Nº D'ORDRE 124-2014 ANNÉE 2014

THÈSE

APPROCHES POUR LA GESTION DE CONFIGURATIONS DE SÉCURITÉ DANS LES SYSTÈMES D'INFORMATION DISTRIBUÉS

Présentée devant :
L'Université Claude Bernard Lyon 1

Pour obtenir :
Le grade de Docteur

Spécialité :
Informatique

Formation doctorale :
Informatique

École doctorale :
Informatique et Mathématiques (InfoMaths)

Par :
Matteo Maria CASALINO

SOUTENUE PUBLIQUEMENT LE 2 JUILLET 2014 DEVANT LE JURY COMPOSÉ DE :

Mohand-Saïd HACID, Professeur des Universités, Université Claude Bernard Lyon 1 Co-directeur de thèse
Romuald THION, Maître de Conférences, Université Claude Bernard Lyon 1 ..... Co-directeur de thèse
Henrik PLATE, Chercheur, SAP Labs France ..... Co-encadrant
Frédéric CUPPENS, Professeur des Universités, ENST Bretagne ..... Rapporteur
Ernesto DAMIANI, Professeur, Université de Milan – Italie ..... Rapporteur
Salima BENBERNOU, Professeur des Universités, Université Paris V ..... Examinateur
Luc BOUGANIM, Directeur de Recherche, INRIA Paris-Rocquencourt ..... Examinateur

LABORATOIRE D'INFORMATIQUE EN IMAGE ET SYSTÈMES D'INFORMATION

SERIAL NUMBER 124-2014 YEAR 2014

THESIS

TECHNIQUES FOR SECURITY CONFIGURATION MANAGEMENT IN DISTRIBUTED INFORMATION SYSTEMS

Presented to:
Université Claude Bernard Lyon 1

In order to obtain:
The degree of Doctor of Philosophy

Domain:
Computer science

Doctoral studies:
Computer science

Doctoral school:
Informatique et Mathématiques (InfoMaths)

Candidate:
Matteo Maria CASALINO

DATE OF THE DEFENCE 2ND OF JULY 2014. COMMITTEE IN CHARGE:

Mohand-Saïd HACID, Professor, Université Claude Bernard Lyon 1 ..... Co-advisor
Romuald THION, Associate Professor, Université Claude Bernard Lyon 1 ..... Co-advisor
Henrik PLATE, Senior Researcher, SAP Labs France ..... Co-supervisor
Frédéric CUPPENS, Professor, ENST Bretagne ..... Reviewer
Ernesto DAMIANI, Professor, University of Milan – Italy ..... Reviewer
Salima BENBERNOU, Professor, Université Paris V ..... Examiner
Luc BOUGANIM, Research Director, INRIA Paris-Rocquencourt ..... Examiner

LABORATOIRE D'INFORMATIQUE EN IMAGE ET SYSTÈMES D'INFORMATION

Preface

Acknowledgments

I owe my sincere gratitude to Mohand-Saïd Hacid, Romuald Thion and Henrik Plate for the kindness, the support and the guidance, without which the work presented in this thesis would not have been possible. It has been a great pleasure working with you over the past three and half years, during which I had the chance to learn a lot and to truly enjoy doing research. I would also like to thank Prof. Stefano Paraboschi, Dr. Aldo Basile, for the support and encouragement, and Prof. Antonio Lioy, who first persuaded me into starting a Ph.D. I am grateful to all the members of the jury: Prof. Frédéric Cuppens, Prof. Ernesto Damiani, Prof. Salima Benbernou and Dr. Luc Bouganim, for agreeing to be reviewers and examiners. In particular, I would like to thank the two reviewers for their insightful comments.

A special thanks to the colleagues and friends at SAP for the collaboration, the discussions and all the fun moments. In particular the PoSecCo team: Henrik, Serena, Wihem, Theodoor; the fellow Ph.D. students: Giancarlo, Corentin, Gabriel, Samuel, Jörn, Ahmad, Mehdi; and all the following people: Sylvine, Jean-Cristophe, Michele, Volkmar, Slim, Luca, Antonino, Francesco, Anderson, Jakub, Stuart, Anne, Antonella, Akram, Stephane, Amine, Johann. Thanks to Florian, Xavier and all the friends of the *big coloc* for the memorable times spent together on the French riviera.

Last, but not least, I would like to thank my parents Franco and Laura, my girlfriend Alice and all my closest friends for the love, the constant encouragement and for sharing the joy of the happy moments as well as enduring my bad mood during the tough ones.

Résumé

La sécurité des services informatiques d'aujourd'hui dépend significativement de la bonne configuration des systèmes qui sont de plus en plus distribués. Au même temps, la gestion des configurations de sécurité est encore fortement basée sur des activités humaines, qui sont coûteuses et sujettes à erreurs. Au cours de la dernière décennie, il a été reporté à plusieurs reprises qu'une partie significative des incidents de sécurité et des pertes de données a été causée par des configurations incorrectes des systèmes.

Pour résoudre ce problème, plusieurs techniques ont été proposées pour automatiser les tâches de gestion des configurations. Beaucoup d'entre elles mettent l'accent sur les phases de planification et de mise en œuvre, où les exigences et les politiques de sécurité abstraites sont conçues, harmonisées et transformées dans des configurations concrètes. Ces techniques nécessitent souvent d'opérer sur des politiques formelles ou très structurées qui se prêtent à un raisonnement automatisé, mais qui sont rarement disponibles dans la pratique. Cependant, moins d'attention a été consacrée aux phases de gestion de suivi et de changement des configurations, qui complètent les étapes précédentes en détectant et en corrigeant les erreurs afin d'assurer que les changements de configuration n'exposent pas le système à des menaces de sécurité.

Les objectifs et les contributions de cette thèse se concentrent sur ce deuxième point de vue, de façon pragmatique sur la base des configurations de sécurité concrètes. En particulier, nous proposons trois contributions visant à analyser et à vérifier des configurations de sécurité :

1. Nous nous concentrons d'abord sur la validation syntaxique des configurations de sécurité, c'est-à-dire, la vérification de l'état d'un système basé sur l'exécution de vérifications syntactiques appelées *checks*. Les approches existantes fixent implicitement la portée des *checks* à une seule machine ou système d'exploitation et elles ne séparent pas clairement l'expression des *checks* de la description des systèmes cibles. Par conséquent, ces techniques ne sont pas appropriées à la détection des problèmes qui sont dûs à la mauvaise configuration simultanée de plusieurs composants d'un système distribué. Notre première contribution étend les techniques de validation de configuration existantes, pour les rendre applicables aux systèmes d'information distribués et les intégrer avec les normes et les pratiques actuelles de l'industrie. Plus particulièrement, nous étendons le standard *Open Vulnerability and Assessment Language* (OVAL) afin de séparer clairement la spécification des *checks* de l'identification des composants à vérifier et des mécanismes de collecte des configurations. Nous décrivons une implémentation preuve de concept d'un interpréteur du langage étendu. Nous discutons de l'intégration de ce prototype dans différents scénarios qui diffèrent selon l'origine des *checks* à effectuer, leurs objectifs et les modalités de leur exécution.

2. Quand l'écart entre la syntaxe et la sémantique d'un langage de configuration de sécurité augmente, les contrôles syntaxiques devient de moins en moins aptes à exprimer des conditions ou des invariants intéressants, ainsi il devient difficile de prévoir l'impact sur la sécurité résultant d'une différence entre deux configurations. L'étude de cette question est notre deuxième contribution. En particulier, nous considérons le problème de l'évaluation de l'impact du changement des configurations de contrôle d'accès dans les applications Web par rapport à leur permissivité, qui malgré l'omniprésence des technologies Web, n'a pas encore été explicitement abordé. Nous proposons une sémantique dénotationnelle du langage de contrôle d'accès des applications web JEE (Java Enterprise Edition), à partir de laquelle nous définissons une procédure pour comparer deux configurations vis-à-vis de leur permissivité. Nous implémentons et évaluons notre modèle en comparant la sémantique formelle que nous proposons à l'implémentation actuelle des conteneurs JEE existants. La batterie de tests automatisés pour réaliser cette évaluation est explicitée. D'une part, nous avons remarqué que notre interprétation formelle modélise les implémentations réelles de façon satisfaisante, et d'autre part, nous avons pu identifier une erreur d'implémentation du contrôle d'accès dans le serveur Apache Tomcat JEE jusqu'ici inconnue.
3. Raisonner sur la sémantique de la configuration d'un seul système de contrôle d'accès n'est pas assez dans le cas des systèmes distribués, où le changement de la configuration d'un composant peut affecter le comportement des autres. Bien que cette problématique ait été largement étudiée dans les domaines de la composition des politiques et de la détection de conflits dans les couches applicatives ou les couches réseaux séparément, le traitement des interactions inter-couches est toujours considéré comme un problème ouvert. Ainsi, notre dernière contribution porte sur la gestion du changement des configurations à des niveaux différents. Nous proposons une technique pour réorganiser (*refactoring*) des politiques inter-couches, c'est-à-dire de réécrire une collection de politiques de contrôle d'accès appartenant à des niveaux architecturaux différents de sorte que : (i) la permissivité de la politique globale est préservée, (ii) le principe du moindre privilège est garanti, (iii) les interactions inter-couches inutiles sont supprimées. À cet effet, nous proposons un modèle générique de contrôle d'accès qui prend en compte les interactions entre les autorisations exprimées à des niveaux différents. Sur ce modèle, nous définissons la composition de politiques de contrôle d'accès et nous montrons que son inverse, la décomposition, fournit, quand elle existe, une solution au problème de la réorganisation. Enfin, nous proposons des algorithmes pour tester si des politiques sont effectivement décomposables, et le cas échéant, calculer la décomposition. Notre principal résultat théorique est une caractérisation des conditions qui garantissent qu'une telle décomposition est possible. Cette contribution s'appuie sur des résultats issus de la théorie des bases de données relationnelles et les étends à nos besoins, suggérant ainsi l'intérêt des ré-

sultats de ce domaine pour la résolution de problèmes concernant les politiques de sécurité. Nous évaluons la faisabilité de notre approche en conduisant une évaluation expérimentale des algorithmes sur des politiques de contrôle d'accès synthétiques dont nous faisons varier plusieurs paramètres. Les résultats expérimentaux donnent des performances comparables à celles obtenues par d'autres algorithmes pour l'analyse statique de configurations de sécurité proposés dans la literature.

Cette thèse a été financée avec SAP AG sous la convention CIFRE no. 154/2011 (`http://www.anrt.asso.fr/fr/espace_cifre/accueil.jsp`) et par le projet européen FP7-ICT-2009-5 no. 257129 : "PoSecCo : Policy and Security Configuration Management" (`http://www.posecco.eu`).

Abstract

THE security of nowadays IT services significantly depends on the correct configuration of increasingly distributed information systems. At the same time, the management of security configurations is still heavily centered on human activities, which are costly and prone to error. Over the last decade it has been repeatedly reported that a significant share of security incidents and data breaches are caused by inaccurate systems configuration.

To tackle this problem, several techniques have been proposed to increase the automation in configuration management tasks. Many of them focus on planning and implementation, i.e., the phases where abstract security requirements and policies are elicited, harmonized, de-conflicted and transformed into concrete configurations. As such, these techniques often require formal or highly structured input policies amenable to automated reasoning, which are rarely available in practice. In contrast, less attention has been dedicated to the monitoring and change management phases, which complement the above steps by detecting and remediating configuration errors and by ensuring that configuration changes do not expose the system to security threats.

The objectives and contributions of this thesis take the latter perspective and, as such, they pragmatically work on the basis of concrete security configurations. In particular, we propose three contributions that move from more concrete syntax-based configuration analysis towards increasingly abstract semantic reasoning.

1. We first focus on configuration validation, i.e., the evaluation of the security state of a system based on the execution of syntactic configuration checks. Existing approaches often implicitly fix the checks' scope to a single machine or operating system and they do not clearly separate the description of the check logic from that of target systems. Hence, such techniques are not suitable for detecting issues that are due to the simultaneous misconfiguration of distributed system components. Our first contribution extends standard-based syntactic configuration validation techniques to make them applicable to distributed information systems and to integrate them with current industry standards and practices. Specifically, we extend the Open Vulnerability and Assessment Language (OVAL) to introduce a clear separation between the specification of check logic, the identification of targets and the mechanisms for collecting to-be-checked configurations. We describe a proof-of-concept implementation of both the language and its interpreter. We discuss their integration in several scenarios that differ in terms of purpose and authorship of configuration checks and modality of invocation of the configuration validation process.
2. As the gap between syntax and security semantics of a configuration language increases, syntactic checks become less suitable to express interesting conditions

or invariants, because different syntactic discrepancies may have more or less relevant security impact. Studying this issue is our second objective. In particular, we consider the problem of evaluating the change impact of access control configurations of web applications with respect to their permissiveness, which, despite the pervasiveness of web technologies, has not been explicitly tackled so far. We provide a denotational semantics for the access control configuration language of JEE (Java Enterprise Edition) web applications, on top of which we define a procedure to compare access control configurations with respect to their permissiveness. We implement our model and evaluate it with respect to the operational semantics of existing JEE container implementations through automated software testing. The findings include not only positive results supporting the correctness of our semantics, but also evidence of discrepancies that led to the discovery of a previously unknown implementation error in the Apache Tomcat JEE container.

