/wsaajava/pdp/wsaajava-pdp-XACMLAuthzCallout.html (Nov 2010)

[4] Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. 21, 181–185 (October 1985)

[5] Amatayakul, M.: Managing information privacy and security in healthcare. http://www.himss.org/content/files/CPRIToolkit/version6/v7/D09_Setting_Standards_in_Healthcare_Information.pdf (2007)

[6] Anderson, J.P.: Computer security technology planning study. ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command 2(1) (1972)

[7] Anderson, R.: A security policy model for clinical information systems. In: Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on. pp. 30 –43 (may 1996)

[8] Anderson, R.J.: Security engineering - a guide to building dependable distributed systems (2. ed.). Wiley (2008)

[9] Anupam, V., Mayer, A.: Security of web browser scripting languages: vulnerabilities, attacks, and remedies. In: Proceedings of the 7th conference on USENIX Security Symposium - Volume 7. pp. 15–29. USENIX Association, Berkeley, CA, USA (1998)

[10] Apache: Apache tomcat server. http://tomcat.apache.org (2011)

[11] Apache: Servicemix esb. http://servicemix.apache.org/home.html (2011)

[12] Armellin, G., Betti, D., Casati, F., Chiasera, A., Martinez, G., Stevovic, J.: Privacy preserving event driven integration for interoperating social and health systems. In: Proceedings of the 7th VLDB conference on Secure data management. pp. 54–69. SDM'10, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1889159.1889166

[13] Armellin, G., Betti, D., Casati, F., Chiasera, A., Martinez, G., Stevovic, J.: Privacy preserving event driven integration for interoperating social and health systems. In: Jonker, W., Petkovic, M. (eds.) Secure Data Management. Lecture Notes in Computer Science, vol. 6358, pp. 54–69. Springer Berlin / Heidelberg (2010)

[14] Arsanjani, A.: Service-oriented modeling and architecture. http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/ (2004)

[15] Atluri, V., Gal, A.: An authorization model for temporal and derived data: securing information portals. ACM Trans. Inf. Syst. Secur. 5, 62–94 (February 2002)

[16] data protection authority, I.: Italian personal data protection code, legislative decree no. 196. `http://www.privacy.it/privacycode-en.html` (June 2003)

[17] Axiomatics: Axiomatics Policy Server 4.0. `http://www.axiomatics.com/products/axiomatics-policy-server.html` (Nov 2010)

[18] Backes, M., Pfitzmann, B., Schunter, M.: A toolkit for managing enterprise privacy policies. In: In Proc. of ESORICSŠ03, LNCS 2808. pp. 162–180. Springer (2003)

[19] Baiardi, F., Martinelli, F., Mori, P., Vaccarelli, A.: Improving grid services security with fine grained policies. In: Proc. On the Move to Meaningful Internet Systems Workshop. LNCS, vol. 3292, pp. 123–134. Springer-Verlag (2004)

[20] Baresi, L., Guinea, S.: Towards dynamic monitoring of WS-BPEL processes. In: ICSOC 2005, Third International Conference of Service-Oriented Computing, volume 3826 of Lecture Notes in Computer Science. pp. 269–282. Springer (2005)

[21] Basel Committee on Bank Supervision: Basel 2. `http://www.centralbank.ae/pdf/news/BaselIIGuidelines.pdf` (mar 2007)

[22] Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: Foundations of Computer Security. DIKU Technical Report, Copenhagen, Denmark (Jul 2002), `http://www.ece.cmu.edu/~lbauer/papers/editauto-fcs02.pdf`

[23] Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. SIGPLAN Not. 40, 305–314 (June 2005)

[24] Bauer, L., Ligatti, J., Walker, D.: Composing security policies with Polymer. In: Proc. ACM SIGPLAN conf. on Programming language design and implementation. pp. 305–314. ACM, New York, NY, USA (2005)

[25] Bellovin, S.M.: On the brittleness of software and the infeasibility of security metrics. IEEE Security and Privacy 4 (July 2006)

[26] Berman, A., Bourassa, V., Selberg, E.: Tron: process-specific file protection for the unix operating system. In: Proceedings of the USENIX 1995 Technical Conference Proceedings. pp. 14–14. TCON'95, USENIX Association, Berkeley, CA, USA (1995)

[27] Bernstein, P.A.: Middleware: a model for distributed system services. Commun. ACM 39, 86–98 (February 1996)

[28] Berthold, A., Alam, M., Breu, R., Hafner, M., Pretschner, A., Seifert, J.P., Zhang, X.: A technical architecture for enforcing usage control requirements in service-oriented architectures. In: Proceedings of the 2007 ACM workshop on Secure web services. pp. 18–25. SWS '07, ACM, New York, NY, USA (2007), `http://doi.acm.org/10.1145/1314418.1314422`

[29] Bielova, N., Massacci, F.: Do you really mean what you actually enforced? edit automata revisited. In: The 5th International Workshop on Formal Aspects in Security and Trust (FAST'08) (2008)

[30] Bishop, M.: What is computer security? Security Privacy, IEEE 1(1), 67 – 69 (jan-feb 2003)

[31] Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual machine support for dynamic join points. In: Proc. 3rd intl conf. on Aspect-oriented software development. pp. 83–92. ACM, New York, NY, USA (2004)

[32] Brunel, J., Cuppens, F., Cuppens, N., Sans, T., Bodeveix, J.P.: Security policy compliance with violation management. In: FMSE '07. pp. 31–40. ACM, New York, NY, USA (2007)

[33] Butt, A.R., Adabala, S., Kapadia, N.H., Figueiredo, R.J., Fortes, J.A.: Grid-computing portals and security issues. Journal of Parallel and Distributed Computing 63(10), 1006 – 1014 (2003), special Issue on Scalable Web Services and Architecture

[34] Cappellaro, G., Longo, F.: Institutional public private partnerships for core health services: evidence from italy. BMC Health Services Research 11(1), 82 (2011), http://www.biomedcentral.com/1472-6963/11/82

[35] Chadwick, D., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: PERMIS: a modular authorization infrastructure. Concurr. Comput. : Pract. Exper. 20, 1341–1357 (2008)

[36] Chadwick, D.W., Su, L., Laborde, R.: Coordinating access control in grid services. Concurrency and Computation: Practice and Experience 20(9), 1071–1094 (2008)

[37] Chandra, D., Franz, M.: Fine-grained information flow analysis and enforcement in a java virtual machine. Computer Security Applications Conference, Annual 0, 463–475 (2007)

[38] Chang, W., Streiff, B., Lin, C.: Efficient and extensible security enforcement using dynamic data flow analysis. In: Proc. CCS '08. pp. 39–50. ACM, New York, NY, USA (2008)

[39] Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA '07: Proc. of the 2007 Intl. Symp. on Software testing and analysis. pp. 196–206. ACM, New York, NY, USA (2007)

[40] Colombo, M., Lazouski, A., Martinelli, F., Mori, P.: A Proposal on Enhancing XACML with Continuous Usage Control Features. In: Grids, P2P and Services Computing. pp. 133–146. Springer US (2010)

[41] Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., Gligor, V.: Subdomain: Parsimonious server security. In: Proceedings of the 14th USENIX conference on System administration. pp. 355–368. USENIX Association, Berkeley, CA, USA (2000)

[42] Crispo, B., Gheorghe, G., Giacomo, V.D., Presenza, D.: Master as a security management tool for policy compliance. In: ServiceWave. pp. 213–214 (2010)

[43] Dalton, M., Kannan, H., Kozyrakis, C.: Raksha: a flexible information flow architecture for software security. In: Proc. 34th annual Intl. Symp. on Computer architecture. pp. 482–493. ACM, New York, NY, USA (2007)

[44] Damianou, N., Dulay, N., Lupu, E., Sloman, M., Tonouchi, T.: Tools for domain-based policy management of distributed systems. In: Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP. pp. 203 – 217 (2002)

[45] Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks. pp. 18–38. Springer-Verlag (2001)

[46] Dan, A., Mohindra, A., Ramaswami, R., Sitaram, D.: Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code (1997)

[47] Dantas, D.S., Walker, D.: Harmless advice. In: Proc. ACM SIGPLAN-SIGACT symp. Principles of programming languages. pp. 383–396. ACM, New York, NY, USA (2006)

[48] Department of Defense: Department of Defense Trusted Computer System Evaluation Criteria (Dec 1985), dOD 5200.28-STD (supersedes CSC-STD-001-83)

[49] Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 109–124. SP '10, IEEE Computer Society, Washington, DC, USA (2010)

[50] DeYoung, H., Garg, D., Jia, L., Kaynar, D., Datta, A.: Experiences in the logical specification of the hipaa and glba privacy laws. In: Proceedings of the 9th annual ACM workshop on Privacy in the electronic society. pp. 73–82. WPES '10, ACM, New York, NY, USA (2010)

[51] Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A policy deployment model for the ponder language. In: Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on. pp. 529 –543 (2001)

[52] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation. pp. 1–6. OSDI'10, USENIX Association, Berkeley, CA, USA (2010)

[53] Erickson, J.S.: Fair use, drm, and trusted computing. Commun. ACM 46, 34–39 (April 2003), http://doi.acm.org/10.1145/641205.641228

[54] Úlfar Erlingsson, Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Proc. of the 1999 workshop on New security paradigms. pp. 87–95. ACM, New York, NY, USA (2000)

[55] Erlingsson, U.: The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University, Ithaca, NY, USA (2004)

[56] Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., Necula, G.C.: XFI: software guards for system address spaces. In: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 75–88. OSDI '06, USENIX Association, Berkeley, CA, USA (2006), http://portal.acm.org/citation.cfm?id=1298455.1298463

[57] Erlingsson, U., Schneider, F.B.: IRM enforcement of Java stack inspection. In: Proc. IEEE Symp. on Security and Privacy. IEEE Computer Society, Washington, DC, USA (2000)

[58] Etalle, S., Winsborough, W.H.: A posteriori compliance control. In: Proceedings of the 12th ACM symposium on Access control models and technologies. pp. 11–20. SACMAT '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1266840.1266843