3. Reasoning on the semantics of the configuration of a single access control system is not enough in the case of distributed systems, whereby different components' configurations may affect each other's behaviour. While this issue has been largely investigated in the domains of either network or application layer policy composition and conflict detection, the treatment of inter-layer interactions is still considered an open problem. Thus, our last objective focuses on the change management of configurations specified on different architectural layers. We propose a technique to perform multi-layered policy refactoring, i.e., to rewrite a collection of access control policies belonging to different architectural layers such that: (i) the global permissiveness is preserved, (ii) the least privilege principle is enforced and (iii) superfluous inter-layer interactions are removed. To this end, we embed a generic access control system into a structure that keeps track of the interactions among authorization decisions taken on different layers. We then define the semantics of composition of such access control layers and show that its inverse, namely decomposition, provides (when it exists) a solution to the problem of refactoring. Finally, we provide algorithms to test for decomposability, as well as to compute (de)composition. Our main theoretical result is the proof of correctness of the decomposability condition for access control layers, which leverages and extends existing results in database dependency theory, and provides novel evidence that the study of database dependencies can be fruitfully applied to help solve security problems. To assess the feasibility of our approach, we evaluate the algorithms with respect to various properties of input policies. The results show comparable performances with previous work on the static analysis of network security configurations.

This thesis has been funded by SAP AG under the CIFRE convention no. 154/2011 (`http://www.anrt.asso.fr/fr/espace_cifre/accueil.jsp`), and by the EU funded project FP7-ICT-2009-5 no. 257129: "PoSecCo: Policy and Security Configuration Management" (`http://www.posecco.eu`).

List of Publications

The work presented in this thesis has been published in one book chapter and in the proceedings of national as well as international peer-reviewed conferences and workshops. The list of publications is the following:

[Basile2013] Cataldo Basile, Matteo Maria Casalino, Simone Mutti & Stefano Paraboschi. In: John Vacca. *Computer and Information Security Handbook*. Edited by John Vacca. 2nd. Morgan Kaufmann, June 2013. Chapter Detection of conflicts in security policies.

[Casalino2012a] Matteo Maria Casalino, Michele Mangili, Henrik Plate & Serena Elisa Ponta. "Detection of Configuration Vulnerabilities in Distributed (Web) Environments". In: *SecureComm*. Edited by Angelos D. Keromytis & Roberto Di Pietro. Volume 106. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Acceptance rate of 28%. Springer, 2012, pages 131–148.

[Casalino2012b] Matteo Maria Casalino, Henrik Plate & Serena Elisa Ponta. "Configuration Assessment as a Service". In: *DPM/SETOP*. Edited by Roberto Di Pietro, Javier Herranz, Ernesto Damiani & Radu State. Volume 7731. Lecture Notes in Computer Science. Springer, 2012, pages 217–226.

[Casalino2012c] Matteo Maria Casalino, Romuald Thion & Mohand-Said Hacid. "Access Control Configuration for J2EE Web Applications: A Formal Perspective". In: *TrustBus*. Edited by Simone Fischer-Hübner, Sokratis K. Katsikas & Gerald Quirchmayr. Volume 7449. Lecture Notes in Computer Science. Springer, 2012, pages 30–35.

[Casalino2013a] Matteo Maria Casalino & Romuald Thion. "Extending Multivalued Dependencies for Access Control Policy Refactoring". In: *29eme journées Bases de Données Avancées (BDA)*. Conference without formal proceedings. Oct. 2013. URL: `http://liris.cnrs.fr/publis/?id=5601` (visited on 07/01/2014).

[Casalino2013b] Matteo Maria Casalino & Romuald Thion. "Refactoring Multi-Layered Access Control Policies Through (De)Composition". In: *Proceedings of the 9th international conference on Network and Service Management, CNSM'13, Zurich, Switzerland*. Acceptance rate of 18%. IEEE Computer Society, Oct. 2013, pages 243–250. URL: `http://www.cnsm-conf.org/2013/`

documents/papers/CNSM/p243-casalino.pdf (visited on 07/01/2014).

Furthermore, the thesis' contributions are part of the EU funded project "PoSecCo: Policy and Security Configuration Management". They appear in the following public project deliverables:

[Basile2013] Cataldo Basile, Marco Vallini & Matteo Casalino. *D3.7 – ENFORCEABILITY ANALYSIS*. Deliverable. FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu), Jan. 2013. URL: http://posecco.eu/fileadmin/POSECCO/user_upload/deliverables/D3.7_EnforceabilityAnalysis.pdf (visited on 02/06/2014).

[Bettan2012] Olivier Bettan, Serena Ponta, Kreshnik Musaraj & Matteo Casalino. *D4.8 – PROTOTYPE: STANDARDIZED AUDIT INTERFACE*. Deliverable. FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu), Oct. 2012. URL: http://posecco.eu/fileadmin/POSECCO/user_upload/deliverables/D4.8_Prototype_Standardized_Audit_Interface_v1.0.pdf (visited on 02/06/2014).

[Ponta2012] Serena Ponta, Madonna Mbatshi, Baptiste Cazagou, Lan Xu, Arnaud De la Bretesche, Henrik Plate & Matteo Casalino. *D4.5 – FINAL VERSION OF A CONFIGURATION VALIDATION LANGUAGE*. Deliverable. FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu), Apr. 2012. URL: http://posecco.eu/fileadmin/POSECCO/user_upload/deliverables/D4.5_v1.0.pdf (visited on 02/06/2014).

Contents

List of Figures

List of Tables

The mantra of any good security engineer is: "Security is a not a product, but a process." It's more than designing strong cryptography into a system; it's designing the entire system such that all security measures, including cryptography, work together.

—Bruce Schneier

1 Introduction

▷ *At the beginning of each chapter we propose a summary of its main contributions. The table of content of the chapter is presented on the following page.*

This chapter is the introduction of the thesis. We first introduce the notion of distributed information system. In this context, we describe the principles of security configuration management and we analyze the problems and challenges within this subject. We then motivate and position the objectives and contributions of the thesis with respect to existing approaches and techniques. To illustrate how our contributions integrate in a common security configuration management framework, we introduce the European research project PoSecCo (Policy and Security Configuration Management), that supports the management of policies and configurations from the point of view of a service provider operating a distributed system infrastructure. In conclusion, we outline the structure of the manuscript. ◁

Chapter Outline

MODERN information systems are more and more constituted by assembling modular off-the-shelf components, the behaviour of which has to be customized by the means of proper configuration. From the system security standpoint, while increased modularization and reuse lead, on the one hand, to the easier implementation of smaller and better tested — hence less vulnerable — components, on the other hand they make the deployment, administration and management tasks more challenging and security-critical.

In particular, over the last decade, the management of security configurations has become increasingly complex and prone to human error, which has made security misconfiguration become one of the topmost causes of security incidents and data breaches. Before discussing the reasons that lie behind this problem, we first introduce the notion of distributed information system and we review the basis of nowadays common security configuration management practices.

1.1 Distributed Information Systems

Over the past half century, from the beginning of the modern computer era to nowadays, computer systems underwent an incredibly fast and unprecedented evolution. Two aspects are typically recognized as the main drivers of this process: the exponential[1] increase in the transistors density of integrated circuits on one side, leading to the development of cheaper microprocessors, and the development of increasingly fast digital telecommunication technologies on the other.

One fundamental consequence of the combination of these two factors is that bigger and bigger amounts of individual computers became easily available and could be connected in networks to share information even at large distances. As such, distributed systems rapidly emerged as a more flexible and scalable paradigm in contrast to that of previous centralized systems (or mainframes).

Tanenbaum and Sten provide the following rather generic definition of distributed systems [Tanenbaum2002]:

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

Moreover, as anticipated earlier, a distinguishing characteristic of a distributed system consists of having its independent components connected through a network and a distribution middleware, enabling computers to coordinate their activities and to share the resources of the system.

[1] According to Moore's law.

Different classes of distributed systems can be distinguished, depending on the main goal they are meant to achieve. An important category is, for instance, that of high-performance computing systems, such as computer clusters or grids, that are designed to tackle complex computational problems by distributing subproblems to many independent nodes in parallel. Another noteworthy type of distributed systems is constituted by pervasive systems, such as sensor networks or home devices (e.g., smartphones, tablets, smart TVs, wearable devices, etc...), for which distribution is not a design choice but rather an intrinsic feature.

The category which is of central interest in the scope of this thesis is that of **distributed information systems**. Information systems are collections of hardware and software that allow people and organizations to collect, manage and process data representing information. Well-known applications of such systems include, e.g., enterprise resource planning, office automation, electronic commerce, search engines, decision support, transaction processing, database management, etc. Originally developed as monolithic single-tier systems, they progressively evolved towards multi-tier architectures, where data presentation, processing (i.e., the implementation of the business logic) and persistency became physically separated functionalities deputed to independent specialized components. Even more fine-grained separation was introduced with the widespread of service oriented paradigms, whereby tighter machine-to-machine integration is envisaged even across organizational boundaries and over public networks, such as the Internet.

Compared to high-performance computing systems, distributed information systems are constituted by more heterogeneous components which handle the whole data lifecycle (instead of only the processing part) and need to be **configured consistently** to comply with a collection of **security policies**, which are ultimately concerned with securing protect-worthy information. Moreover, unlike pervasive systems that are typically designed for self-adaptability, they are directly under human administrative control, which, as we will argue in the remainder of this chapter, is costly and prone to error. For these reasons, **distributed information systems constitute the context wherein the techniques proposed in this thesis are meant to apply**.

1.2 Security Configuration Management

Over the years, several common practices and standards have been adopted by organizations and individuals to structure and facilitate the management of IT related risk. One of the best known such practices is the PDCA (Plan-Do-Check-Act) cycle [Moen2010], also known as the Shewhart or Deming cycle, that was made popular by Dr. W. Edwards Deming as a means to constantly improve the quality of generic processes and services. Deming's *plan* and *do* phases correspond respectively to the setting of objectives and their subsequent implementation. The *check* and *act* steps serve

instead to identify changes and deviations with respect to the planned objectives and to react accordingly.

A prominent example application of the PDCA cycle to the management of information security in the IT industry is provided as part of the Information Technology Infrastructure Library (ITIL). ITIL is a set of processes and best practices to guide the management of the full lifecycle of IT services and it is widely accepted as the de facto standard for the management of IT systems. Its latest version, ITIL v3, provides processes and functions covering the full lifecycle of services. The lifecycle of a service comprises the various stages through which the service passes and, in ITIL, it is described from the point of view of the service provider. The service lifecycle consists of 5 stages which are guided by best practices, namely service strategy, design, transition, operation and continual improvement.

Several processes are defined to structure management activities throughout such stages. Of particular interest from the point of view of security is the *information security management* process, which gained increasing attention in the latest version of ITIL. This process is part of the service design phase and its purpose is to provide a focus for all aspects of IT security and manage all IT security activities. As stated in [ITIL2007], it "ensures the confidentiality, integrity and availability of the organization's assets, information, data and IT services" and it is actually a customization of the four PDCA phases for the management of information security [OGC2007]. In particular, as depicted on the left-hand side of Figure 1.1, the *plan* phase is dedicated to the elicitation of security requirements and policies, which typically result from a risk analysis phase and define the organization's attitude on security matters. The *do* phase, here named *implementation*, involves putting in practice all the measures that are necessary to enforce the policies, e.g., configuring network and application security features and establishing appropriate access rights, but also training the employees and preventing unauthorized physical access to the premises. The effectiveness of such measures is constantly monitored in the *evaluate* (corresponding to Deming's *check*) phase, by the means of internal as well as external audits. Any detected potential issue, as well as actual security incident, is analyzed and appropriate countermeasures are taken in the *maintain* (corresponding to Deming's *act*) phase.

To support administrators in the implementation of this process, as well as other related ones, ITIL introduces the concept of Configuration Management System (CMS), which comprises the "set of tools and databases that are used to manage an IT service provider's configuration data". An essential part of the CMS is the Configuration Management Database (CMDB) that stores the information about all manageable system components, such as their attributes and relationships, together with their configuration data [ITIL2007]. Several major software vendors, such as SAP, IBM, and HP, nowadays complement their product portfolio with configuration management tools that offer many of the functionalities of ITIL's CMS and CMDB.

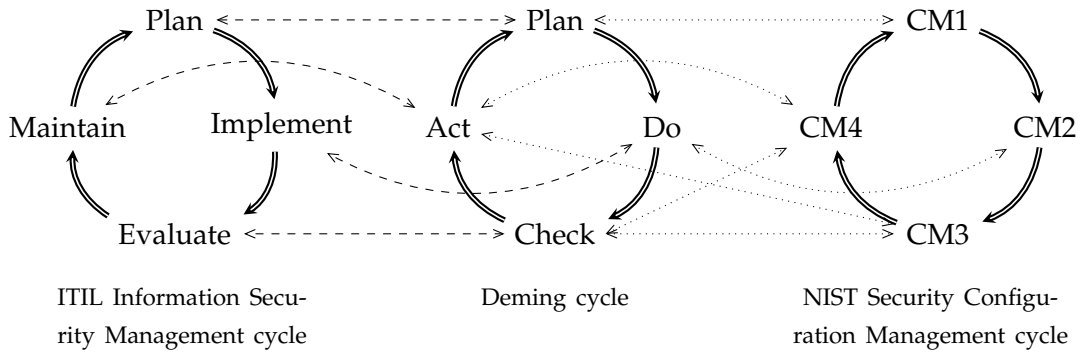


Figure 1.1: Comparison of PDCA cycles: dashed (resp. dotted) lines highlight the correspondences between Deming's and ITIL's (resp. NIST's) phases.

Within ITIL's high-level guidelines and recommendations we observe several references to the concepts of security policies and configurations. Moreover, it becomes clear that the proper management of security configurations is crucial for the effective administration of information security in IT systems. In the remainder of this section we provide a more precise description of these concepts, while illustrating their involvement in a second, more specific, instance of the PDCA cycle, which constitutes the core of the Security Configuration Management (SCM) process.

We rely on the description, given by the U.S. National Institute of Standards and Technology (NIST) in [Johnson2011], which defines SCM as:

> *the management and control of secure configurations for an information system to enable security and facilitate the management of risk.*

In a nutshell, SCM is again structured as a closed-loop cycle composed of the four following phases (cf. right-hand side of Figure 1.1):

CM1) Planning;

CM2) Identifying and implementing configurations;

CM3) Controlling configuration changes;

CM4) Monitoring.

During the planning phase (CM1), the security goals of the information system are identified and expressed in the form of a collection of *security policies*. Several different definitions of the concept of policy have been given in literature. In this context, and throughout all this thesis, a policy denotes:

> *a definite goal, course or method of action to guide and determine present and future decisions. [Westerinen2001]*

In particular, a security policy concerns specifically security goals, such as ensuring the confidentiality and integrity of sensitive data or enforcing proper access control over the network and within the different applications that collaboratively deliver business services. Policies are most commonly expressed in prose language as a sequence of informal mandatory statements.

The purpose of phase (CM2) is twofold. First, a collection of security mechanisms suitable to (cost effectively) enforce the policies resulting from phase (CM1) are identified. Next, *security configurations* specifying the intended behaviour of such mechanisms must be devised accordingly and deployed to the system. More precisely, by the term configuration, we name:

> *the set of parameters in network elements and other systems that determine their function and operation. [Westerinen2001]*

Security configurations are those that specifically affect the security-relevant behaviour of systems.

Configurations must have a machine-readable representation, as they have to be interpreted by a system to adapt its runtime behaviour accordingly. Moreover, as they are meant to be provided by system administrators, they can be most often expressed in a format that is understandable to humans as well, that is, they respect a well-defined syntax which we refer to as *configuration language*. A configuration language can feature simple constructs, such as key-value pairs to configure a set of enumerable settings, or may involve more complex expressions like, e.g., to associate a specific behaviour to all the system's states that match a given pattern, or to configure a complex behaviour as a result of the composition of simpler statements. For instance, the `SSLRequireSSL` directive [ASF2014b] is an example key-value security configuration of the Apache web server that precludes any access to one or more URLs unless a secure channel (HTTPS) is established between client and server. A firewall ruleset is instead a more complex example of network access control configuration, whereby network packets are either allowed or blocked according to the action specified by the first rule that matches to a given set of packets' features (e.g., IP source and destination addresses, transport protocol, TCP/UDP ports, etc.).

The last two phases, (CM3) and (CM4), are necessary to handle, respectively, foreseen and unforeseen changes in the system's configuration. Due to the dynamicity of nowadays information systems, changes are likely to occur for several reasons, e.g., installation or replacement of technical equipment, restoration of broken functionalities, handling of patches and upgrades, etc. In principle, any change in the security configurations could break the compliance with the desired security policies, hence it is essential to carefully plan, test and document changes, which is precisely the purpose of phase (CM3). Crucial to this phase is the ability to anticipate the impact of configuration changes on the global security properties guaranteed by the system.

Even if substantial effort is spent in managing and controlling changes, the risk for unplanned deviations to occur is not negligible in reality. Therefore appropriate configuration monitoring activities should also be put in practice, as prescribed by phase (CM4). In this phase, the actual state of the entire system configuration needs to be periodically compared to the desired state, which was determined in phase (CM2). Moreover, it is important to detect all possible misconfigurations which, although not explicitly violating any policy, still expose the system to unforeseen security vulnerabilities. Finally, even if the system does not expose known vulnerabilities, it is highly recommendable to check whether configuration guidelines and best practices, which are typically released by software vendors and security experts, are correctly followed, in order to harden the system and minimize the risk of incidents. It is worth noting that not every discrepancy between the actual and desired state of the system configuration necessarily constitutes a source of problems. This can become an issue especially in large systems, where handling every single discrepancy alert may rapidly become impractical. It becomes then crucial to assess the severity of misconfigurations that is to determine their potential impact on the overall system security in order to prioritize remediation actions.

Note how phases (CM1) and (CM2) closely relate to Deming's *plan* and *do* ones respectively. Deming's *check* and *act* phases cover instead two aspects that are common to both phases (CM3) and (CM4): namely (i) the detection of either undesired deviations or evolving security needs, and (ii) the implementation of changes for remediation and improvement. Other than tailoring these concepts to the domain of security configuration management, the NIST's definition factorizes them differently, by distinguishing the task of handling planned changes from that of dealing with unplanned ones.

1.3 Problems and Challenges in SCM

Nowadays, configuration management activities still largely rely on human-centric processes, often involving the collaboration of multiple stakeholders with different domains of expertise. Over the last decade, however, researchers and analysts have shown that this practice does not cope adequately with the increasing scale and complexity of modern IT infrastructures. At the same time, numerous data breach reports and surveys revealed that a significant share of security vulnerabilities and, consequently, incidents are due to the improper configuration of existing defense mechanisms.

1.3.1 Managing Distributed Systems' Configurations is Complex and Error-prone

Already in 2003 the results of a study conducted by Oppenheimer et al. [Oppenheimer2003b] on the main causes of failures in three large-scale Internet services

showed that (i) errors committed by human operators are the first cause of service failure and (ii) systems misconfiguration constitutes the largest category of such errors.

A 2002 survey conducted by the Yankee group [Kerravala2004] yields similar conclusions for what concerns network configuration management, that is shown to be affected by human error in the 62% of cases. Furthermore, given the increasing criticality of the services offered over computer networks, the cost of downtime is estimated to grow substantially, motivating the need for more effective configuration management techniques.

Likewise, a more recent paper published in 2008 by Juniper Networks reports that different studies attributed from 50 to 80 percent of network outages to human errors [Juniper2008]. In particular, it is pointed out how such errors are mainly due to the "system complexity with multiple components and many types of interactions", that "creates an environment where the relationship between actions and outcomes is not always obvious".

On the same line [Oppenheimer2003a] argues that operator errors are caused by the poor understanding of the existing configuration, which is hindered by the increasingly distributed nature of nowadays systems. The authors state in fact that, in a distributed system, "[...] due to the possibility of cascading failures, configuration options that control cross-component interactions are more likely to have global effects than are single-component ones".

A further confirmation of the above findings comes from a recent study involving support data of both commercial and open-source software deployed at thousands of customers, reporting that configuration issues cause the largest percentage (31%) of high-severity support requests [Yin2011]. Moreover, the complexity of configuring distributed systems is again remarked as a major issue: "[...] still a significant portion [of misconfigurations] (21.7%~57.3%) involve configurations beyond the system itself or span over multiple hosts".

1.3.2 Misconfiguration is a Major Security Threat

Being configuration management such a complex and challenging task, it is not surprising that the security of IT systems, that tightly depends on the correct configuration of many different hardware and software components, is severely threatened by the risk of misconfiguration. Indeed, as reported by Forrester [Kark2006], organizations often either cannot prove that system configurations correctly enforce their security policies, or "it is prohibitively expensive to do so". This fact has been most recently confirmed by a sample of over 900 IT professionals who, when surveyed in 2012, ranked the task of enforcing security policies as the second most difficult IT security challenge after that of managing the complexity of security [Davis2012].

As a matter of facts, configuration errors have been repeatedly found to be among the causes of data breach incidents and cyber-attacks, the cost of which has been (and still is) steadily increasing [Ponemon2013b; Ponemon2013a]. For instance, a study released by British Telecom and Gartner in 2004 concluded that up to 65% of successful cyber-attacks were directly related to configuration errors [BT2004]. Four years later, the american CSIS (Center for Strategic and International Studies) Commission on Cybersecurity, which was instituted to provide findings and recommendations to secure cyberspace in the 44th United States Presidency, reported that "inappropriate or incorrect security configurations were responsible for 80% of United States Air Force vulnerabilities" [CSIS2008].

Several data breach reports revealed similar findings too. Both in 2009 and 2010 the Verizon Data Breach Investigation Report showed that misconfiguration was the leading category of errors contributing to data compromise and explicitly stated that "contributory error is almost always involved in a breach" [Verizon2009; Verizon2010]. Accordingly, the 2010 UK Security Breach Investigation Report attributed the 30% of analyzed security breaches to inaccurate server or network filtering configuration [7Safe2010].

Server misconfiguration has also been the most prevalent category of security vulnerabilities reported recently by a 3 years long penetration testing study conducted on web applications: "in all three years [2010 to 2012], insecure server configuration and information leakage accounted for the highest number of vulnerabilities identified". Moreover, "the server configuration category is the only category which saw consistent increases each year" [Tudor2013].

As a further confirmation of the severity of the risks stemming from improper security configuration management, security experts and analysts progressively adapted their recommendations and best practices to mention it as an important issue. As of 2010 security misconfiguration appears among the top 10 most critical web application security risks according to OWASP [OWASP2010; OWASP2013]. In 2011, Gartner considered secure configuration management as a must-have rather than a nice-to-have control, ranking it first on the list of server protection priorities [MacDonald2011]. Most recently, in 2013, SANS [SANS2013] lists secure configuration for systems, servers and end points as a third critical control, and secure configuration for network and security devices as a tenth critical control.

1.3.3 Research Challenges

In order to improve the effectiveness of current configuration management practices, it is convenient to identify the dimensions of complexity that characterize this problem. As mentioned earlier, several authors identify in human error the main cause of misconfiguration issues [Oppenheimer2003b; Kerravala2004], but why is this the case?

What makes configuration management tasks prone to error? To answer these questions we decline the problem's complexity into five main challenges, according to the categorization proposed by [AlShaer2011].

Semantic gap. Security configurations are typically expressed according to a respective configuration language with well-defined syntax. The semantics of this language is ultimately given by the behaviour of the configured system at runtime. As a matter of fact, the gap of abstraction which lies between a security configuration and the corresponding enforced policy is not dissimilar, conceptually, to the difference between a program's source code and the behaviour realized by an interpreter while executing it.

Configuration authors need to have a thorough understanding of the interpretation semantics of syntactic constructs, so that they can configure the system behaviour exactly according to the policy they want the system to enforce. Unfortunately, such a semantics is often described in prose language within lengthy documents such as user manuals or technical specifications, which can lead to ambiguities and misinterpretation. Several authors [Ni2009; Ramli2011; Kassab1998; Cuppens2004; Bishop2006] argue that this is dangerous and advocate for the need of providing formal semantics to configuration languages, that, on top of removing ambiguity, enables automated reasoning and verification.

Large scale and heterogeneous. Configuration files can be very large in size (e.g., up to several thousand rules for largest firewall rulesets [Wool2010]), and hence hard to be consumed and fully understood by system administrators. The heterogeneity of the various configuration languages that, in practice, often coexist in the same environment complicates even more this issue.

Furthermore, as argued in [Bellovin2009], "managing the configuration of 100 machines is a different problem than managing one or two; managing 1000 is different still". Here, qualitative rather than merely quantitative difference is meant: in many cases the processes that work for small-scale systems are simply not applicable to large-scale ones. For instance, it is well known that, due to budget constraints, auditors are often forced to resort to sampling techniques, thereby limiting their analysis to a small sample of an organization's assets [Hall2002]. In the context of an IT audit, where the effectiveness and compliance of technical controls (e.g., configuration settings) with respect to the control objectives (e.g., security policies) have to be assessed, this means that potentially dangerous misconfigurations may remain overlooked.