[59] European Parliament: Directive 97/66/EC of the European Parliament and of the Council. http://www.aip-bg.org/lichnidanni/pdf/directive_97_66.pdf (Dec 1997)

[60] European Parliament: Directive 2002/58/EC of the European Parliament and of the Council. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2002:201:0037:0047:EN:PDF (Jul 2002)

[61] European Parliament: Directive 2006/24/EC of the European Parliament and of the Council. http://www.dataretention2010.net/files/legislation/dataretention/Directive_2006_24_EC_EN.pdf (Mar 2006)

[62] European Parliament: Directive 95/46/ec of the european parliament and of the council. http://ec.europa.eu/justice_home/fsj/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf (Jun 2009)

[63] European Parliament: Directive 95/46/ec of the european parliament and of the council. http://ec.europa.eu/justice_home/fsj/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf (2009)

[64] Evans, D., Twyman, A.: Flexible policy-directed code safety. Security and Privacy, IEEE Symposium on 0, 0032 (1999)

[65] Farmer, D., Venema, W.: Forensic Discovery. Addison-Wesley Professional Computing Series (December 2004)

[66] Ferraiolo, D., Kuhn, D.: Role-Based Access Control. 15th National Computer Security Conference pp. 554–563 (Oct 1992)

[67] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (2000)

[68] Fong, P.W.L.: Access control by tracking shallow execution history. Security and Privacy, IEEE Symposium on 0, 43 (2004)

[69] Forgy, C.: A network match routine for production systems. Working paper, Carnegie-Mellon University (1974)

[70] Forrester, Inc: The Forrester Wave(tm): Enterprise Service Bus, Q2 2011. http://www.oracle.com/us/corporate/analystreports/infrastructure/forrester-wave-esb-q2-2011-395900.pdf (april 2011)

[71] Frank Cohen: Discover SOAP encoding's impact on Web service performance. http://www.ibm.com/developerworks/webservices/library/ws-soapenc/ (2003)

[72] Fridsma, D.: The Role of SOA in the Nationwide Health Information Network: An Update. http://www.omg.org/news/meetings/workshops/SOA-HC/presentations-2011/15_KN_Fridsma.pdf (july 2011)

[73] Frisch, A.M., Peugniez, T.J., Doggett, A.J., Nightingale, P.W.: Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. J. Autom. Reason. 35, 143–179 (October 2005)

[74] Garfinkel, T.: Traps and pitfalls: Practical problems in in system call interposition based security tools. In: Proc. Network and Distributed Systems Security Symposium. The Internet Society, Reston, VA, USA (February 2003)

[75] Garfinkel, T., Pfaff, B., Rosenblum, M.: Ostia: A delegating architecture for secure system call interposition. In: In Proc. Network and Distributed Systems Security Symp. pp. 187–201. NDSS (feb 2004)

[76] Gartner, Inc.: Gartner says worldwide application infrastructure and middleware market revenue increased 7.3 percent in 2010. http://www.gartner.com/it/page.jsp?id=1632714 (April 2011)

[77] Gebel, G., Peterson, G.: Authentication and TOCTOU. http://analyzingidentity.com/2011/03/18/ (2011)

[78] Gheorghe, G., Crispo, B.: A Survey on Runtime Policy Enforcement Techniques and Their Implementations. http://eprints.biblio.unitn.it/archive/00002268 (2011), Technical Report DISI-11-447

[79] Gheorghe, G., Crispo, B., Carbone, R., Desmet, L., Joosens, W.: Deploy, Adjust and Readjust: Supporting Dynamic Reconfiguration of Policy Enforcement. In: ACM/IFIP/USENIX 12th International Middleware Conference (Dec 2011)

[80] Gheorghe, G., Crispo, B., Schleicher, D., Anstett, T., Leymann, F., Mietzner, R., Monakova, G.: Combining enforcement strategies in service oriented architectures. In: ICSOC. pp. 288–302 (2010)

[81] Gheorghe, G., Massacci, F., Neuhaus, S., Pretschner, A.: GoCoMM: a governance and compliance maturity model. In: Proceedings of the first ACM workshop on Information security governance. pp. 33–38. WISG '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1655168.1655175

[82] Gheorghe, G., Mori, P., Crispo, B., Martinelli, F.: Enforcing UCON Policies on the Enterprise Service Bus. In: OTM Conferences (2). pp. 876–893 (2010)

[83] Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In: IFIPTM. pp. 63–78 (2010)

[84] Ghormley, D.P., Petrou, D., Rodrigues, S.H., Anderson, T.E.: Slic: an extensibility system for commodity operating systems. In: Proceedings of the annual conference on USENIX Annual Technical Conference. pp. 4–4. ATEC '98, USENIX Association, Berkeley, CA, USA (1998)

[85] Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A secure environment for untrusted helper applications confining the wily hacker. In: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6. USENIX Association, Berkeley, CA, USA (1996), http://portal.acm.org/citation.cfm?id=1267569.1267570

[86] Goovaerts, T., De Win, B., Joosen, W.: Infrastructural support for enforcing and managing distributed application-level policies. Electron. Notes Theor. Comput. Sci. 197(1), 31–43 (2008)

[87] Goovaerts, T., Desmet, L., Joosen, W.: Scalable authorization middleware for service oriented architectures. In: ESSoS, Madrid, Spain (Feb 2011)

[88] Goovaerts, T., Win, B.D., Joosen, W.: Infrastructural support for enforcing and managing distributed application-level policies. In: Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007). vol. 197, pp. 31 – 43 (2008)

[89] Group, H.S.T.S.I.: Health Level Seven Security Services Framework Part 1: Basics of HL7 Security (Final Draft). ww.hl7.org/documentcenter/public/wg/secure/HL7BASICS3.RTF (1999)

[90] Guernic, G.L.: Automaton-based confidentiality monitoring of concurrent programs. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium. pp. 218–232. IEEE Computer Society, Washington, DC, USA (2007), http://portal.acm.org/citation.cfm?id=1270382.1270653

[91] Haldar, V., Chandra, D., Franz, M.: Practical, dynamic information flow for virtual machines. In: 2nd Intl. Workshop on Programming Language Interference and Dependence (PLID'05) (2005)

[92] Haldar, V., Chandra, D., Franz, M.: Dynamic taint propagation for Java. In: Proc. 21st Annual Computer Security Applications Conf. pp. 303–311. IEEE Computer Society, Washington, DC, USA (2005)

[93] Hamlen, K., Morrissett, G., Schneider, F.B.: Certified in-lined reference monitoring on .NET. In: PLAS Š06. ACM Press, New York, NY, USA (2006)

[94] Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: Proc. ACM SIGPLAN workshop on Programming languages and analysis for security. pp. 11–20. ACM, New York, NY, USA (2008)

[95] Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. 28(1), 175–205 (2006)

[96] Haupt, M., Mezini, M.: Micro-measurements for dynamic aspect-oriented systems. In: Weske, M., Liggesmeyer, P. (eds.) Object-Oriented and Internet-Based Technologies. Lecture Notes in Computer Science, vol. 3263, pp. 277–305. Springer Berlin Heidelberg (2004)

[97] Health Level Seven: Health level seven standards. http://www.hl7.org/implement/standards (1996)

[98] Hilty, M., Pretschner, A., Basin, D., Schaefer, C., Walter, T.: A policy language for distributed usage control. In: 12th European Symposium on Research in Computer Security (ESORICS 2007). LNCS, vol. 4734, pp. 531–546. Springer-Verlag (2007)

[99] HL7: HL7 Role-Based Access Control (RBAC) Role Engineering Process, v1.3. http://www.hl7.org/documentcenter/public/wg/secure/Stds_20070919_SW%2022.3_HL7%20RBAC%20Role%20Engineering%20Process%20v1.3.doc (sept 2007)

[100] Howard, M.: Fending off future attacks by reducing attack surface. http://msdn.microsoft.com/en-us/library/ms972812 (2003)

[101] Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J.W., Evans, D., Knight, J.C., Nguyen-Tuong, A., Rowanhill, J.: Secure and practical defense against code-injection attacks using software dynamic translation. In: Proc. of the 2nd Intl. Conf. on Virtual execution environments. pp. 2–12. ACM, New York, NY, USA (2006)

[102] IBM: IBM Tivoli Access Manager. http://www-01.ibm.com/software/tivoli/products/access-mgr-e-bus/ (Nov 2010)

[103] IEEE: IEEE 2011 International Workshop on Web Services in Healthcare and Application (WSHA 2011) (2011)

[104] IETF: Policy Framework Architecture. http://tools.ietf.org/html/draft-ietf-policy-arch-00 (Feb 1999)

[105] Internet2MiddlewareInitiative/MACE: Shibboleth 2. https://wiki.shibboleth.net/confluence/display/SHIB2/Home (4 2011)

[106] Ioannidis, S., Bellovin, S.M., Ioannidis, J., Keromytis, A.D., Anagnostakis, K.G., Smith, J.M.: Virtual private services: Coordinated policy enforcement for distributed applications. I. J. Network Security 4(1), 69–80 (2007)

[107] Irvine, C.E.: The Reference Monitor Concept as a Unifying Principle in Computer Security Education. http://www.cisr.nps.navy.mil/downloads/99paper_unifyseced.pdf

[108] Irwin, K., Yu, T., Winsborough, W.H.: On the modeling and analysis of obligations. In: CCS '06. pp. 134–143. ACM, New York, NY, USA (2006)

[109] Irwin, K., Yu, T., Winsborough, W.H.: Assigning responsibility for failed obligations. IFIP Intl. Federation for Information Processing 263, 327–342 (2008)

[110] Jager, T., Juraj, S.: How to break xml encryption. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 413–422. CCS '11, ACM, New York, NY, USA (2011)

[111] Jain, K., Sekar, R.: User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In: NDSS. IEEE Computer Society (2000)

[112] Jansen, W.: Directions in Security Metrics Research. Tech. Rep. NISTIR7564, National Institute of Standards and Technology (2009)

[113] Jaquith, A.: Security Metrics: Replacing Fear, Uncertainty, and Doubt. Addison-Wesley Professional (2007)