Distributed yet interdependent. IT systems typically rely on the cooperation of a variety of components, lying on different architectural layers, to deliver services. For instance, *network-layer* components such as switches, routers, firewalls, VPN gateways ensure and regulate connectivity; *platform-layer* components like operating systems, application servers, virtual machine hypervisors provide execution envi-

ronments suitable for more or less specific purposes; *application-layer* components implement the actual services: email, database, web, etc. It is widely recognized that such a strong inter-component interaction necessarily influences the security properties of the system as a whole. Therefore security configurations cannot be only considered individually for each component, but need to take into account the context as well.

For example, a firewall configuration must be consistent with that of other firewalls lying upstream or downstream in the network to avoid anomalies such as conflict, redundancy or shadowing [AlShaer2005; Alfaro2008; Basile2012]. Traffic encryption (e.g., via IPSec) does not allow packet inspection, hence intrusion detection systems and layer-7 firewalls must be placed and configured accordingly [Fu2001].

Not only it is necessary to account for interactions among components on the same layer (e.g., network), but inter-layer interactions often play a crucial role too. According to Sloman and Lupu [Sloman2002], the study of the interdependencies among multiple levels of policies constitutes a relevant research topic in policy and configuration management, and entails some interesting open questions and issues. For instance, "an application-specific policy may be more efficiently interpreted within a network component, or an application may need to adapt its behaviour as a result of adaptation within the network".

Dynamic nature. IT systems evolve over time, driven by changing requirements on the one side and evolving technology on the other. As a consequence, configurations need to change accordingly in order to ensure a correct operational behaviour. Ensuring that configuration management processes are able to flexibly cope with such dynamics is a challenging issue.

Furthermore, in many cases configurations have to change and adapt depending on the context. One prominent example is given by the need of dynamic policy enforcement in context-aware access control models [Covington2002; Thomas2004; Wullems2004], whereby authorization is affected, for instance, by spatiotemporal constraints that can be enforced by dynamically reconfiguring the system according to the user's context.

Finally, there are situations where it is necessary to model a system as if the state of its configuration evolved according to rules that are themselves part of the configuration. This is typically the case, for example, of stateful firewalls [Gouda2005], where some rules may (or may not) apply to a given traffic flow depending on whether previous packets triggered other rules in the past. Expressive access control models that support delegation and the assignment of permissions about permissions constitute another example. Several interesting and difficult problems exist in such cases, like that of checking that an adversary could never gain unauthorized access to certain resources [Guelev2004].

Multiple stakeholders. Guaranteeing the consistency among independently-specified

security policies in large distributed systems is a well-known problem that attracted the attention of several researchers in the past years [Moffett1994; Lupu1999; Hamed2006; Satoh2008; Uszok2003; Davy2008a]. Although different authors focus on different kinds of security areas (e.g., authorization, obligation, network filtering, data protection) and on different abstraction layers (e.g., policies vs. configurations), their works share similar motivations: if multiple stakeholders are involved in the authoring of distributed security policies, there exist a substantial risk of introducing conflicts or anomalies that must be detected and resolved.

This problem is even more exacerbated in the context of emerging service delivery paradigms such as those proposed in cloud computing; namely infrastructure, software and platform as a service. In fact, in such scenarios it is common for different stakeholders (e.g., cloud or platform provider, service provider and service consumer) to control and interact with different parts of the same IT infrastructure. Therefore consistent configuration management is required not only within individual organizations, but also across them. Misconfiguration in cloud environments, according to [Behl2012], is "very critical with multi-tenancy, where each tenant has its own security configurations that may conflict with each other leading to security holes".

1.4 Objectives and Contributions

In order to tackle the issues affecting current configuration management practices, a variety of techniques have been proposed by researchers that aim at supporting system and security administrators throughout the different SCM phases.

The activities involved in phases (CM1) and (CM2) mainly concern (i) the elicitation of security requirements (what needs to be protected and why) as well as corresponding policies (how shall the system behave to be secure), and (ii) the configuration of suitable enforcement mechanisms to implement such policies. Substantial effort has been dedicated to structure, formalize and partially automate these tasks. For instance, the requirement engineering community proposed several approaches, nicely surveyed in [Fabian2010; Mellado2010], to integrate non-functional and, more specifically, security requirements into modeling techniques for software and system engineering (mostly based on the UML standard). Many security policy languages have been proposed in literature, as summarized in [Sloman2002; Vimercati2007; Bonatti2009; Han2012]; most of them allow to express authorization and some support more advanced features like obligation or delegation.

Some of the approaches for security requirement and policy specification have formal foundations and therefore are amenable to various kinds of automated reasoning. For instance, the composition of policies [Bonatti2002; Wijesekera2003] — possibly

specified in different languages — and the detection of conflicts within policies [Moffett1994; Lupu1999; Uszok2003; Davy2008a] are useful in phase (CM1), to obtain a consistent and harmonized policy specification. Transformation, or refinement, techniques [Lodderstedt2002; Davy2008b; Craven2010; Preda2010; Zhao2011] support instead especially phase (CM2), by automating the translation of high-level security requirements or policies to low-level configuration settings to be deployed in the system. Furthermore, complementary configuration analysis approaches [Fu2001; AlShaer2005; Satoh2008; Alfaro2008; AlShaer2009; Basile2012] can be used to ensure that configurations, no matter whether manually authored or automatically refined from more abstract specifications, enjoy desirable properties, such as consistency, conflict-freeness, non-redundancy, etc.

Note how the majority of the above techniques strongly rely on the availability of formally-specified security policies and requirements. It is often recognized, however, that formal policies are yet unlikely to be adopted in industry, either because of exceeding complexity or lack of flexibility. In fact, no actual configuration management tool supports policy languages with formal foundations [Han2012] and industrially-accepted languages, such as the eXtensible Access Control Markup Language (XACML) [OASIS2003], tend to be very expressive and hard to formalize; for instance, several works provide formal semantics for different subsets of XACML [Bryans2005; Kolovski2007; Ni2009; Ahn2010; Ramli2011], but not for the full specification. At the same time, the emergence of standards such as the Security Content Automation Protocol [NIST2009] and the increasing availability of products implementing the functionalities of ITIL's Configuration Management System (CMS) and Database (CMDB) concepts, constitutes evidence of an increasing interest of the IT industry in topics related to configuration validation and change management, which so far received comparably less attention from researchers. Moreover, in many cases, administrators are "reluctant to define a whole security policy from scratch" [Alfaro2007] each time a change is necessary, and they rather prefer to directly modify existing configurations to cope with evolving security needs. For these reasons, **in this thesis we tackle the challenges of security configuration management from an opposite and complementary perspective to that of techniques, mostly applicable to phases (CM1) and (CM2), that require input high-level formal policies**. Instead, by pragmatically working on the basis of **low-level security configurations**, we target primarily phases (CM3) and (CM4). More specifically, we first focus on purely **syntactical approaches for configuration validation (CM4)** and we then move towards increasingly **semantics-aware analysis techniques for managing configuration change (CM3)**.

1.4.1 Syntactic Configuration Validation for Distributed Systems

Syntactic configuration validation is a technique whereby configuration checks, authored either by security experts or by system administrators, assess whether a system configuration complies with a given security policy or ensure it does not expose the system to security vulnerabilities. As such, it constitues a powerful tool to perform systematic monitoring, as required by phase (CM4). Standards [NIST2009] and tools [Nessus] implementing this concept have been increasingly adopted and have rapidly led to the growth of a knowledge base of machine-readable security checks [NVD]. However, these approaches limit the scope of checks to single hosts or operating systems, which makes it difficult to detect security issues that are due to the simultaneous misconfiguration of distributed system components. Our first objective is therefore focused on overcoming this drawback, to improve the applicability of automated configuration validation practices to distributed information systems.

Objective 1:

Extend the expressiveness of standard-based syntactic configuration validation languages to integrate configuration validation in the management of distributed information systems' security.

To achieve this objective we propose the following contribution. Because of (i) the heterogeneity and the potentially large number of configuration settings in real systems, and (ii) the explicit focus on distributed but interdependent misconfigurations, this contribution targets respectively the second and third challenges of the list presented in Section 1.3.3.

Contribution 1:

We elicit requirements for a syntax-based configuration validation language, and a corresponding interpretation engine, suitable to be employed in distributed environments and to be integrated with current configuration management practices and standards.

We then propose an extension (in terms of syntax and evaluation semantics) to standard-based configuration validation languages that fullfils such requirements and, specifically, improves the state of the art by allowing for a clear separation between the specification of check logic, check targets and the mechanisms for collecting to-be-checked configurations.

We describe a proof-of-concept implementation of both the language and its interpreter and discuss their integration in several scenarios that differ in terms of purpose and authorship of configuration checks and modality of invocation of the configuration validation process.

1.4.2 Formalization and Change Impact Analysis of JEE Authorizations

One of the advantages of purely syntactical configuration analysis is that it applies to virtually any kind of configuration independently from its semantics, as the actual settings — interpreted as simple common data types, e.g., strings, integers, booleans, etc. — are directly compared with the expected desired (or undesired) values by the means of a fixed set of operators. This is especially true when the gap between the configuration language's syntax and semantics is small, whereas, as the language expressiveness increases, it becomes more and more difficult to express syntactic checks that encode interesting semantic conditions. For instance, a rule-based access control configuration may contain rules with complex and mutually overlapping conditions: while a syntactic check looking for the exact same sequence of rules would be semantically *sound*, it may not be *complete*, i.e., different configurations that enforce the same policy would produce an alert despite being semantically equivalent. Moreover, different syntactic changes may have substantially different security implications, which in turn determine the severity of the misconfiguration and thus shall be taken into account when prioritizing remediation actions. For instance, all the changes that make the access control policy more restrictive may be considered less severe. The ability of semantically assessing the impact of configuration changes is not only useful to complement syntactic validation in phase (CM4), but it is also an important *what-if* analysis tool for phase (CM3), where changes are planned. In this case it can prevent inexperienced administrators from introducing unforeseen side effects by anticipating the result of their modifications.

In order to reason about the semantic properties of security configurations, we shall restrict to those that have a well-defined formal characterization, which is a prerequisite to provable soundness and completeness. Various security properties have been shown to correspond to safety or liveness conditions on labelled transition systems [Schneider2000; Ligatti2009]; however this characterization requires a model of the behaviour of the system, which may not be available in practice. Access control, in its most general formulation, is one of such properties, belonging specifically to the class of those enforceable by a system execution monitor [Schneider2000]. However, in many practical cases, it can be decoupled and modeled independently from the behaviour of the monitored system [Tripunitara2007; Habib2009; Crampton2012b].

The second objective of this thesis is therefore focused on evaluating the semantic impact of syntactic changes in access control configurations. In particular, unlike previous works in this area [Fisler2005; Liu2007], we aim at studying the formal semantics of authorization policies for hierarchical resources (like URLs) that are crucial to securing web applications, which have been employed more and more extensively as a lightweight front-end for business services in distributed information systems.

Objective 2:

Investigate the benefits of assigning formal semantics to an access control configuration language for web applications, especially when evaluating the impact of syntactic configuration changes.

As it aims at bridging the gap between security configurations' syntax and semantics, the main challenge concerning this objective is the first one (semantic gap) listed in Section 1.3.3, which we address in the following contribution.

Contribution 2:

We provide a denotational semantics for the access control configuration language of the JEE (Java Enterprise Edition) framework, one of the most widespread web application frameworks currently available. On top of this, we define a procedure to compare access control configurations with respect to their permissiveness and we prove its correctness.

Finally, we implement our model and evaluate it with respect to the operational semantics of existing JEE container implementations through automated software testing. The findings include not only positive results supporting the correctness of our semantics, but also evidence of discrepancies that led to the discovery of a previously unknown implementation error in the Apache Tomcat JEE container.

1.4.3 Multi-Layered Access Control Policy Refactoring

Although providing formal semantics for access control configuration languages is often enough to support interesting configuration analysis tasks for individual system components, it is well known that, in a distributed system, inter-component interactions have to be modeled too, because changes in the configuration of one component can easily affect the behaviour of other components. Previous works on anomaly detection in distributed firewalls and VPN gateways [Fu2001; AlShaer2005] address precisely this issue in the domain of network-layer access control. However, access control is pervasive within several different layers of IT infrastructures, e.g., network filtering and application-layer authorization policies are different forms of access control that typically cooperate in real scenarios. The access control process is distributed across several IT components, each one potentially operating on different architecture layers and residing on different hosts. For instance, a classical network firewall is able to take allow/deny decisions for network requests, having parameters such as IP addresses and TCP/UDP ports. A Web server instead handles a different kind of requests that rather belong to the application ISO/OSI layer, e.g., having parameters such as the requested URL. Moreover, the separation between network and application layers is typically not as neat. For example, many common services (e.g., the Apache

Web server, the MySQL database server or the anti-spam features of the sendmail mail server), perform access control based not only on application-specific parameters (e.g., respectively, URLs, tables, mail addresses), but can overlap with the lower layer (e.g., by filtering on the IP address of the requester). Conversely, modern firewalls are more and more capable of inspecting application-layer fields.

While such an inter-layer overlap allows for greater expressiveness, in practice, as argued in Section 1.3, more complexity increases the risk of misconfiguration and also contributes to the increase of IT management costs observed during the last decade(s). Hence, the last objective of this thesis is about studying how inter-layer relationships can be incorporated with the formal description of access control configurations to support a form of *inter-layer access control policy refactoring*, i.e., the task of finding the least permissive rewriting of a collection of policies that belong to different layers such that the global composed policy remains identical.

Policy refactoring is a means to accomplish several tasks that conceptually belong to phase (CM3), such as: (i) checking whether local policies can be simplified without changing the global one, in order to reduce management overhead, (ii) enforcing the least privilege principle in multi-layered policy-based access control systems, and (iii) adapting to changing security capabilities of single components.

Objective 3:

> *Assuming formal semantics is available for the access control configuration of distributed components lying on different architectural layers in a system, we aim to answer the following questions: Is there a notion of inter-layer policy overlap? Is there a refactoring of the components' configuration that removes such an overlap by preserving the global permissiveness?*

As fulfilling this objective requires dealing with independently-authored and distributed security configurations, both the third and fifth challenges of Section 1.3.3 are concerned. In particular, the modeling and exploitation of inter-layer policy interactions, identified as an open research problem in [Sloman2002], constitute key elements of our contribution.

Contribution 3:

> *We formally define the problem of multi-layered access control policy refactoring and we develop a necessary and sufficient condition to determine whether it admits a solution, together with a provably correct procedure to compute it.*
>
> *To this end, we embed a generic access control system into a structure that keeps track of the interactions among authorization decisions taken on different layers. We then define the semantics of composition of such access control layers and show that its inverse, namely decomposition, provides (when it exists) a solution to the*

problem of refactoring. Finally, we provide algorithms to test for decomposability as well as to compute (de)composition. Our model is inspired from database theory: we borrow key concepts from the literature on both constraint databases [Revesz1995] and provenance [Karvounarakis2012]. To prove the correctness of the decomposability condition, we extend a previous result of dependency theory, linking lossless join decomposition with so-called multivalued dependencies, to larger-than-boolean relations.

To assess the applicability of our approach in practice, we evaluate the algorithms with respect to various stochastic properties of input policies. The results show comparable performances with previous work on network security configuration analysis.

1.5 The PoSecCo Project

The work presented in this thesis has been carried out in the context of the European research project PoSecCo (Policy and Security Configuration Management)[2] [Posecco2011]. PoSecCo aims at enabling service providers (i) to achieve, maintain and prove compliance with security requirements stemming from internal needs, 3rd party demands and international regulations and (ii) to cost-efficiently manage policies and security configuration in operating conditions. Service providers are organizations that operate a distributed information system in order to deliver services to consumers. As such, they need to properly manage the security configuration of their infrastructure, which, as argued in Section 1.3, is a challenging task. To tackle the challenges of security configuration management, PoSecCo proposes to establish and maintain a consistent, transparent, sustainable and traceable link between high-level, business-driven security and compliance requirements on one side and low-level technical configuration settings of individual services on the other side. In the remainder of this section we first provide an overview on PoSecCo and then position our contributions with respect to the project's framework.

PoSecCo supports the entire security configuration management process (cf. Section 1.2) by the means of automated techniques where possible and by offering decision support where human interaction is inevitable. This is achieved through the combination of two complementary approaches:

The top-down approach (corresponding to phases (CM1) and (CM2)) comprises a collection of techniques that take as input the various laws, regulations, best practices and standards for security and compliance, capture them in the form of secu-

[2]Co-funded by the European Community under the Information and Communication Technologies (ICT) theme of the 7th Framework Programme for R&D (FP7) with grant agreement number 257129. `http://www.posecco.eu`.

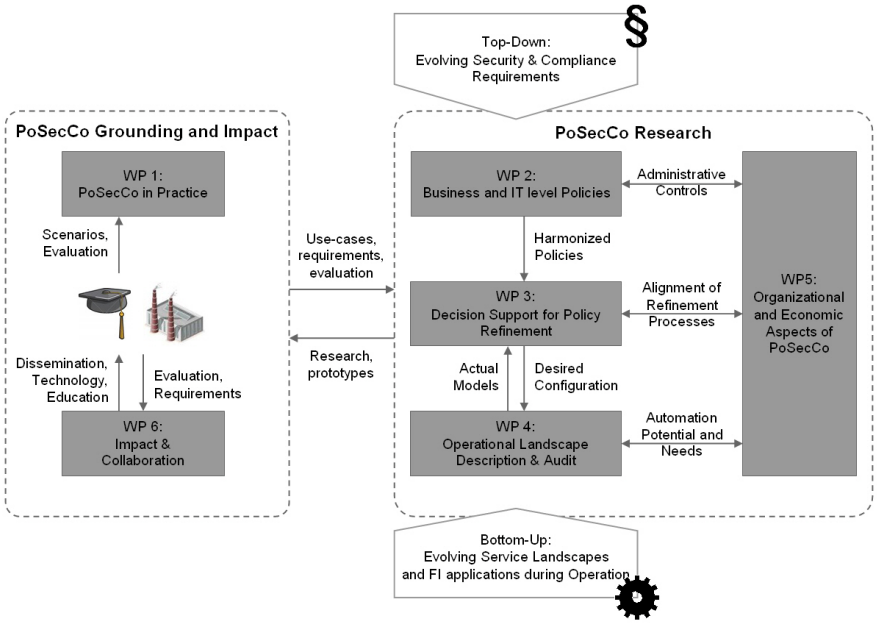


Figure 1.2: PoSecCo's architecture and work packages [Posecco2011].

rity policies, i.e., more detailed descriptions of security and compliance goals, and translate them into system-level configurations. All the top-down policy translation steps are recorded and structured in the so-called *policy chain*, which links high-level security policies and requirements with low-level configuration settings.

The bottom-up approach (corresponding to phases (CM3) and (CM4)) serves two main purposes. At policy *design time*, it builds a model of the service provider's system infrastructure, to be used as input by the top-down policy refinement tasks. This is done by interfacing and collecting information from existing network and configuration management software. At *run time*, it monitors the state of the system to detect discrepancies in either (i) the value of configuration settings, when found different from those derived by the top-down process, or (ii) in the behaviour of the system, when it is not compliant with the policies. When a discrepancy is detected, the information contained in the policy chain is leveraged to go back up to the linked high-level security requirements, which allows system administrators to better estimate the impact on security and compliance and to plan remediation actions.

The above description is summarized in Figure 1.2, which also depicts the organization of PoSecCo's work packages. The top-down tasks are split between work packages 2 and 3. The former handles the gathering of prose-specified security requirements and the formalization thereof in security policies referring to an abstract description of

systems and security properties. The latter transforms such policies into concrete configurations by supporting the user in progressively refining and enriching them with system and technology-dependent details. Where necessary, in order to choose among several possible alternative configurations, the user is asked to trade off cost with security level; e.g., to choose whether to enforce a channel confidentiality policy through network or application-level encryption. Such an evaluation is based on the cost models that are provided by the work package 5. Finally, the work package 4 is responsible for the bottom-up approach, whereby the system infrastructure is monitored to (i) provide other work packages with an up-to-date model of the system, and (ii) to detect deviations in either the configuration or the behaviour of the system. This is done by leveraging syntactic configuration validation techniques as well as *ad-hoc* log and process mining-based verification.

As it comprises the task of validating the security configurations of an entire service provider's (distributed) system infrastructure, the work package 4 constitutes a natural use case for our first contribution, where we propose a language to express configuration checks for distributed systems. The same work package also covers the task of assessing the impact of a misconfiguration with respect to the high-level security policies. This is aligned with our second contribution, which includes a provably correct procedure to compare different access control configurations for Web applications with respect to their permissiveness. Our third contribution is instead positioned at the interface between work packages 3 and 4. At the end of the top-down policy refinement process it is necessary to configure the systems according to the policies. Depending on the capabilities of the available security mechanisms, this may require configuring consistently multiple devices belonging to different architectural layers: for instance, a layer-3 firewall would not be sufficient to enforce an access control policy that involves application-layer parameters (such as URLs). Our third contribution allows to determine whether or not an access control policy can be enforced by the collection of policy decision points available in the system.

1.6 Structure

The rest of this manuscript is organized as follows.

Chapter 2 introduces a concrete example of distributed information system that is used to illustrate the different phases of security configuration management and that will constitute a common use case scenario for the contributions presented in the following chapters.

Chapters 3, 4 and 5 present respectively the three main contributions of the thesis. They all have a similar structure: first, they introduce and detail the respective technique; next, they describe its implementation and provide elements of evaluation; the

relevant related work is then presented and compared with the chapter's proposition; finally, a discussion, providing concluding remarks and observations, is presented and a synthesis concludes the chapter with a short summary of contributions and results.

Chapter 6 draws the conclusions and provides an outlook on future work and perspectives.

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.

—Leslie Lamport, "Security Engineering: A Guide to Building Dependable Distributed Systems"

2 Scenario

▷ *In order to concretely illustrate the security configuration management process, this chapter presents an example of a distributed information system built on top of a common, off-the-shelf open-source software architecture and thereby prototypic for many real-life scenarios. We consider the case of an imaginary service provider which is inspired by one of the use cases of the PoSecCo project. We introduce example policies reflecting internal as well as external security requirements. We then detail the service provider's system infrastructure from a technical standpoint. Finally, we describe how the various system components have to be configured in order to enforce the policies. Each of the remaining chapters of the thesis will make use of this scenario to exemplify in detail the chapter's objective and related contribution.* ◁

Chapter Outline

THE imaginary service provider ACME operates a system infrastructure accessible through the Internet and offering some custom functionalities developed as web applications. Among the services are offered by ACME, we consider the "ACME DEx" (Document Exchange) service, which allows customers to exchange EDI (Electronic Data Interchange) documents with their business partners through the Internet.

EDI is a standard defining the structure of messages suitable for the exchange of business documents — such as cheques, invoices or bills of landing — directly between computer applications (most commonly ERP systems) and without intermediary human involvement. The entities, typically different organizations, exchanging EDI documents are referred to as trading partners. Trading partners can either interact directly with each other, according to a peer-to-peer model, or rely on third parties, often named value-added networks, which provide additional services such as document transformation between different formats.

The ACME DEx service is an example of EDI value-added network. The application front-end for managing customers' trading partners and exchanging documents is implemented as a JEE web application. The application is split into two main parts: an administration console that lets customers manage the list of their trading partners, and a partner area exposing a web service interface for exchanging documents. Instances of this web application, each dedicated to one customer, are deployed in the Tomcat application server, under customer-specific context roots.

In the remainder of this chapter we describe the outcome of the first two phases of the configuration management process on the ACME scenario. As these steps lie outside the scope of this thesis, we assume that they are performed either manually or by (semi-)automated techniques such as the PoSecCo's top-down policy refinement (cf. Section 1.5). We first formulate a small yet illustrative set of example security policies (CM1). We then provide a technical overview of ACME's infrastructure, in terms of functionalities and system architecture, with a particular emphasis on the security mechanisms available in the system. Finally, we discuss how such mechanisms have to be configured in order to enforce the policies (CM2).

2.1 ACME's Security Policies

ACME faces a variety of security requirements coming from different sources and addressing different kinds of security needs. Generally, the possible sources of such requirements can be many, e.g., (inter)national legal or regulatory requirements, contractual agreements, best practices and standards, prioritized risk mitigation, etc. In this example we consider two such categories that are particularly relevant to ACME's scenario: **service level agreements** and **internal risk mitigation**. Considering that the involved stakeholders are (potentially large) organizations, the type of service offered

by ACME is likely to be regulated by a detailed service level agreement. We therefore consider both generic and customer-specific security aspects that may be part of such an agreement. Furthermore, as ACME operates an IT infrastructure that is constantly exposed to the Internet, the need of mitigating the risk of intrusion and data breaches constitutes a second major security concern. A subset of ACME's security policies that address examples of such requirements is presented in Table 2.1.

Client authentication.

Through the DEx service, interested parties exchange documents carrying information that is valuable to conduct their business and that is typically confidential. Client authentication, i.e., ensuring that senders' as well as recipients' identities are properly established, is therefore a crucial property that has to be guaranteed. Without authentication it would not be possible to ensure that messages are delivered only to the interested parties (thereby breaking confidentiality) or that message authorship cannot be rejected (non-repudiation). Authentication is also a fundamental part of access control, which is needed to ensure that only authorized actions are performed in the system. As such, a policy requiring client authentication is part of the DEx generic service level agreement.