[114] Jaquith, A.: Security Metrics: Replacing Fear, Uncertainty, and Doubt. Addison-Wesley Professional (2007)

[115] Java Community Process: JSR 208: Java Business Integration. `http://www.jcp.org/en/jsr/detail?id=208` (2005)

[116] Juneja, G., Dournaee, B., Natoli, J., Birkel, S.: Service Oriented Architecture Demistified. Intel Press (2007)

[117] Karastoyanova, D., Wetzstein, B., van Lessen, T., Wutke, D., Nitzsche, J., Leymann, F.: Semantic Service Bus: Architecture and Implementation of a Next Generation Middleware. In: Proceedings of the Second International ICDE Workshop on Service Engineering (SEIW 2007). pp. 347–354. IEEE Computer Society (April 2007)

[118] Katt, B., Zhang, X., Breu, R., Hafner, M., Seifert, J.P.: A general obligation model and continuity: enhanced policy enforcement engine for usage control. In: Proc. SACMAT '08. pp. 123–132. ACM (2008)

[119] Kepser, S.: A simple proof of the Turing-Completeness of XSLT and XQuery. In: Extreme Markup Languages (2004)

[120] Khalaf, R., Karastoyanova, D., Leymann, F.: Pluggable framework for enabling the execution of extended bpel behavior. In: ICSOC Workshops. Lecture Notes in Computer Science, vol. 4907, pp. 376–387. Springer (2007)

[121] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting started with AspectJ. Commun. ACM 44(10), 59–65 (2001)

[122] Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium. pp. 191–206. USENIX Association, Berkeley, CA, USA (2002)

[123] Knuth, D.E.: The art of computer programming, vol. 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)

[124] Krafzig, D., Banke, K., Slama, D.: Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series). Prentice Hall PTR, Upper Saddle River, NJ, USA (2004)

[125] Krautsevich, L., Lazouski, A., Martinelli, F., Mori, P., Yautsiukhin, A.: Usage control, risk and trust. In: Katsikas, S., Lopez, J., Soriano, M. (eds.) Trust, Privacy and Security in Digital Business, Lecture Notes in Computer Science, vol. 6264, pp. 1–12. Springer Berlin / Heidelberg (2010)

[126] Krell, E., Krishnamurthy, B.: COLA: Customized overlaying. In: Proceedings of the Winter USENIX Conference. pp. 3–7. USENIX (1992)

[127] Krishnamurthi, S., Fisler, K.: Foundations of incremental aspect model-checking. ACM Trans. Softw. Eng. Methodol. 16 (April 2007), `http://doi.acm.org/10.1145/1217295.1217296`

[128] Laird, L.M., Brennan, M.C.: Software Measurement and Estimation: A Practical Approach (Quantitative Software Engineering Series). Wiley-IEEE Computer Society Pr (2006)

[129] Lam, L.C., Chiueh, T.c.: A general dynamic information flow tracking framework for security applications. In: Proceedings of the 22nd Annual Computer Security Applications Conference. pp. 463–472. IEEE Computer Society, Washington, DC, USA (2006), http://portal.acm.org/citation.cfm?id=1191820.1191901

[130] Lam, T., Minsky, N.: A collaborative framework for enforcing server commitments, and for regulating server interactive behavior in soa-based systems. In: Proceedings of the5th International Conference on Collaborative Computing: Networking, Applications and Worksharing. pp. 1–10 (Nov 2009)

[131] Lamanna, P.: Adaptive Security Policies Enforced by Software Dynamic Translation. Ph.D. thesis, Faculty of the School of Engineering and Applied Science University of Virginia (2002)

[132] Lampson, B.W.: Protection. Proc. 5th Princeton Conf. on Information Sciences and Systems 8(1), 18–24 (1971)

[133] Lauer, G.: Public-private partnerships help train healthcare workforce. http://www.californiahealthline.org/features/2011/publicprivate-partnerships-help-train-health-care-work-force.aspx (2011)

[134] Layer 7 Technologies: Policy Manager for XML Gateways. http://www.layer7tech.com/products/policy-manager-for-xml-gatewaysl (Nov 2010)

[135] Leiwo, J., Hänle, C., Homburg, P., Gamage, C., Tanenbaum, A.S.: A security design for a wide-area distributed system. In: Proceedings of the Second International Conference on Information Security and Cryptology. pp. 236–256. ICISC '99, Springer-Verlag, London, UK (2000), http://dl.acm.org/citation.cfm?id=646281.687819

[136] van Lessen, T., Leymann, F., Mietzner, R., Nitzsche, J., Schleicher, D.: A Management Framework for WS-BPEL. In: Proceedings of the 6th IEEE European Conference on Web Services 2008. pp. 187–196. IEEE Computer Society (November 2008)

[137] Leune, K., van den Heuvel, W.J., Papazoglou, M.: Exploring a multi-faceted framework for soc: how to develop secure web-service interactions? Research Issues on Data Engineering, Proc. 14th Intl. Workshop on pp. 56–61 (March 2004)

[138] Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust management framework. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy. pp. 114–130. IEEE Computer Society Press (May 2002)

[139] Li, P., Zdancewic, S.: Practical information-flow control in Web-based information systems. In: Proc. CSFW '05. pp. 2–15. IEEE Computer Society, Washington, DC, USA (2005)

[140] Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. Intl. Journal of Information Security 4, 2–16 (2005)

[141] Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. In: Proc. European Symp. Research in Computer Security (ESORICSŠ05. pp. 355–373. Springer (2005)

[142] Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12, 19:1–19:41 (January 2009), http://doi.acm.org/10.1145/1455526.1455532

[143] Ligatti, J., Reddy, S.: A theory of runtime enforcement, with results. In: Proceedings of the 15th European conference on Research in computer security. pp. 87–100. ESORICS'10, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1888881.1888889

[144] Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[145] Lucco, S., Sharp, O., Wahbe, R.: Omniware: A universal substrate for web programming. In: Proceedings of the 4th International World Wied Web Conferece. pp. 359–368. O'Reilly and Associates (1995)

[146] Maierhofer, A., Dimitrakos, T., Titkov, L., Brossard, D.: Extendable and adaptive message-level security enforcement framework. Networking and Services, 2006. ICNS '06 pp. 72–72 (2006)

[147] Manadhata, P.K., Wing, J.M.: An attack surface metric. IEEE Transactions in Software Engineering (2010), to appear

[148] Maullo, M., Calo, S.: Policy management: an architecture and approach. In: Systems Management, 1993., Proceedings of the IEEE First International Workshop on. pp. 13 –26 (apr 1993)

[149] Meyerovich, L.A., Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. Security and Privacy, IEEE Symposium on 0, 481–496 (2010)

[150] Mezini, M., Ostermann, K.: Conquering aspects with Caesar. In: Proc. 2nd intl. conf. on Aspect-oriented software development. pp. 90–99. ACM, New York, NY, USA (2003)

[151] Michael Hafner, Mukhtiar Memon, R.B.: SeAAS - a reference architecture for security services in SOA. Journal of Universal Computer Science 15:15, 2916–2936 (2009)

[152] Michiels, S., Verslype, K., Joosen, W., De Decker, B.: Towards a software architecture for drm. In: Proceedings of the 5th ACM workshop on Digital rights management. pp. 65–74. DRM '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1102546.1102559

[153] Miller, B.P., Christodorescu, M., Iverson, R., Kosar, T., Mirgorodskii, A., Popovici, F.: Playing inside the black box: Using dynamic instrumentation to create security holes. Parallel Processing Letters 11(2 & 3), 267–280 (June & September 2001)

[154] Mitre: Common Vulnerabilities and Exposures. http://cve.mitre.org/ (2011)

[155] MITRE Corporation: Electronic health records overview. http://www.ncrr.nih.gov/publications/informatics/ehr.pdf (2006)

[156] Mont, M., Pearson, S., Bramhall, P.: Towards accountable management of identity and privacy: sticky policies and enforceable tracing services. In: Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on. pp. 377 – 382 (sept 2003)

[157] Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel. In: WWW. pp. 815–824 (2008)

[158] Myers, A.C.: Jflow: practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 228–241. POPL '99, ACM, New York, NY, USA (1999)

[159] Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th international conference on Software engineering. pp. 452–461. ICSE '06, ACM, New York, NY, USA (2006)

[160] Nair, S.K., Simpson, P.N.D., Crispo, B., Tanenbaum, A.S.: A virtual machine based information flow control system for policy enforcement. Electron. Notes Theor. Comput. Sci. 197(1), 3–16 (2008)

[161] National Institute of Standards and Technology: NIST definition of Cloud Computing (Draft). http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf (2011)

[162] Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 89–100. PLDI '07, ACM, New York, NY, USA (2007)

[163] Newsome, J., Song, D.X.: Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In: NDSS. IEEE Computer Society (2005)

[164] NHS: Nhs information centre. http://www.ic.nhs.uk/about-us (2011)

[165] OASIS: OASIS Web Services Resource Framework (WSRF). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf (2004)

[166] OASIS: Web Services Security: SOAP message security 1.1. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf (2004)

[167] OASIS: Saml 2.0. http://saml.xml.org/saml-specifications (2005)

[168] OASIS: Universal DDI Version 3.0.2. http://uddi.org/pubs/uddi-v3.0.2-20041019.pdf (2005)

[169] OASIS: Reference Model for Service Oriented Architecture 1.0. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf (october 2006)

[170] OASIS: OASIS Web Service Business Execution Language. http://www.oasis-open.org/committees/wsbpel/ (2007)

[171] OASIS: WS-SecureConversation 1.3. http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html (march 2007)

[172] OASIS: WS-Trust 1.3. http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html (2007)

[173] OASIS: Reference Architecture Foundation for Service Oriented Architecture 1.0. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-cd-02.pdf (2009)

[174] OASIS eXtensible Access Control Markup Language (XACML) TC: extensible access control markup language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#XACML20 (May 2009)

[175] de Oliveira, A.S., Wang, E.K., Kirchner, C., Kirchner, H.: Weaving rewrite-based access control policies. In: Proc. ACM workshop on Formal methods in security engineering. pp. 71–80. ACM, New York, NY, USA (2007)