Client authorization.

ACME's authorization policy for the DEx service is expressed according to the Role-Based Access Control (RBAC) paradigm, whereby users are assigned to roles which in turn refer to the permissions they are granted. For each customer there exist two roles, namely `dex-mgr` and `dex-tp`, representing respectively users belonging to the customer's organization and to those of its trading partners. The management console must be accessible only by members of the `dex-mgr` role, as it allows to administer customer-specific information, such as the list of its trading partners. Members of either the `dex-mgr` or `dex-tp` roles are instead allowed to exchange EDI documents through the web service interface available in the partner area.

Data integrity and confidentiality.

As EDI documents carry sensitive business information, it is important to ensure that no malicious third party can either access or temper with such information. This requirement is particularly critical whenever communications occur on untrusted channels which are not under the control of the interested parties. This is indeed the case for ACME, that offers its services over the public Internet. A policy specific to this purpose is therefore stated, which mandates the use of cryptography to guarantee the integrity and confidentiality of data transiting on untrusted networks.

Security Purpose	Policy	Source
Client authentication	The functionalities of the DEx service shall be available only prior to authentication	DEx generic service level agreement
Client authorization	Access to the DEx management console is granted only to members of the customer-specific `dex-mgr` role	
	Only members of either the `dex-tp` or `dex-mgr` roles can access the partner area where they can exchange EDI documents	
Data integrity and confidentiality	Cryptography shall be employed to protect sensitive information, such as EDI documents as well as confidential information about customers' business partners, when transiting on untrusted networks	
Geographic-based service restriction	Customer A has trading partners from all over the world except country Y, from which access is denied	Customer-specific service level agreement
	Customer B exclusively operates in country X, access to the service is denied from elsewhere	
Network access filtering	Customer A shall access the management console exclusively from its network that has a pool of assigned public IP addresses	Customer A's service level agreement
	A De-Militarized Zone (DMZ) is configured, being be the only network location directly reachable from the Internet where no sensitive assets shall be located	Internal risk mitigation
	The only host in the DMZ reachable from the Internet is a reverse proxy relaying the protocols required by the DEx service	

Table 2.1: Excerpt of ACME's security policies.

Geographic-based service restriction.

On top of the above generic policies that apply to all DEx users, premium customers with specific security requirements motivate the need of dedicated custom policies. For instance, as part of the agreement with its trading partners, customer A commits to never let document exchange occur with trading partners from country Y, which is considered untrusted. Customer B, instead, being part of country X's public administration, requires the access to the service to be restricted to country X's clients only.

Network access filtering.

Finally, we consider some example network access filtering policies stemming partially from customer A's service level agreement and partially from the risk mitigation plan put in place by ACME as a result of a risk analysis process. In order to minimize the risk of unauthorized access to the management console, customer A explicitly requires that access to the console shall be granted only to clients that are located within its own organization. A portion of the network named De-Militarized Zone (DMZ) is distinguished, where all the hosts that shall be reachable from the Internet are located. Sensitive assets such as database or application servers must be located in a different subnetwork, not directly reachable from the Internet. A second policy specifies that the DEx service should be accessible through a reverse proxy relaying the traffic to backend servers.

2.2 ACME's System Infrastructure

An overview of ACME's network topology and installed software components is shown in Figure 2.2. ACME's IP address space is split into two portions: the lower half of the interval of addresses (1.1.1.0/25) is assigned to the DMZ, hosting internet-facing services; the upper half (1.1.1.128/25) is the internal subnetwork where more sensitive and protect-worthy assets are located. A firewall (identified as `FW` in the picture) regulates the network traffic among these two subnetworks and the Internet.

Furthermore, in order to later help illustrate the enforcement of some of Table 2.1 policies, Figure 2.2 highlights three blocks of IP addresses within the Internet public address space: two are those assigned to all the ISPs belonging to respectively country X and country Y and the third one represents the address range assigned to ACME's customer A (simple and contiguous ranges have been chosen for the sake of conciseness).

Tomcat instances run inside the internal subnet, and are proxied by the Apache HTTP Server installed on the machine having address 1.1.1.1 within the DMZ. Requests for a customer-dedicated sub-domain of `acme.com` are forwarded by the reverse-proxy, with help of the module `mod_proxy`, to the respective customer-

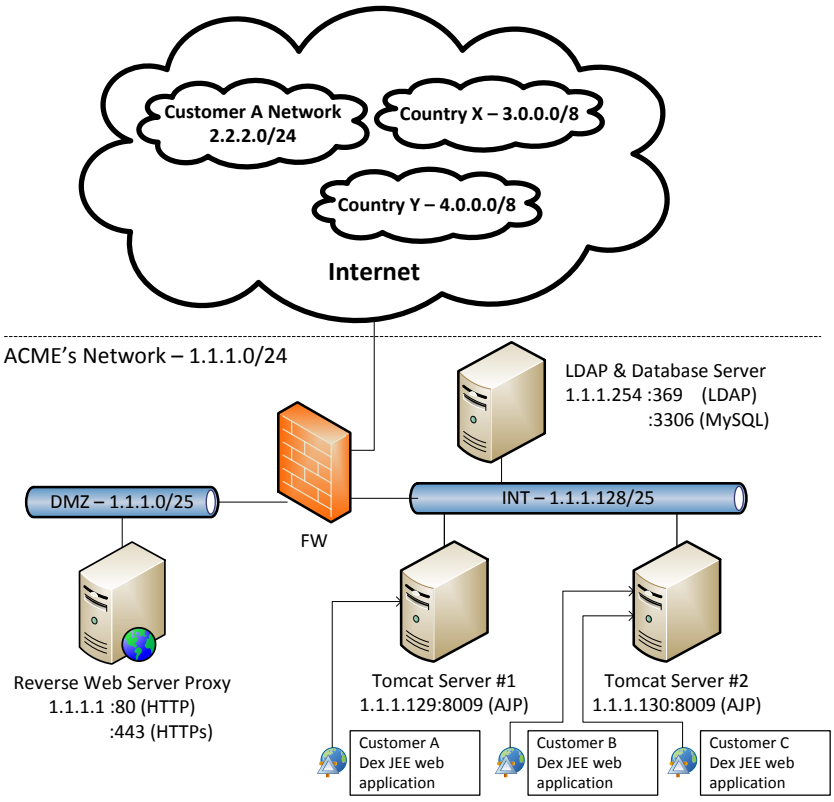


Figure 2.2: ACME's landscape.

dedicated instance of the web application via the Apache JServ Protocol (AJP). For brevity, we will only consider three such instances: one specific to customer A, deployed into the application server running on machine 1.1.1.129, and two others, specific to customers B and C respectively, both deployed on machine 1.1.1.130. Requests directed to, for instance, `https://cust-a.acme.com/` are dispatched to `ajp://1.1.1.129:8009/cust-a/` and likewise for other customers.

The reverse proxy also terminates incoming TLS connections thanks to the Apache module `mod_ssl`. The unencrypted HTTP requests, encapsulated in the AJP protocol, traverse then the firewall and reach the appropriate Tomcat instance where they get served.

The JEE web application implementing the DEx service is developed and maintained by ACME. It is divided into three main parts:

1. a static part welcoming users and providing them with public information, which is directly accessible from the web application context-root (e.g., for customer A,

`/cust-a/`);

2. a management console, reachable under the path `/manager/` relative to the root (e.g., `/cust-a/manager/`), that allows authenticated customers to manage the accounts of their trading partners;
3. a trading partner area organized as follows:
   (a) the `/partner/` location provides trading partners with information about exchanged documents and other partners;
   (b) the `/partner/edi/` location exposes a RESTful web service that allows trading partners to retrieve (via the HTTP GET method), submit (HTTP PUT), modify (HTTP POST) or delete (HTTP DELETE) incoming or outgoing EDI messages.

Role-based access control is performed at the level of the application servers by leveraging the appropriate declarative mechanisms standardized in the JEE servlet specification [Coward2003]. In order to authenticate users consistently across the different Tomcat instances, their credentials are looked up and matched in a central LDAP directory, that also provides Tomcat with the association between users and roles. Once a user is authenticated and his/her role established, an authorization check is performed to determine his/her access rights, according to the web application access control configuration.

The LDAP directory service for the management of user accounts is provided by an instance of OpenLDAP installed on another machine (1.1.1.254) harbored in the internal network. The same server hosts a MySQL database as well, used by the application servers for persistency.

2.3 ACME's Security Configurations

In this section we describe how the different system components appearing in Figure 2.2 can be configured to enforce the security policies listed in Table 2.1. We first describe in detail authorization (Section 2.3.1) and filtering (Section 2.3.2) configurations as these two categories constitute the focus of our second and third contributions (Chapters 4 and 5). For completeness, we then briefly cover authentication and encryption (Section 2.3.3). Overall, it will become evident that implementing a single policy often requires configuring consistently more than one component. This issue will be particularly relevant to illustrate our first contribution (Chapter 3), where we propose to increase the expressiveness of current configuration validation languages to allow the specification of checks spanning over multiple system components.

```
1  <security-constraint>
2    <web-resource-collection>
3      <web-resource-name>Forbidden Methods</web-resource-name>
4      <url-pattern>/manager/*</url-pattern>
5      <http-method>DELETE</http-method>
6      <http-method>PUT</http-method>
7    </web-resource-collection>
8    <auth-constraint/>
9  </security-constraint>
10 <security-constraint>
11   <web-resource-collection>
12     <web-resource-name>Management Console</web-resource-name>
13     <url-pattern>/manager/*</url-pattern>
14   </web-resource-collection>
15   <auth-constraint>
16     <role-name>dex-mgr</role-name>
17   </auth-constraint>
18 </security-constraint>
19 <security-constraint>
20   <web-resource-collection>
21     <web-resource-name>Partner Area</web-resource-name>
22     <url-pattern>/partner/*</url-pattern>
23   </web-resource-collection>
24   <auth-constraint>
25     <role-name>dex-mgr</role-name>
26     <role-name>dex-tp</role-name>
27   </auth-constraint>
28 </security-constraint>
```

Figure 2.3: Security constraints in the deployment descriptor (web.xml).

2.3.1 Authorization

To enforce the authorization policy for ACME's DEx service, it is sufficient to include the snippet presented in Figure 2.3 in the *deployment descriptor* of each DEx web application. The deployment descriptor is a configuration file written according to a standard XML-based language whose syntax and (informal) semantics are defined by the JEE servlet specification [Coward2003]. It allows to configure several aspects of a web application, such as the mappings from URL patterns to the corresponding Servlets, the default error pages, etc. Security features, which are in the scope of this thesis, are among such aspects. In particular, this chapter introduces, by the means of examples, authorization and authentication configuration settings.

To configure authorization, three so-called *security constraints*, constraining the access to all the URL paths prefixed by either `/manager/` or `/partner/` (lines 4, 13 and 22), are specified. Access privileges are granted by listing authorized roles in the

`auth-constraint` tag within each security constraint. In this case only `dex-mgr` members can access the management console, while both `dex-mgr` and `dex-tp` members are granted access to the partner area (lines 15–17 and 24–27 respectively). If the `auth-constraint` tag is empty, then the access to the corresponding resources is forbidden to anyone. This is used to prevent any access to the management console through the `PUT` and `DELETE` HTTP methods that are not implemented by the web application (lines 4–6 and 8).

Note that different security constraints may be specified for the same URL or for overlapping URL patterns, in which case the actual policy is determined according to a set of composition rules that are informally described in the JEE Servlet specification. For instance, a denial constraint (cf. lines 1–9) always takes precedence on other constraints for the same URL pattern (cf. lines 10–18).

When authoring or modifying security constraint configurations, system administrators may commit mistakes due to disattention or misinterpretation of the language evaluation semantics. In Chapter 4 we will show how, in fact, minor configuration changes can yield unpredicted and sometimes counterintuitive outcomes. To prevent this issue, we will provide a formal semantics for the language of security constraints which is suitable for static verification tasks, such as determining the impact on permissiveness of a change in the configuration.

2.3.2 Network and Geographic-based Filtering

Network filtering policies are typically enforced either by dedicated network equipment (usually routers equipped with firewall functionalities) or at the endpoint hosts, e.g., by the operating system, or by a personal firewall, or even directly within the client or server application.