[176] Opincaru, C., Gheorghe, G.: Service oriented security architecture. In: EMISA. pp. 61–74 (2007)

[177] Oracle: Enterprise Java Beans, 3.0 specification. http://www.oracle.com/technetwork/java/docs-135218.html (2011)

[178] O'Reilly, T.: What is Web 2.0. http://oreilly.com/lpt/a/6228 (2005)

[179] Ousterhout, J.K., Levy, J.Y., Welch, B.B.: The safe-tcl security model. In: Mobile Agents and Security. pp. 217–234. Springer-Verlag, London, UK (1998), http://portal.acm.org/citation.cfm?id=648051.746190

[180] Palmer, D.: Dynamic aspect-oriented programming in an untrusted environment. In: Workshop on Foundations of Middleware Technologies (collocated with DOA'02). Springer Verlag (November 2002)

[181] Pandey, R., Hashii, B.: Providing fine-grained access control for mobile programs through binary editing. Tech. rep., Proce. of the European Conf. on Object-Oriented Programming (1999)

[182] Papazoglou, M., van den Heuvel, W.J.: Service oriented architectures: Approaches, technologies and research issues. The VLDB Journal 16(3), 389–415 (July 2007)

[183] Park, J., Sandhu, R.: Towards usage control models: beyond traditional access control. In: Proceedings of the seventh ACM symposium on Access control models and technologies. pp. 57–64. SACMAT '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/507711.507722

[184] Park, J., Sandhu, R.: The UCON$_{ABC}$ usage control model. ACM Trans. Inf. Syst. Secur. 7(1), 128–174 (2004)

[185] Park, J., Sandhu, R.: The UCON$_{ABC}$ usage control model. ACM Trans. Inf. Syst. Secur. 7(1), 128–174 (2004)

[186] Parliament, U.: The data protection act. http://www.legislation.gov.uk/ukpga/1998/29/contents (1998)

[187] Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. "big"' web services: making the right architectural decision. In: Proceeding of the 17th international conference on World Wide Web. pp. 805–814. WWW '08, ACM, New York, NY, USA (2008)

[188] Popescu, B.C., van Steen, M., Tanenbaum, A.S.: A security architecture for object-based distributed systems. In: Proceedings of the 18th Annual Computer Security Applications Conference. pp. 161–. ACSAC '02, IEEE Computer Society, Washington, DC, USA (2002), http://dl.acm.org/citation.cfm?id=784592.784804

[189] Popovici, A., Alonso, G., Gross, T.: AOP support for mobile systems. In: OOPSLA'01 Workshop on Advanced Separation of Concerns in Object-Oriented Systems. ACM International (2001)

[190] Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: Proceedings of the 1st international conference on Aspect-oriented software development. pp. 141–147. AOSD '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/508386.508404

[191] Povey, D.: Optimistic security: A new access control paradigm. In: In Proceedings of 1999 New Security Paradigms Workshop. pp. 40–45. ACM Press (1999)

[192] Pretschner, A., Hilty, M., Basin, D., Schaefer, C., Walter, T.: Mechanisms for usage control. In: Proceedings of the 2008 ACM symposium on Information, computer and communications security. pp. 240–244. ASIACCS '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1368310.1368344

[193] Pretschner, A., Schütz, F., Schaefer, C., Walter, T.: Policy evolution in distributed usage control. In: 4th Intl. Workshop on Security and Trust Management (06 2008)

[194] Pretschner, A., Hilty, M., Basin, D.: Distributed usage control. Commun. ACM 49, 39–44 (September 2006), http://doi.acm.org/10.1145/1151030.1151053

[195] Pretschner, A., Massacci, F., Hilty, M.: Usage control in service-oriented architectures. In: Lambrinoudakis, C., Pernul, G., Tjoa, A. (eds.) Trust, Privacy and Security in Digital Business. Lecture Notes in Computer Science, vol. 4657, pp. 83–93. Springer Berlin / Heidelberg (2007)

[196] Provos, N.: Improving host security with system call policies. In: SSYM'03: Proc. of the 12th Conf. on USENIX Security Symp. USENIX Association, Berkeley, CA, USA (2003)

[197] Regio, M.: Web Services enablement for Healthcare HL7 Applications - Web Service Basic Profile Reference Implementation. http://msdn.microsoft.com/en-us/library/ms954603.aspx (2005)

[198] Reichert, M., Dadam, P.: Adeptflex: Supporting dynamic changes of workflow without loosing control. Journal of Intelligent Information Systems 10, 93–129 (1998)

[199] Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. T. Petri Nets and Other Models of Concurrency 2, 115–135 (2009)

[200] Ribeiro, C., Zúquete, A., Ferreira, P., Guedes, P.: Spl: An access control language for security policies with complex constraints. In: In Proceedings of the Network and Distributed System Security Symposium. pp. 89–107 (1999)

[201] Roger, M., Goubault-Larrecq, J.: Log auditing through model-checking. In: Proceedings of the 14th IEEE workshop on Computer Security Foundations. CSFW '01, IEEE Computer Society, Washington, DC, USA (2001), http://portal.acm.org/citation.cfm?id=872752.873518

[202] Samarati, P., Vimercati, S.D.C.d.: Access control: Policies, models, and mechanisms. In: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures. pp. 137–196. FOSAD '00, Springer-Verlag, London, UK (2001), http://portal.acm.org/citation.cfm?id=646206.683112

[203] Sandhu, R., Ferraiolo, D., Kuhn, R.: The nist model for role-based access control: towards a unified standard. In: Proceedings of the fifth ACM workshop on Role-based access control. pp. 47–63. RBAC '00, ACM, New York, NY, USA (2000), http://doi.acm.org/10.1145/344287.344301

[204] Sandhu, R., Samarati, P.: Access control: principle and practice. Communications Magazine, IEEE 32(9), 40 –48 (sep 1994)

[205] Satyanarayanan, M.: The evolution of coda. ACM Trans. Comput. Syst. 20, 85–124 (May 2002)

[206] Schaefer, C.: Usage control reference monitor architecture. In: Security, Privacy and Trust in Pervasive and Ubiquitous Computing, 2007. SECPerU 2007. Third International Workshop on. pp. 13 –18 (july 2007)

[207] Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)

[208] Schneider, F.B., Morrisett, J.G., Harper, R.: A language-based approach to security. In: Informatics - 10 Years Back. 10 Years Ahead. pp. 86–101. Springer-Verlag, London, UK (2001)

[209] Scott, K., Kumar, N., Childers, B.R., Davidson, J.W., Soffa, M.L.: Overhead reduction techniques for software dynamic translation. Parallel and Distributed Processing Symposium, International 11, 200a (2004)

[210] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J.W., Soffa, M.L.: Retargetable and reconfigurable software dynamic translation. In: In CGO Š03: Proc. of the Intl. Symp. on Code generation and optimization. pp. 36–47. IEEE Computer Society (2003)

[211] Scott, K., Davidson, J.: Safe Virtual Execution Using Software Dynamic Translation. In: ACSAC '02: Proc. of the 18th Annual Computer Security Applications Conf. IEEE Computer Society, Washington, DC, USA (2002)

[212] Serban, C., McMillin, B.: Run-time security evaluation (rtse) for distributed applications. In: Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on. pp. 222 –232 (may 1996)

[213] bo Shen, H., Hong, F.: An attribute-based access control model for web services. In: Parallel and Distributed Computing, Applications and Technologies, 2006. PDCAT '06. Seventh International Conference on. pp. 74 –79 (dec 2006)

[214] Shostack, A., Stewart, A.: The New School of Information Security. Addison-Wesley Professional (2008)

[215] Silver, S.M.: Implementation and analysis of software based fault isolation. Tech. rep., Darmouth College, Hanover, NH, USA (1996)

[216] Small, C., Seltzer, M.: Misfit: constructing safe extensible systems. Concurrency, IEEE 6(3), 34–41 (nov 1998)

[217] Song, Y., Fleisch, B.D.: Utilizing binary rewriting for improving end-host security. IEEE Trans. Parallel Distrib. Syst. 18(12), 1687–1699 (2007)

[218] for Standards, N.I., Technology: Common criteria for information technology security evaluation. `http://www.commoncriteriaportal.org/cc/` (2001)

[219] van Steen, M., Homburg, P., Tanenbaum, A.: Globe: a wide area distributed system. Concurrency, IEEE 7(1), 70 –78 (jan-mar 1999)

[220] Sterne, D.F.: On the Buzzword "Security Policy". Security and Privacy, IEEE Symposium on 0, 219 (1991)

[221] Svirskas, A., Isachenkova, J., Molva, R.: Towards secure and trusted collaboration environment for european public sector. Collaborative Computing: Networking, Applications and Worksharing, 2007. CollaborateCom 2007. International Conference on pp. 49–56 (Nov 2007)

[222] Systems, S.: Sun's XACML Implementation. `http://sunxacml.sourceforge.net` (2006)

[223] Thion, R., Metayer, D.L.: Flavor: A formal language for a posteriori verification of legal rules. Policies for Distributed Systems and Networks, IEEE International Workshop on 0, 1–8 (2011)

[224] Trade, U.S., Agency, D.: United states and china launch public private partnership on healthcare. `http://www.ustda.gov/news/pressreleases/2011/EastAsiaEurasia/China/USChinaHCP_011911.asp` (2011)

[225] Tripakis, S.: A combined on-line/off-line framework for black-box fault diagnosis. In: Runtime Verification (RV'09). pp. 152–167 (2009)

[226] Tsai, W., Zhou, X., Wei, X.: A policy enforcement framework for verification and control of service collaboration. Information Systems and E-Business Management 6, 83–107 (2008), 10.1007/s10257-007-0059-8

[227] Tsai, W.T., Zhou, X., Chen, Y.: Soa simulation and verification by event-driven policy enforcement. In: ANSS-41 '08: Proceedings of the 41st Annual Simulation Symposium (anss-41 2008). pp. 165–172. IEEE Computer Society, Washington, DC, USA (2008)

[228] Türpe, S.: What is the shape of your security policy? security as a classification problem. In: NSPW '09: Proceedings of the 2009 workshop on New security paradigms workshop. pp. 23–36. ACM, New York, NY, USA (2009)

[229] Ueno, K., Tatsubori, M.: Early capacity testing of an enterprise service bus. Web Services, IEEE International Conference on 0, 709–716 (2006)

[230] U.K. Parliament: The privacy and electronic communications (ec directive) regulations 2003. `http://www.opsi.gov.uk/si/si2003/20032426.htm` (Jun 2009)

[231] United States Department of Defence: Trusted Computer System Evaluation Criteria. `http://www.fas.org/irp/nsa/rainbow/std001.htm` (1985)

[232] United States Government: Sarbanes-Oxley Act of 2002. `http://www.gpo.gov/fdsys/pkg/PLAW-107publ204/content-detail.html` (2002)

[233] U.S. Congress: Health Insurance Portability and Accountability Act. `http://www.gpo.gov/fdsys/search/pagedetails.action?granuleId=CRPT-104hrpt736&packageId=CRPT-104hrpt736` (1996)

[234] US Department of Health and Human Services: Partnership for patients: Better care, lower costs. `http://www.healthcare.gov/center/programs/partnership` (2011)

[235] Vachharajani, N., Bridges, M.J., Chang, J., Rangan, R., Ottoni, G., Blome, J.A., Reis, G.A., Vachharajani, M., August, D.: RIFLE: An architectural framework for user-centric information-flow security. In: In Proc. 37th Annual IEEE/ACM Intel. Symp. on Microarchitecture. pp. 243–254. IEEE Computer Society (2004)

[236] Vanoverberghe, D., Piessens, F.: A caller-side inline reference monitor for an object-oriented intermediate language. In: Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems. pp. 240–258. FMOODS '08, Springer-Verlag, Berlin, Heidelberg (2008)

[237] Vanoverberghe, D., Piessens, F.: Security enforcement aware software development. Information and Software Technology 51(7), 1172 – 1185 (2009), `http://www.sciencedirect.com/science/article/B6V0B-4RV17HM-1/2/2b9c3f8c68ebc9e4dd6c1cf5e556202b`, special Section: Software Engineering for Secure Systems - Software Engineering for Secure Systems

[238] Viega, J., Bloch, J.T., Ch, P.: Applying aspect-oriented programming to security. Cutter IT Journal 14(2) (February 2001)

[239] Vordel Limited: The Vordel Gateway (6 2011)

[240] W3C: XML Encryption and Syntax Processing. `http://www.w3.org/TR/xmlenc-core/` (2002)

[241] W3C: Web services policy 1.5 - attachment. `http://www.w3.org/TR/ws-policy-attach/` (2007)

[242] W3C: XML Signature Syntax and Processing, Second Edition. `http://www.w3.org/TR/xmldsig-core/` (2008)

[243] Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: SOSP '93: Proc. ACM symp. on Operating systems principles. pp. 203–216. ACM, New York, NY, USA (1993)

[244] Web Services Policy Working Group: Web services policy 1.5 - framework. `http://www.w3.org/TR/2007/REC-ws-policy-20070904/` (may 2009)

[245] Wei, Q.: Towards Improving the Availability and Performance of Enterprise Authorization Systems. Ph.D. thesis, University of British Columbia (2009)

[246] Win, B.D., Joosen, W., Piessens, F.: Developing secure applications through aspect-oriented programming. In: Aspect-Oriented Software Development. pp. 633–650. Addison-Wesley (2005)

[247] World Wide Web Consortium: Web Service Description Language 2.0. `http://wwww.w3c.org/standards/techs/wsdl` (2007)

[248] World Wide Web Consortium: Web Application Description Language. `http://www.w3.org/Submission/wadl` (2009)

[249] World Wide Web Consortium Working Group: Web Services Architecture. `http://www.w3c.org/TR/ws-arch/` (2004)

[250] World Wide Web Consortium Working Group: SOAP version 1.2. `http://www.w3.org/TR/soap12-part1/` (2007)

[251] Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: Proc. 15th conf. USENIX Security Symp. USENIX Association, Berkeley, CA, USA (2006)

[252] Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native Client: A sandbox for portable, untrusted x86 native code. Communications of the ACM 53(1), 91–99 (2010)

[253] Yoshihama, S., Yoshizawa, T., Watanabe, Y., Kudo, M., Oyanagi, K.: Dynamic information flow control architecture for web applications. In: Proc. ESORICS. LNCS, vol. 4734, pp. 267–282. Springer (2008)

[254] Yu, D., Chander, A., Islam, N., Serikov, I.: Javascript instrumentation for browser security. In: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 237–249. POPL '07, ACM, New York, NY, USA (2007)

[255] Zhang, X., Nakae, M., Covington, M.J., Sandhu, R.: Toward a usage-based security framework for collaborative computing systems. ACM Trans. Inf. Syst. Secur. 11, 3:1–3:36 (February 2008)

Appendix A

Comparison of enforcement implementations

Table A.1 shows the most relevant implementations we have surveyed: for each implementation we mention its type (recognizer (R) or sanitizer (S)), its operation level (high or low), and where it belongs in the taxonomy in Figure 8.4.

Technique	Implementation	Type	Level
Call Interposition	Cola [126]	R	low
	Janus [85]	R	low
	TRON [26]	R	low
	Ostia [75]	S	low
	Systrace [196]	S	low
	Janus2 [111]	S	low
	SLIC [84]	R	low
	ChakraVyuha [46]	R	low
	SudDomain [41]	R	low
	[236]	S	high
	[88]	S	high
Safe Interpreter	Safe-Tcl [179]	R	high
	SecureJS [9]	R	high
	CoreScript [254]	S	high
	Conscript [149]	R	high
SFI	[243]	S	low
	Naccio [64]	R	low
	MiSFIT [216]	S,R	low
	Omniware [145]	S	low
	NativeClient [252]	S	low
	CFI [2]	R	low
	XFI [56]	R	low
	[215]	S	low
IRM	SASI [54]	R	high
	[55]	R	high
	[57]	R	high
	[93]	R	high
	Polymer [23]	S	high
	[236]	S	high
SDT	Valgrind [162]	S,R	high
	Strata [211]	S	high
	Java	S,R	high
	DynamoRIO [122]	R	low
Dynamic Weavers	Prose [189]	S,R	high
	Wool [190]	S,R	high
	Tom [175]	S	high
	SPoX [94]	R	high
Data Flow Trackers	RIFLE [235]	S	low
	TaintDroid [52]	S	low
	[38]	S,R	high
	[129]	S,R	high
	[91]	R	high
	[90]	S	high
	TaintCheck [163]	R	low
	Trishul [160]	S,R	high
	Dytan [39]	S,R	high