In ACME's scenario, the different filtering policies are enforced by two distinct components: namely the Internet-facing firewall and the reverse proxy. The system infrastructure presented in Figure 2.2 features a common layer-3 firewall, that is, a network filtering device incapable of inspecting and tracking protocols (e.g., HTTP, FTP, SMTP, etc.) lying above the transport layer in the ISO/OSI stack. As noted in [AlShaer2004], configurations for such a firewall can be conveniently expressed in a generic format that is vendor-independent yet specific enough to be translatable with minimum effort to most vendor-specific firewall configuration languages. Such generic configurations are constituted by sets of rules of the form:

```
<order> <protocol> <ip_s> <ip_d> <port_s> <port_d> <action>
```

where `<protocol>` identifies a transport protocol (e.g., TCP or UDP), `<ip_s>` and `<ip_d>` are respectively IP source and IP destination (either single addresses or subnetworks), `<port_s>` and `<port_d>` identify either TCP or UDP ports, and `<action>`

```
1   ord proto ip_s    ip_d          port_s  port_d  action
2   1   TCP   any     1.1.1.1       any     80      accept
3   2   TCP   any     1.1.1.1       any     443     accept
4   3   TCP   1.1.1.1 1.1.1.129     any     8009    accept
5   4   TCP   1.1.1.1 1.1.1.130     any     8009    accept
6   5   any   any     any           any     any     deny
```

(a) Firewall configuration expressed in generic format [AlShaer2004].

```
1   <VirtualHost *:443>
2     ServerName cust-a.acme.com
3     <Location /partner>
4       Order Allow,Deny
5       Allow from All
6       Deny from 4.0.0.0/8
7     </Location>
8     <Location /manager>
9       Order Deny,Allow
10      Deny from All
11      Allow from 2.2.2.0/24
12    </Location>
13  </VirtualHost>
```

(b) Apache web server IP-based access restriction (httpd.conf).

Figure 2.4: Network and geographic-based filtering configuration.

is either `accept` or `deny`. For every network packet, the firewall interprets the rules by ascending values of the `<order>` field and executes the action associated to the first rule that matches to the packet. The ruleset shown in Figure 2.4a, interpreted according to the above informal semantics, enforces the general (i.e., not customer-specific) network filtering policies of Table 2.1. In fact, the reverse proxy (1.1.1.1) is the only host reachable from the Internet on the standard HTTP(s) ports. Moreover, from the DMZ to the internal network, only communications coming from the proxy on the AJP port (TCP 8009) are allowed, which is needed to relay the HTTP requests to the backend application servers. All other packets are dropped by default (cf. rule no. 4).

To enforce customer-specific filtering policies, such as the geographic-based restrictions in Table 2.1, it is instead necessary to discriminate network-layer as well as application-layer protocol features; for instance communications directed to different customers are distinguishable only if HTTP requests are inspected. In our example this can be done by the reverse proxy, prior to redirecting requests. For every customer-specific filtering policy, there exists a configuration similar to the one expressed for customer A in Figure 2.4b. Note how this configuration enforces all customer A's network filtering policies. Trading partners can access from everywhere except from country Y (lines 5–6) and only clients from customer A's network can access to the management

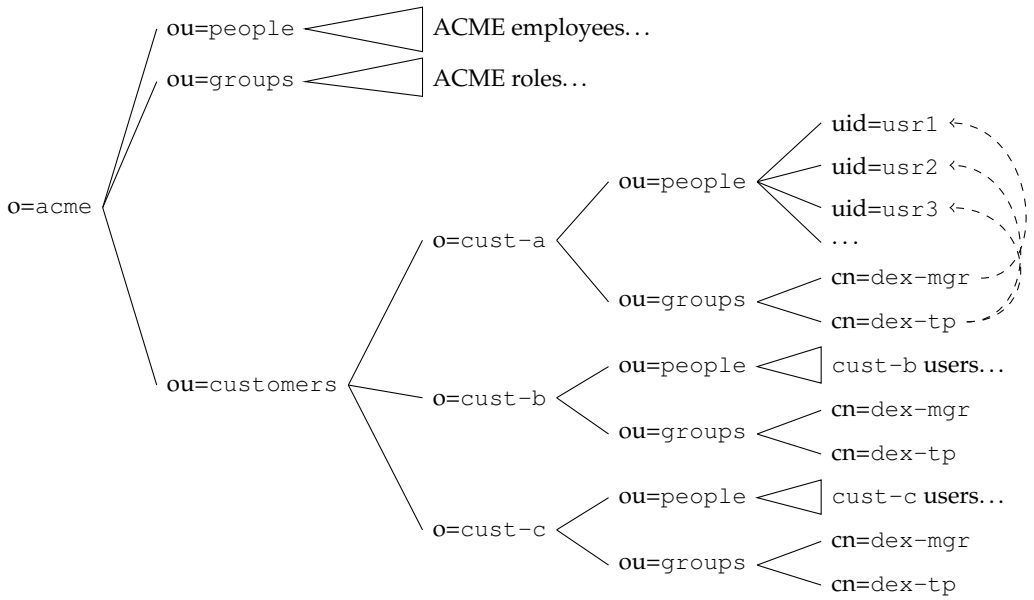


Figure 2.5: ACME's LDAP directory structure.

console (lines 10–11). The `Order Allow,Deny` and `Order Deny,Allow` directives implement respectively a blacklist and whitelist access control behaviour (line 9).

Note that there exists a partial overlap between the firewall and the reverse proxy policies. In particular, all requests directed to the IP address of the proxy (1.1.1.1) will be filtered a first time by the firewall and a second time by the proxy itself. The overlap consists in the proxy being able to also inspect the network part of the request, namely the client's IP address, which has already been inspected by the firewall. Maintaining a consistent network filtering policy distributed across different system components, however, requires more effort and it is more prone to error than managing it in a centralized point. In Chapter 5 we will contribute to address this issue by developing a theory that allows to model inter-layer policy interactions and to refactor policies by removing (when possible) unnecessary overlap without changing the global permissiveness.

2.3.3 Client Authentication, Data Integrity and Confidentiality.

Clients need to be authenticated prior to serving their requests. According to the JEE security model, web applications shall delegate this task to the application server, which must implement mechanisms to let the user prove his/her identity by exhibiting the appropriate credentials and, once established, associate the user's identity to a session which will be the context of all subsequent requests from the same user.

```
1   <Context debug="0" reloadable="true">
2     <Realm  className="org.apache.catalina.realm.JNDIRealm"
3             connectionURL="ldap://1.1.1.254:389"
4             userPattern="uid={0},ou=people,o=cust-a,ou=customers,o=acme"
5             roleBase="ou=groups,o=cust-a,ou=customers,o=acme"
6             roleName="cn"
7             roleSearch="(uniqueMember={0})"
8     />
9   </Context>
```

(a) Configuration of context-specific LDAP authentication realm (context.xml).

```
1  <login-config>
2    <auth-method>FORM</auth-method>
3    <form-login-config>
4      <form-login-page>/login.html</form-login-page>
5      <form-error-page>/error.html</form-error-page>
6    </form-login-config>
7  </login-config>
```

(b) Configuration of authentication method (web.xml).

Figure 2.6: Configuration of authentication in Tomcat.

There are two authentication-related features that have to be configured in any JEE application server: (i) the specification of a method to verify users' credentials, as well as to associate them with the roles they are granted, and (ii) the selection of an authentication protocol for each web application. The former is mostly not standardized, i.e., different products feature different vendor-specific configuration directives, whereas the latter is defined as part of the JEE servlet specification.

In our example user accounts and roles are maintained in a centralized LDAP repository. This is a common practice whenever authentication information has to be available to several distributed components, which is the case of ACME's multiple Apache Tomcat instances. The structure of ACME's LDAP directory is shown in Figure 2.5. Every path in the tree, from any node to the root, uniquely identifies an entry in the directory, for instance the path `uid=usr1, ou=people, o=cust-a, ou=customers, o=acme` identifies the user `usr1` of customer A's dedicated application. Some entries can represent groups of other entries, which is used to model roles. For example, the entry `cn=dex-tp, ou=groups, o=cust-a, ou=customers, o=acme` represents the `dex-tp` role for customer A's users (associated to both users `usr2` and `usr3`).

In order to instruct the application server to make use of the LDAP as a source of authentication information, one must provide the appropriate queries to search the directory for a user and her/his roles. In the case of Tomcat, such queries are provided in the so-called *realm* configuration. A realm represents any source of authentication in-

```
1   <VirtualHost *:443>
2     ServerName cust-a.acme.com
3     ProxyPass / ajp://1.1.1.129:8009/cust-a/
4     ProxyPassReverse / ajp://1.1.1.129:8009/cust-a/
5     SSLEngine on
6     SSLCertificateFile /path/to/acme.com.cert
7     SSLCertificateKeyFile /path/to/acme.com.key
8     <Location />
9       SSLRequireSSL
10      SSLCipherSuite HIGH:!aNULL:!MD5
11    </Location>
12  </VirtualHost>
```

Figure 2.7: TLS configuration of an Apache virtual host in reverse proxy mode (httpd.conf).

formation. The configuration excerpt in Figure 2.6a illustrates how the realm specific to LDAP directories (JNDIRealm) is configured for customer A's web application. Such a configuration is included in the `context.xml` file bundled with each web application. The choice of which authentication method to use is instead specified in the deployment descriptor of each web application. As shown in Figure 2.6b, the chosen authentication method makes use of an HTML form to collect users' credentials; the name of the pages that respectively display the form to the users and report login errors are configured by lines 4 and 5. Of course this configuration assumes that the underlying channel guarantees confidentiality between client and server, which is indeed explicitly required by ACME's policies.

The natural way to enforce the policy requiring the integrity and confidentiality of data in transit is to force clients to establish a secure transport channel with the server. In ACME's scenario the reverse proxy is implemented by an Apache web server, which can be configured to enforce such a policy as shown in Figure 2.7. The snippet sets up a *virtual host* on TCP port 443 (lines 1–2) dedicated to customer A such that: (i) TLS (Transport Layer Security) is enabled and configured to rely on a given server certificate and a corresponding private key to perform cryptographic operations (lines 5–7), and (ii) any request can be only served if TLS is enabled (line 9) and the selected ciphersuite meets some minimum security requirements[1] (line 10). Moreover, requests directed to the customer-specific subdomain are dispatched to the appropriate dedicated web application deployed in one of the backend Tomcat application servers (lines 3–4). A comparable configuration is provided as well for the other customers, with the `ServerName` and `ProxyPass` directives adapted accordingly.

[1] In particular, the use of the NULL encryption algorithm, which transmits data in plaintext, is forbidden as well as the use of MD5 hashing to compute message authentication codes, which is often considered dangerous as more and more prone to collision.

We know that the only way to avoid error is to detect it and that the only way to detect it is to be free to inquire.

—J. Robert Oppenheimer, in L. Barnett, "Life", Vol. 7, No. 9, International Edition (24 October 1949), p.58

3

Syntactic Configuration Validation for Distributed Systems

▷ *Configuration validation is a key activity of the security configuration management process which allows to detect security vulnerabilities caused by system misconfiguration. Existing tools and approaches that automate this task typically perform a syntactic comparison between the actual system configurations and description of an expected state, which is provided in the form of a collection of configuration checks expressed in a machine-readable language. Existing configuration validation languages, however, implicitly fix the scope of checks to entire machines or operating systems, which makes it hard and sometimes impossible to express configuration checks for distributed, fine-grained software components.*

In this chapter we elicit requirements for a configuration validation language, and a corresponding interpretation engine, suitable to be employed in distributed environments and to be integrated with current configuration management practices and standards. We then propose an extension (in terms of syntax and evaluation semantics) to standard-based configuration validation languages that fullfils such requirements and, specifically, improves the state of the art by allowing for a clear separation between the specification of check logic, check targets and the mechanisms for collecting to-be-checked configurations. We describe a proof-of-concept implementation of both the language and its interpreter and discuss their integration in several scenarios that differ in terms of purpose and authorship of configuration checks and modality of invocation of the configuration validation process. ◁

Chapter Outline

SYSTEM misconfiguration, as argued in Chapter 1, is a major cause of security incidents, which motivates the need to continuously monitor the configuration of security-critical system components. This activity is often referred to as *configuration validation* and it is common to several practical use cases. First, configurations shall be checked for compliance with respect to high-level security policies. Moreover, even though policy-compliant, a system may still expose critical security vulnerabilities, which in many cases can be mitigated by proper configuration. For this reason software vendors issue an increasing number of security advisories, but system administrators often struggle to understand if a given vulnerability is exploitable under their particular conditions and requires immediate patching. Finally, as a means to preventively increase system security, it is often recommendable to check whether configurations conform to the best-practices and guidelines that are typically provided in the form of prose documentation by security experts.