Table A.1: Summary on runtime enforcement implementations

Appendix B

Syntax of the xESB policy language

The syntax of the xESB policy language, as presented in Chapter 4, is the result of a previous collaboration [83]. The code below implements the syntax as input to the JavaCC parser generator.

```
options {
    JDK_VERSION = "1.5";
    STATIC = false;
}
PARSER_BEGIN(EsbEcaParser)
package it.unitn.disi.esb_eca.compiler;

import java.io.InputStream;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.HashSet;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import it.unitn.disi.esb_eca.vm.Instruction;
import it.unitn.disi.esb_eca.vm.InstructionFactory;
import it.unitn.disi.esb_eca.vm.StackElement;
import it.unitn.disi.esb_eca.vm.VmInteger;
import it.unitn.disi.esb_eca.vm.VmString;

/** Compiles an ESB ECA ruleset into binary form.
 *
 * Usage: EsbEcaParser eca-file
 * Output: ecb-file
 */
public class EsbEcaParser {
    public static final String version = "0.2alpha";
    private SymbolTable symbolTable = new SymbolTable();
```

```
private InstructionList program = new InstructionList();
private List<Counter> counters = new LinkedList<Counter>();
private List<Hash> hashes = new LinkedList<Hash>();
private List<Timer> timers = new LinkedList<Timer>();
private List<InstructionList> conditions = new LinkedList<InstructionList>();

/** Captures whether we're parsing a rule or an obligation.
 *
 * A rule can contain actions that lead to verdicts; an obligation cannot.
 */
private boolean inRule;

/** Source file containing the policies in ECA format. */
private static String sourceFile = null;

/** Input file as a stream. */
private static InputStream in = null;

/** Destination file containing the compiled policies in ECB format. */
private static String destFile = null;

public enum EventType {
   REQUEST,
       REPLY,
}

private static String stripQuotes(String s) {
    return s.substring(1, s.length() - 1);
}

private List<Counter> getCounters() {
    return counters;
}

private List<Timer> getTimers() {
    return timers;
}

private List<Hash> getHashes() {
    return hashes;
}

private List<StackElement[]> getConditions() {
    if (conditions.size() == 0) {
        return null;
    } else {
        List<StackElement[]> ret = new LinkedList<StackElement[]>();

        for (InstructionList l : conditions) {
            ret.add(l.getInstructions());
        }

        return ret;
    }
}
```

```
/** Returns the compiled ruleset.
 *
 * Call this method after calling Ruleset(). You will get the compiled
 * ruleset which is suitable for initializing a VirtualMachine. Calling
 * go() on the VM will actually run the ruleset on a message.
 *
 * @return the compiled ruleset.
 */
public StackElement[] getProgram() {
    return program.getInstructions();
}

public static void main (String args[]) {
    boolean verbose = false;

    if (args.length > 3) {
        System.out.println("Usage:");
        System.out.println("  java EsbEcaParser [-v] [inputfile] [outputfile]");
        System.out.println("  -v\tbe verbose");
        return;
    }

    int i;
    for (i = 0; i < args.length; i++) {
        if (args[i].startsWith("-")) {
            if (args[i].equals("-v")) {
                verbose = true;
            }
        } else {
            break;
        }
    }

    try {
        if (i >= args.length) {
            sourceFile = "[stdin]";
            in = System.in;
        } else {
            sourceFile = args[i++];
            in = new java.io.FileInputStream(sourceFile);
        }

        if (i >= args.length) {
            destFile = "aout.ecb";
        } else {
            destFile = args[i++];
        }
    } catch (java.io.FileNotFoundException e) {
        System.out.println("ESB ECA Parser Version " + version + ":  File "
            + sourceFile + " not found.");
        return;
    }

    if (in == null) {
```

```
            return;
        }

        if (verbose) {
            System.out.println("ESB ECA Parser Version " + version + ": " + sourceFile);
        }

        EsbEcaParser parser = new EsbEcaParser(in);

        try {
            parser.Ruleset();
                EcbFile f = new EcbFile(8);
            f.setProgram(parser.getProgram());
            f.setCounters(parser.getCounters());
            f.setTimers(parser.getTimers());
            f.setHashes(parser.getHashes());
            f.setConditions(parser.getConditions());
                f.write(destFile);
        } catch(ParseException e) {
            System.out.println("Encountered errors during parse.");
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Encountered other errors during parse.");
            e.printStackTrace();
        }
    }
}

PARSER_END(EsbEcaParser)

SKIP: { " " | "\t" | "\n" | "\r" }

// When a /* is seen in the DEFAULT state, skip it and switch to the IN_COMMENT state
SKIP : { "/*": IN_COMMENT }

// When any other character is seen in the IN_COMMENT state, skip it.
<IN_COMMENT> SKIP : { < ~[] > }

// When a */ is seen in the IN_COMMENT state, skip it and switch back to the DEFAULT state
<IN_COMMENT> SKIP : { "*/": DEFAULT }

/* According to the JavaCC "Tips for writing a good JavaCC lexical
 * specification" https://javacc.dev.java.net/doc/lexertips.html,
 * the lexical analyzer will be faster if the reserved words are mentioned
 * in decreasing order of length.  That may be, but I refuse to count
 * characters; that's what computers are there for. */
TOKEN: { <DEFAULTACTION: "default-action"> }

TOKEN: { <RULE: "rule"> }
TOKEN: { <OBLIGATION: "obligation"> }
TOKEN: { <EVENT: "if"> }
TOKEN: { <CONDITION: "when"> }
TOKEN: { <ACTION: "do"> }

/* Reserved words for events */
```

```
TOKEN: { <REQUEST: "invocation"> }
TOKEN: { <REPLY: "response"> }

/* Reserved words for conditions */
TOKEN: { <SOURCE: "source"> }
TOKEN: { <DESTINATION: "destination"> }
TOKEN: { <HEADER: "h"> }

TOKEN: { <COUNTER : "counter"> }
TOKEN: { <TIMER: "timer"> }
TOKEN: { <HASH: "hash"> }

TOKEN: { <OR: "||"> }
TOKEN: { <AND: "&&"> }
TOKEN: { <NOT: "!"> }
TOKEN: { <EQ: "=="> }
TOKEN: { <NE: "!="> }
TOKEN: { <GT: ">"> }
TOKEN: { <GE: ">="> }
TOKEN: { <LT: "<"> }
TOKEN: { <LE: "<="> }

TOKEN: { <PLUS: "+"> }
TOKEN: { <MINUS: "-"> }
TOKEN: { <MUL: "*"> }
TOKEN: { <DIV: "/"> }
TOKEN: { <REM: "%"> }

TOKEN: { <EQUALS: "equals"> }
TOKEN: { <CONTAINS: "contains"> }
TOKEN: { <STARTSWITH: "startswith"> }
TOKEN: { <SUBSTRING: "substring"> }

TOKEN: { <EXTERNALPLACEHOLDER: "external-condition"> }
TOKEN: { <COUNTERPLACEHOLDER: "counter-condition"> }
TOKEN: { <HISTORYPLACEHOLDER: "history-condition"> }

/* Reserved words for actions */
TOKEN: { <BLOCK: "block"> }
TOKEN: { <ALLOW: "allow"> }
TOKEN: { <DELAY: "delay"> } /* Delays are always in the same units */
TOKEN: { <UNTIL: "until"> }
TOKEN: { <MODIFY: "modify"> }
/* Reserved words for timer operations */
TOKEN: { <ARM: "arm"> }
TOKEN: { <DISARM: "disarm"> }
TOKEN: { <FIRE: "fire"> }
TOKEN: { <AFTER: "after"> }
TOKEN: { <SECONDS: "sec"> }
TOKEN: { <MILLISECONDS: "msec"> }
TOKEN: { <MICROSECONDS: "usec"> }
TOKEN: { <NEXT: "next"> }
TOKEN: { <MONTH: "month"> }

TOKEN: { <ISARMED: "armed"> }
```

```
TOKEN: { <ISDISARMED: "disarmed"> }
TOKEN: { <ISFIRED: "fired"> }
TOKEN: { <ISUNFIRED: "unfired"> }

TOKEN: { <GETS: ":="> }
TOKEN: { <INC: "+="> }
TOKEN: { <DUPLICATE: "duplicate"> }
TOKEN: { <UPDATECOUNTER: "update-counter"> }
TOKEN: { <CLEAR: "clear"> }

/* Numbers and strings */
TOKEN: { <STRING: "\"" ( ~["\"","\\","\n","\r"] | "\\" ( ["n","t","b","r","f","\\","\'","\""]
    | ["0"-"7"] (["0"-"7"])?  | ["0"-"3"] ["0"-"7"] ["0"-"7"] | ( ["\n","\r"] | "\r\n")))* "\""
    "> }
TOKEN: { <NUMBER: (["+","-"])?(["0"-"9"])+> }
TOKEN : {
 <ID: <LETTER> (<LETTER> | <DIGIT>)*>
|  <#LETTER: ["A"-"Z","_","a"-"z"]>
|  <#DIGIT: ["0"-"9"]>
}

/*TOKEN: { <ID: ["A"-"Z","a"-"z"](["A"-"Z","a"-"z","0"-"9"])*> }*/

void Ruleset() : {
    InstructionList defaultAction = null;
}
{
    ( defaultAction = DefaultAction() ) ? ( RuleOrObligation() | Declaration() )* {
        if (defaultAction == null) {
            defaultAction = new InstructionList();
            defaultAction.add(InstructionFactory.get(Instruction.Opcode.STOP));
        }
        program.add(defaultAction);
    }
}

InstructionList DefaultAction() : {
    InstructionList ret;
}
{
    <DEFAULTACTION> { inRule = true; } "{" ret = Actions() "}" {
        return ret;
    }
}

/* *** DECLARATIONS *** */
void Declaration() : {}
{
    ( CounterDeclaration ()
    | TimerDeclaration()
    | HashDeclaration()
    )
}

void CounterDeclaration() : {
```

```
    Token t;
    Token num = null;
}
{
    <COUNTER> t=<ID> ( "=" num = <NUMBER> )? ";" {
        long initialValue = num == null ? 0L : Long.parseLong(num.image);
        Counter s = new Counter(t.image, t.beginLine, t.beginColumn, initialValue);
        try {
            symbolTable.addSymbol(s);
        } catch (SymbolAlreadyPresentException e) {
            Symbol knownSymbol = symbolTable.getSymbol(t.image);
            throw new ParseException("Identifier␣\""
                + knownSymbol.getName() + "\"␣already␣used␣in␣line␣"
                + knownSymbol.getDefinitionLine() + ":" + knownSymbol.getDefinitionColumn());
        }
        counters.add(s);
    }
}

void TimerDeclaration() : {
    Token t;
    boolean initiallyArmed = false;
    long ns = -1;
}
{
    <TIMER> t=<ID> ( "=" ns = RelativeTime() { initiallyArmed = true; } ) ? ";" {
        Timer s;
        if (initiallyArmed) {
            s = new Timer(t.image, t.beginLine, t.beginColumn, ns);
        } else {
            s = new Timer(t.image, t.beginLine, t.beginColumn);
        }
        try {
            symbolTable.addSymbol(s);
        } catch (SymbolAlreadyPresentException e) {
            Symbol knownSymbol = symbolTable.getSymbol(t.image);
            throw new ParseException("Identifier␣\""
                + knownSymbol.getName() + "\"␣already␣used␣in␣line␣"
                + knownSymbol.getDefinitionLine() + ":" + knownSymbol.getDefinitionColumn());
        }
        timers.add(s);
    }
}

void HashDeclaration() : {
    Token t;
    Token num = null;
}
{
    <HASH> t=<ID> ( "=" num = <NUMBER> )? ";" {
        long initialValue = num == null ? 0L : Long.parseLong(num.image);
        Hash s = new Hash(t.image, t.beginLine, t.beginColumn, initialValue);
        try {
            symbolTable.addSymbol(s);
        } catch (SymbolAlreadyPresentException e) {
```

```
            Symbol knownSymbol = symbolTable.getSymbol(t.image);
            throw new ParseException("Identifier␣\""
                + knownSymbol.getName() + "\"␣already␣used␣in␣line␣"
                + knownSymbol.getDefinitionLine() + ":" + knownSymbol.getDefinitionColumn());
        }
        hashes.add(s);
    }
}



/* *** RULES AND OBLIGATIONS *** */
void RuleOrObligation() : {
    EventType e;
    InstructionList c;
    InstructionList a;
    Token rOrO;
}
{
    ( rOrO = <RULE> | rOrO = <OBLIGATION> ) { inRule = rOrO.kind == RULE; }
      "{" e = EventPart() c = ConditionPart() a = ActionPart() "}" {
        InstructionList evalCondition = new InstructionList();
            evalCondition.add(new VmInteger(a.size()));
        evalCondition.add(InstructionFactory.get(Instruction.Opcode.CNJUMP));

        InstructionList evalEvent = new InstructionList();
            evalEvent.add(InstructionFactory.get(Instruction.Opcode.EVENTTYPE));
            evalEvent.add(new VmInteger(c.size() + evalCondition.size() + a.size()));
            if (e == EventType.REQUEST) {
               evalEvent.add(InstructionFactory.get(Instruction.Opcode.CJUMP));
            } else {
               evalEvent.add(InstructionFactory.get(Instruction.Opcode.CNJUMP));
        }

        program.add(evalEvent);
        program.add(c);
        program.add(evalCondition);
        program.add(a);
    }
}

EventType EventPart() : {
    EventType t;
}
{
    <EVENT> t = EventExpression() {
        return t;
    }
}

EventType EventExpression() : {
     EventType ret;
}
{
```

```
    ( <REQUEST> { ret = EventType.REQUEST; }
    | <REPLY> { ret = EventType.REPLY; }
    ) {
        return ret;
    }
}

InstructionList ConditionPart() : {
    InstructionList condition = null;
}
{
    <CONDITION> "{" condition = ConditionExpression() "}" {
        return condition;
    }
}


InstructionList ConditionExpression() : {
    ArrayList<InstructionList> disjunctions = new ArrayList<InstructionList>();
    int totalSize = 0;
    InstructionList element;
}
{
    element = ConditionTerm() {
        disjunctions.add(element);
        totalSize += element.size();
    } (
        LOOKAHEAD(1) ConditionOrOperator() element = ConditionTerm() {
            disjunctions.add(element);
            totalSize += element.size();
        }
    )*
    {
        boolean rest = false;
        /* Offset to end of instructions (= jump target for short-circuit)
             * is initially total size of disjunction code, plus (number of
         * disjunctions - 1)*(size of short-circuit code), and the latter
         * is 2. */
        int offset = totalSize + (disjunctions.size() - 1)*2;
            InstructionList ret = new InstructionList();
            for (InstructionList e : disjunctions) {
                if (rest) {
                    ret.add(new VmInteger(offset));
                    ret.add(InstructionFactory.get(Instruction.Opcode.CPJUMP));
                }
                ret.add(e);
                rest = true;
                /* New offset is diminished by length of inserted code, which is
                 * length of disjunction + length of short-circuit code. */
                offset -= e.size() + 2;
            }
        return ret;
    }
}
```

```
void ConditionOrOperator() : {}
{
    <OR>
}

InstructionList ConditionTerm() : {
    ArrayList<InstructionList> conjunctions = new ArrayList<InstructionList>();
    int totalSize = 0;
    InstructionList element;
}
{
    element = ConditionFactor() {
        conjunctions.add(element);
        totalSize += element.size();
    } (
        LOOKAHEAD(1) ConditionAndOperator() element = ConditionFactor() {
            conjunctions.add(element);
            totalSize += element.size();
        }
    )*
    {
        boolean rest = false;
            /* Offset to end of instructions (= jump target for short-circuit)
             * is initially total size of conjunction code, plus (number of
             * conjunctions - 1)*(size of short-circuit code), and the latter
             * is 2. */
            int offset = totalSize + (conjunctions.size() - 1)*2;
            InstructionList ret = new InstructionList();
            for (InstructionList e : conjunctions) {
                if (rest) {
                    ret.add(new VmInteger(offset));
                    ret.add(InstructionFactory.get(Instruction.Opcode.CNPJUMP));
                }
                ret.add(e);
                rest = true;
                /* New offset is diminished by length of inserted code, which is
                 * length of conjunction + length of short-circuit code. */
                offset -= e.size() + 2;
            }
            return ret;
    }
}

void ConditionAndOperator() : {}
{
    <AND>
}

InstructionList ConditionFactor() : {
     InstructionList ret = null;
     InstructionList expr = null;
     InstructionList arrayExpr = null;
     StackElement op;
     Token str;
     Token header;
```

```
    Token counterName;
    Token hashName;
    Token timerName;
    Token timerOp;
    Token sOrD;
}
{
    ( <NOT> expr = ConditionExpression() {
        ret = new InstructionList();
        ret.add(expr);
            ret.add(InstructionFactory.get(Instruction.Opcode.LNOT));
    } | "(" expr = ConditionExpression() ")" {
        ret = expr;
    } | ( sOrD = <SOURCE> | sOrD = <DESTINATION> ) op = StringOperator() expr =
        StringExpression() {
        ret = new InstructionList();
        ret.add(InstructionFactory.get(sOrD.kind == SOURCE
           ? Instruction.Opcode.SOURCE
            : Instruction.Opcode.DESTINATION));
        ret.add(expr);
        ret.add(op);
    } | <HEADER> header = <STRING> op = StringOperator() expr = StringExpression() {
        ret = new InstructionList();
        ret.add(new VmString(stripQuotes(header.image)));
        ret.add(InstructionFactory.get(Instruction.Opcode.GETHEADER));
        ret.add(expr);
        ret.add(op);
    } | LOOKAHEAD(2) counterName=<ID> op = NumericComparisonOperator() expr =
        ArithmeticExpression() {
        Symbol s = symbolTable.getSymbol(counterName.image);
        if (s == null) {
            throw new ParseException("Line␣" + counterName.beginLine
                + ":" + counterName.beginColumn
                + ":␣Identifier␣\"" + counterName.image
                + "\"␣undeclared");
        } else if (! (s instanceof Counter)) {
            throw new ParseException("Line␣" + counterName.beginLine
                + ":" + counterName.beginColumn
                + ":␣Identifier␣\"" + counterName.image
                + "\"␣not␣a␣counter");
        }
        ret = new InstructionList();
        ret.add(new VmString(counterName.image));
        ret.add(InstructionFactory.get(Instruction.Opcode.GETCOUNTER));
        ret.add(expr);
        ret.add(op);
    } | LOOKAHEAD(2) hashName=<ID> "[" arrayExpr = StringExpression() "]" op =
        NumericComparisonOperator() expr = ArithmeticExpression() {
        Symbol s = symbolTable.getSymbol(hashName.image);
        if (s == null) {
            throw new ParseException("Line␣" + hashName.beginLine
                + ":" + hashName.beginColumn
                + ":␣Identifier␣\"" + hashName.image
                + "\"␣undeclared");
        } else if (! (s instanceof Hash)) {
```

```
            throw new ParseException("Line " + hashName.beginLine
                + ":" + hashName.beginColumn
                + ": Identifier \"" + hashName.image
                + "\" not a counter");
        }
        ret = new InstructionList();
        ret.add(new VmString(hashName.image));
        ret.add(arrayExpr);
        ret.add(InstructionFactory.get(Instruction.Opcode.GETHASH));
        ret.add(expr);
        ret.add(op);
    } | timerName=<ID> "." ( timerOp = <ISARMED>
                           | timerOp = <ISDISARMED>
                           | timerOp = <ISFIRED>
                           | timerOp = <ISUNFIRED>) {
        Symbol s = symbolTable.getSymbol(timerName.image);
        if (s == null) {
            throw new ParseException("Line " + timerName.beginLine
                + ":" + timerName.beginColumn
                + ": Identifier \"" + timerName.image
                + "\" undeclared");
        } else if (! (s instanceof Hash)) {
            throw new ParseException("Line " + timerName.beginLine
                + ":" + timerName.beginColumn
                + ": Identifier \"" + timerName.image
                + "\" not a counter");
        }
        ret = new InstructionList();
        ret.add(new VmString(timerName.image));
        switch (timerOp.kind) {
        case ISARMED: ret.add(InstructionFactory.get(Instruction.Opcode.ISTIMERARMED)); break;
        case ISDISARMED: ret.add(InstructionFactory.get(Instruction.Opcode.ISTIMERDISARMED));
            break;
        case ISFIRED: ret.add(InstructionFactory.get(Instruction.Opcode.ISTIMERFIRED)); break;
        case ISUNFIRED: ret.add(InstructionFactory.get(Instruction.Opcode.ISTIMERUNFIRED));
            break;
        }
    }
    ) {
      return ret;
    }
}


StackElement NumericComparisonOperator() : {
    StackElement ret;
}
{
    ( <EQ> { ret = InstructionFactory.get(Instruction.Opcode.EQ); }
    | <NE> { ret = InstructionFactory.get(Instruction.Opcode.NE); }
    | <GT> { ret = InstructionFactory.get(Instruction.Opcode.GT); }
    | <GE> { ret = InstructionFactory.get(Instruction.Opcode.GE); }
    | <LT> { ret = InstructionFactory.get(Instruction.Opcode.LT); }
    | <LE> { ret = InstructionFactory.get(Instruction.Opcode.LE); }
    )
```

```
    {
        return ret;
    }
}


StackElement StringOperator() : {
    StackElement ret;
}
{
    ( <EQUALS> { ret = InstructionFactory.get(Instruction.Opcode.STRINGEQUALS); }
    | <STARTSWITH> { ret = InstructionFactory.get(Instruction.Opcode.STARTSWITH); }
    | <CONTAINS> { ret = InstructionFactory.get(Instruction.Opcode.CONTAINS); }
    ) {
        return ret;
    }
}

InstructionList ActionPart() : {
    InstructionList ret;
}
{
    <ACTION> "{" ret = Actions() "}"
    {
        return ret;
    }
}

InstructionList Actions() : {
    InstructionList actions = new InstructionList();
    InstructionList action = null;
    InstructionList expr = null;
    InstructionList conditionCode = null;
    InstructionList headerExpr;
    Token delay;
    boolean verdictAction = false;
}
{
    ( ( <ALLOW> {
        if (inRule) {
           verdictAction = true;
               action = new InstructionList();
               action.add(InstructionFactory.get(Instruction.Opcode.ALLOW));
            } else {
               throw new ParseException("Obligations can't have \"allow\" parts");
            }
        } | <BLOCK> {
        if (inRule) {
           verdictAction = true;
               action = new InstructionList();
               action.add(InstructionFactory.get(Instruction.Opcode.BLOCK));
        } else {
           throw new ParseException("Obligations can't have \"block\" parts");
        }
        } | LOOKAHEAD(2) <DELAY> delay = <NUMBER> {
```

```
        if (inRule) {
           verdictAction = true;
               action = new InstructionList();
               action.add(new VmInteger(Long.parseLong(delay.image)));
               action.add(InstructionFactory.get(Instruction.Opcode.DELAY));
        } else {
           throw new ParseException("Obligations can't have \"delay\" parts");
        }
        } | <DELAY> <UNTIL> expr = ConditionExpression() {
        if (inRule) {
           verdictAction = true;
           action = new InstructionList();
           action.add(new VmInteger(conditions.size()));
           action.add(InstructionFactory.get(Instruction.Opcode.DELAYUNTIL));

           conditionCode = new InstructionList();
           conditionCode.add(expr);
           conditionCode.add(InstructionFactory.get(Instruction.Opcode.STOP));
           conditions.add(conditionCode);
        } else {
           throw new ParseException("Obligations can't have \"delay until\" parts");
        }
        } | <MODIFY> <HEADER> headerExpr = StringExpression() <GETS> expr = StringExpression()
              {
        if (inRule) {
           verdictAction = true;
               action = new InstructionList();
               action.add(expr);
           action.add(headerExpr);
           action.add(InstructionFactory.get(Instruction.Opcode.SETHEADER));
        } else {
           throw new ParseException("Obligations can't have \"modify\" parts");
        }
        }
        ) ";" { actions.add(action); } )?
    ( action = Action() ";" { actions.add(action); } )*
    {
        if (inRule && !verdictAction) {
           throw new ParseException("Rules must have either \"allow\", \"block\", \"delay\",
               \"delay until\", or \"modify\"");
        }
        // On obligation, continue executing; on rule, stop.
        if (inRule) {
            actions.add(InstructionFactory.get(Instruction.Opcode.STOP));
        }
        return actions;
    }
}

InstructionList StringExpression() : {
    InstructionList ret = new InstructionList();
    StackElement op;
    InstructionList element;
}
{
```

```
    element = StringTerm() {
        ret.add(element);
    } (
        LOOKAHEAD(1) op = StringAdditionOperator() element = StringTerm() {
        ret.add(element);
        ret.add(op);
    }
    )*
    {
        return ret;
    }
}

StackElement StringAdditionOperator() : {
    StackElement ret;
}
{
    ( <PLUS> {
        ret = InstructionFactory.get(Instruction.Opcode.STRCAT);
    } ) {
        return ret;
    }
}

InstructionList StringTerm() : {
    InstructionList ret = new InstructionList();
    InstructionList expr;
    Token str;
    Token from;
    Token to;
    Token name;
}
{
    ( str = <STRING> {
        ret.add(new VmString(stripQuotes(str.image)));
    } | <HEADER> expr = StringExpression() {
        ret.add(expr);
        ret.add(InstructionFactory.get(Instruction.Opcode.GETHEADER));
    } | <SUBSTRING> "(" expr = StringExpression() "," from = <NUMBER> "," to = <NUMBER> ")" {
        ret.add(new VmInteger(Integer.parseInt(to.image)));
        ret.add(new VmInteger(Integer.parseInt(from.image)));
        ret.add(expr);
        ret.add(InstructionFactory.get(Instruction.Opcode.SUBSTRING));
    } | name = <ID> "[" expr = StringExpression() "]" {
        ret.add(new VmString(name.image));
        ret.add(expr);
        ret.add(InstructionFactory.get(Instruction.Opcode.GETHASH));
    }
    )
    {
        return ret;
    }
}

InstructionList Action() : {
```

```
    InstructionList ret = new InstructionList();
    InstructionList expr;
    InstructionList arrayExpr = null;
    Token t;
    Token counterOp;
    long ns = -1;
    Token times = null;
    Token number = null;
}
{
    ( <DUPLICATE> t = <STRING> {
        ret.add(new VmString(stripQuotes(t.image)));
        ret.add(InstructionFactory.get(Instruction.Opcode.DUPLICATE));
    } | <UPDATECOUNTER> t=<ID> ("[" arrayExpr = StringExpression() "]")? (counterOp = <GETS> |
         counterOp = <INC> ) expr = ArithmeticExpression() {
        Symbol s = symbolTable.getSymbol(t.image);
        if (s == null) {
            throw new ParseException("Line " + t.beginLine
                + ":" + t.beginColumn
                + ": Identifier \"" + t.image
                + "\" undeclared");
        }
        if (arrayExpr == null) {
            if (!(s instanceof Counter)) {
                throw new ParseException("Line " + t.beginLine
                    + ":" + t.beginColumn
                    + ": Identifier \"" + t.image
                    + "\" not a counter");
            }
            ret.add(expr);
            ret.add(new VmString(t.image));
            if (counterOp.kind == GETS) {
                ret.add(InstructionFactory.get(Instruction.Opcode.SETCOUNTER));
            } else {
                ret.add(InstructionFactory.get(Instruction.Opcode.INCCOUNTER));
            }
        } else {
            if (!(s instanceof Hash)) {
                throw new ParseException("Line " + t.beginLine
                    + ":" + t.beginColumn
                    + ": Identifier \"" + t.image
                    + "\" not a hash");
            }
            ret.add(expr);
            ret.add(arrayExpr);
            ret.add(new VmString(t.image));
            if (counterOp.kind == GETS) {
                ret.add(InstructionFactory.get(Instruction.Opcode.SETHASH));
            } else {
                ret.add(InstructionFactory.get(Instruction.Opcode.INCHASH));
            }
        }
    } | <CLEAR> t = <ID> {
        ret.add(new VmString(t.image));
        ret.add(InstructionFactory.get(Instruction.Opcode.CLEARHASH));
```

```
    } | <ARM> t = <ID> ( <FIRE> ns = RelativeTime() )? {
        ret.add(new VmInteger(ns));
        ret.add(InstructionFactory.get(Instruction.Opcode.ARMTIMER));
    } | <DISARM> t = <ID> {
        ret.add(new VmString(t.image));
        ret.add(InstructionFactory.get(Instruction.Opcode.DISARMTIMER));
    }
    )
    {
        return ret;
    }
}

long RelativeTime() : {
  Token number = null;
  long ns = -1;
  long ret;
}
{
  ( <AFTER> number = <NUMBER> ns = Nanoseconds() {
    ret = Long.parseLong(number.image) * ns;
  } | <NEXT> <MONTH> {
    GregorianCalendar gc = new GregorianCalendar();
    gc.set(Calendar.DAY_OF_MONTH, 0);
    gc.set(Calendar.HOUR_OF_DAY, 0);
    gc.set(Calendar.MINUTE, 0);
    gc.set(Calendar.SECOND, 0);
    gc.set(Calendar.MILLISECOND, 0);

    gc.add(Calendar.MONTH, 1);

    ret = (gc.getTimeInMillis() - System.currentTimeMillis())*1000000;
  }
  )
  {
    return ret;
  }
}
/** Returns the number of microseconds in a given time unit. */
long Nanoseconds() : {
    int ret;
}
{
    ( <SECONDS> { ret = 1000000000; } /* 1e9 nsec in 1 sec */
    | <MILLISECONDS> { ret = 1000000; } /* 1e6 nsec in 1 msec */
    | <MICROSECONDS> { ret = 1000; } /* 1e3 nsec in 1 usec */
    )
    {
        if (ret <= 0) {
            ret = 1000000000;
        }
        return ret;
    }
}
```

```
InstructionList ArithmeticExpression() : {
    InstructionList ret = new InstructionList();
    StackElement op;
    InstructionList element;
}
{
    element = ArithmeticTerm() {
        ret.add(element);
    } (
        LOOKAHEAD(1) op = AdditionOperator() element = ArithmeticTerm() {
        ret.add(element);
        ret.add(op);
    }
    )*
    {
        return ret;
    }
}

StackElement AdditionOperator() : {
    StackElement ret;
}
{
    ( <PLUS>  { ret = InstructionFactory.get(Instruction.Opcode.ADD); }
    | <MINUS> { ret = InstructionFactory.get(Instruction.Opcode.SUB); }
    )
    {
        return ret;
    }
}

InstructionList ArithmeticTerm() : {
    InstructionList ret = new InstructionList();
    StackElement op;
    InstructionList element;
}
{
    element = ArithmeticFactor() {
        ret.add(element);
    } (
        LOOKAHEAD(1) op = MultiplicationOperator() element = ArithmeticFactor() {
        ret.add(element);
        ret.add(op);
    }
    )*
    {
        return ret;
    }
}

StackElement MultiplicationOperator() : {
    StackElement ret;
}
{
    ( <MUL>  { ret = InstructionFactory.get(Instruction.Opcode.MUL); }
```

```
    | <DIV> { ret = InstructionFactory.get(Instruction.Opcode.DIV); }
    | <REM> { ret = InstructionFactory.get(Instruction.Opcode.REM); }
    )
    {
        return ret;
    }
}

InstructionList ArithmeticFactor() : {
    InstructionList ret = new InstructionList();
    Token num;
    Token counter;
    Token str;
    Token hash;
    InstructionList index;
}
{
    ( num = <NUMBER> {
        ret.add(new VmInteger(Long.parseLong(num.image)));
    } | <HEADER> str = <STRING> {
        ret.add(new VmString(stripQuotes(str.image)));
        ret.add(InstructionFactory.get(Instruction.Opcode.GETHEADER));
        ret.add(InstructionFactory.get(Instruction.Opcode.INT));
    } | LOOKAHEAD(2) hash = <ID> "[" index = StringExpression() "]" {
        ret.add(index);
        ret.add(new VmString(hash.image));
        ret.add(InstructionFactory.get(Instruction.Opcode.GETHASH));
    } | counter = <ID> {
        Symbol s = symbolTable.getSymbol(counter.image);
        if (s == null) {
            throw new ParseException("Line " + counter.beginLine
                + ":" + counter.beginColumn
                + ": Identifier \"" + counter.image
                + "\" undeclared");
        }
        if (!(s instanceof Counter)) {
            throw new ParseException("Line " + counter.beginLine
                + ":" + counter.beginColumn
                + ": Identifier \"" + counter.image
                + "\" not a counter");
        }
        ret.add(new VmString(counter.image));
        ret.add(InstructionFactory.get(Instruction.Opcode.GETCOUNTER));
    } | "(" ret = ArithmeticExpression() ")"
    )
    {
        return ret;
    }
}
```

Listing B.1: The xESB language syntax as input to the JavaCC parser generator.