Performing above tasks manually is clearly (i) time-consuming, which may require to restrict the scope of analysis through sampling, and (ii) hindered by the fact that many essential pieces of information, such as policies, security advisories and best practices, are expressed in prose, which can be too broad or ambiguous and therefore subject to misinterpretation. Recent trends aim at mitigating this issue by promoting standards for security automation, e.g., the Security Content Automation Protocol (SCAP) [NIST2009], provided by the National Institute of Standard and Technology (NIST), whose specifications receive a lot of attention in the scope of the configuration baseline for IT products used in U.S. federal agencies. Among other specifications, SCAP comprises the Open Vulnerability Assessment Language (OVAL) [Baker2012] that allows to specify machine-readable security checks to facilitate the detection of vulnerabilities caused by misconfiguration. While this represents an important step towards the standardization and exchange of security knowledge, OVAL focuses on the granularity of single hosts and operating systems, and as such cannot be easily applied to fine-grained and distributed system components.

To address these limitations and make the advantages of SCAP applicable to distributed system infrastructures, this chapter proposes a OVAL-based language for the declarative specification and execution of configuration checks targeting collections of fine-grained components in a distributed environment. This approach improves the state of the art in that it clearly separates the checks logic from the specification of their target systems and from the retrieval of the to-be-checked configuration settings, for which it integrates with existing system management procedures and technologies, e.g., Configuration Management Databases (CMDB) as defined in the IT Infrastructure Library (ITIL).

The rest of the chapter is structured as follows: Section 3.1 describes a set of use case scenarios for configuration validation, exemplified with configuration checks that are applicable to ACME's distributed system introduced in Chapter 2. From the analysis

of such use cases, we derive requirements for a configuration validation language to describe checks for distributed systems. Section 3.2 introduces the main SCAP specifications and highlights their limitations with respect to the requirements. Section 3.3 presents a formal language that builds on and extends the OVAL standard where checks' target can be specified intensionally as queries over the properties of distributed software components. Section 3.4 defines the interpretation of such intensional targets on the basis of a data source providing extensional information about the components of a system infrastructure. Section 3.5 describes the implementation of both the language and its interpretation semantics in a tool that relies on an external CMDB as data source and that integrates configuration validation with state-of-the-art system and configuration management technologies. Section 3.6 compares our proposal with related work on configuration validation. In conclusion, Section 3.7 presents a discussion of some key technical issues and Section 3.8 synthesises our contribution.

3.1 Motivating Scenarios and Requirements

Use case scenarios for configuration validation differ in terms of periodicity, validation scope and authorship of configuration checks. In this section, we describe four examples of such scenarios that pose challenges to the automatic validation of configurations in distributed environments. From these premises, we then drive requirements for the design of a configuration validation language suitable to express configuration checks for distributed systems.

3.1.1 Scenarios

Vulnerability Assessment (S1).

Upon the disclosure of a new security vulnerability of end-user applications or software libraries, system administrators need to investigate the susceptibility of their system. First, they need to check for the presence of affected release and patch levels. Second, they need to check whether additional conditions for a successful exploitation are met. Such conditions often concern specific configuration settings of the affected software, as well as the specific usage context and system environment. The automation of both activities with the help of machine-readable vulnerability checks decreases time and effort required to discover a system vulnerability and, at the same time, increases the precision with which the presence of vulnerabilities can be detected.

As an example, [CVE-2011-3190] reported a vulnerability in the AJP connector implementation of several Apache Tomcat releases, which, however, only applies under certain conditions, e.g., if certain connector classes are used, and reverse proxy and Tomcat do not use a shared secret. A machine check looking at the Tomcat release level

and related configuration settings could be easily provided by the application vendor (Apache Software Foundation). An example for a critical security bug in a software library is [CVE-2012-0392] which describes a vulnerability in Apache Struts, a common framework to support the Model-View-Controller paradigm in JEE web applications. The detection of this vulnerability is made more problematic by the fact that end-users typically do not know if applications installed in their environment make use of such a library, and they cannot rely on the presence of a well-established security response process at each of their application vendors. Thus security bugs may be dormant in libraries without the end-user being aware.

Configuration Best-Practice (S2).

This scenario focuses on establishing if best practices are followed. During operations time, system administrators need to periodically check whether the system configurations follow best-practices, for single and distributed system components. Today, these are often described in prose and evolve over time thus requiring continuous human intervention. Typical examples for best-practices are the Tomcat security guide from OWASP [OWASP2007], and the SANS recommendations for securing Java deployment descriptors [SANS2010]. Configuration best-practices may also cover a set of distributed components, e.g., the how-to about Apache HTTP server as a reverse proxy for Apache Tomcat [ASF2014a].

Compliance with Security Policy (S3).

This scenario focuses on the periodic validation of landscape specific configurations implementing the designed policy. A configuration policy specifies a set of mandated configuration settings that an organization expects to be active in its system, namely a *golden configuration*. As discussed in Chapter 1, such a configuration is typically the result of a top-down refinement process, which started at system design time with the elicitation of security requirements and the specification of high-level policies, and ends with a selection of security mechanisms whose behavior depends on the said configuration.

Configuration checks aiming to assess the compliance with a given security policy strongly reflect a particular system and environment, and are therefore authored by the end-user organization rather than by externals, as in the previous scenarios. Moreover, they target specific software component instances (e.g., a particular application server running on a specific machine), rather than generic predicates over software components (e.g., all the application servers in the network).

Prevention of insecure application execution (S4).

While the previous scenarios assumed that configuration validation happens periodically, we now consider the need of applications to automatically check configurations at runtime. In this scenario configuration validation becomes a preventive security con-

trol, in that the detection of insecure or non-compliant system states is linked to an application's runtime. Upon startup or invocation of a given application functionality, the application checks its own configuration as well as that of all other components that compose its software stack or with which it interacts (operating system, database, etc.) and behaves accordingly.

By using any of the above-discussed types of configuration checks, it is possible to check if the entire system and application stack comply with an expected state before allowing the execution of critical functionalities. An online shop application, for instance, could prevent any purchase in case the TLS configuration of the web server is incorrect, in order to protect customers from man-in-the-middle attacks.

Example 3.1: Example Checks

This example illustrates different configuration checks for some of the above scenarios instantiated on the ACME distributed system, which was introduced in Chapter 2.

SANS recommendations.

In [SANS2010], SANS recommends to configure the cookie-based session handling for JEE web applications such that (i) session cookies are marked as *http-only*, hence the browser won't allow malicious Javascript code to steal them, and (ii) they are transmitted over a secure (encrypted) channel. Moreover, in order to hinder session hijacking, the session timeout should be set to a value greater than 0 (which means infinite) and less than a maximum amount (e.g., 15 minutes).

To achieve this, the following configuration snippet has to be included in the deployment descriptor of the web application:

```
1  <session-config>
2    <cookie-config>
3      <http-only>true</http-only>
4      <secure>true</secure>
5    </cookie-config>
6    <session-timeout>15</session-timeout>
7  </session-config>
```

In particular, the cookie settings are an example of recommendation that only applies to web application containers that comply with the most recent releases of the Servlet specification (i.e., $\geq$ version 3.0).

A corresponding *best practice check* (cf. Scenario (S2)) that verifies whether cookies and sessions are properly secured should then be structured as follows:

Check Content SANS best practices are followed if all the below conditions are true:

1. access to session cookies is prevented (`<http-only>` set to `true`),
2. cookies are transmitted securely (`<secure>` set to `true`),

3. session timeout is configured (`<session-timeout>` comprised between 1 and 15).

Check Target tests no. 1 and 2 apply to JEE web applications deployed in a container that supports a version of the Servlet specification greater than or equal to 3.0, whereas test no. 3 applies to JEE web applications deployed in any container.

ACME access control configuration.

As shown in Section 2.3, a golden configuration that enforces ACME's access control policy embraces configuration settings of several distributed system components, i.e., the realm definition of each Tomcat instance, as well as the deployment descriptor of each Java web application instance.

The following is the description of a *policy compliance check* (cf. Scenario (S3)) for customer A's access control policy. Note how its target refers to a specific system component, as opposed to the previous check that applies to all components that satisfy certain conditions.

Check Content ACME's access control policy for customer A is satisfied if both the following conditions are true:

1. the security constraints in the deployment descriptor grant the access to all URLs matching to the pattern `/manager/*` to the role `dex-mgr`, whereas both `dex-mgr` and `dex-tp` roles have access to the `/partner/*` sub-hierarchy,
2. the authentication realm refers to the LDAP server located at 1.1.1.254 and fetches users/roles information under the path `o=cust-a, ou=customers, o=acme`.

Check Target both tests apply to customer A's dedicated instance of the DEx web application, deployed on the application server running on the machine 1.1.1.129.

Synthesis.

Table 3.1 summarizes the characteristics of each scenario. Note that a given configuration check may be processed in the context of several scenarios. For example, a best-practice check recommended by a software vendor and considered during system design, may be later adapted and integrated into an organization's policy. A recommendation related to the session timeout of web applications, for instance, would be refined by an organization to reflect its particular policy. Furthermore, the different configuration checks described in the context of scenarios (S1) to (S3) are likely to be combined in reality to form complex checklists, which produce reports on the security status of an entire IT infrastructure. Checklists are mainly useful to help humans understand, score and prioritize checks results. The execution of such checklists can be either performed manually by system administrators or can be part of the automated provisioning lifecycle of the software components managed by a Configuration Management Systems (CMS). In contrast, in the context of scenario (S4), an application executes single specific configuration checks in order to establish whether critical functionalities

	Validation goal	Check authors[1]	Trigger & periodicity	Scope	Check struct.
S1	Detect whether system components are susceptible to a known vulnerability	External[2]	Upon vulnerability disclosure	All instances of a given software component	
S2	Check whether configurations follow best-practices	External[2]	Periodical	Single system components or a set of related ones	Checklist
S3	Check whether a configuration policy is in place	Internal[3]	Periodical	Single or multiple system components according to a given policy	
S4	Prevent execution in case of misconfiguration	Any	At runtime	Any	None

[1] From the perspective of an end-user organization
[2] E.g., researchers, software vendors
[3] E.g., security administrators

Table 3.1: Summary of scenarios' characteristics

need to be disabled due to system misconfiguration. As no human is directly involved in this process, there is no particular need to organize and structure check results in a checklist.

3.1.2 Requirements

We elicit requirements for a configuration language whose purpose is to allow the specification of unambiguous, machine-readable checks that can be used to validate configuration settings of distributed system components. Key requirements include the possibility of associating checks with abstract target systems (i.e., described by a query on their properties) as well as concrete ones, and the separation between the checks' logic and the retrieval of configuration settings.

(RL1) The language must support the definition of configuration checks for diverse software components (e.g., network-level firewalls or application-level access control systems) and diverse technologies.

(RL2) The language must be expressive enough to cover new technologies or configuration formats without requiring extensions. This would avoid the need to update the language interpreter every time a new extension is published.

(RL3) It must be possible to specify target components by defining conditions over properties such as name, release, and supported specification, or over the existence of relationships between components. This is necessary in cases where externally provided checks must be applied to all instances of the affected software components (scenarios S1 and S2).

(RL4) Motivated by scenario S3, it must be possible to specify target components by referring to specific instances of a software component.

(RL5) It must be possible to validate the configurations of different, potentially distributed system components within one check.

(RL6) Checks must be uniquely identifiable, declarative, standardized and certifiable, to support trusted knowledge exchange among security tools and stakeholders, e.g., software vendors, security experts, auditors, or operations staff.

(RL7) The language must support parametrization in order to adopt externally provided checks to a specific configuration policy.

(RL8) The specification of checks must be separated from the description of the mechanisms to collect the involved configuration settings from the actual system infrastructure. This separation of concerns is required in situations (e.g., scenarios S1 and S2) where the roles of check authors and system administrators are separated.

(RL9) It must be possible to collect, structure and prioritize checks to facilitate human consumption of large collections of checks and related results.

3.2 Security Content Automation Protocol

The Security Content Automation Protocol (SCAP) [Waltermire2011; NIST2009] is a suite of specifications that support automated configuration, vulnerability and patch checking, as well as security measurement. Among other specifications, SCAP comprises a language for the definition of checklists (XCCDF), a language that allows the specification of security checks to detect misconfiguration (OVAL), and a language for defining classes of platforms (CPE).

3.2.1 Extensible Configuration Checklist Description Format

The Extensible Configuration Checklist Description Format (XCCDF) [Ziring2008] is an XML-based language to represent a structured security checklist. An XCCDF document

consists of Rules, each of which corresponds to a recommendation in a piece of guidance. The other structures in XCCDF, namely Groups Profiles and Values, serve to organize and refine Rules. In addition to supporting the structuring of prose guidance and compliance documentation, XCCDF Rules also contain Check elements, which support the automated validation of systems, by referencing or encapsulating machine-readable check specifications.

When processing such checks, an XCCDF interpreter invokes the appropriate external interpreter, according to the language each check is expressed with. The latter performs the check and returns a result value. Combining the results of all individual rules, the XCCDF interpreter returns *Pass* if the recommendation has been followed and *Fail* if not. Thus, an XCCDF document serves not only for documenting the desired security state of a system, but it can be actively used to evaluate the system with respect to the security guidance. For the latter case, though, it is necessary to rely on some external language that XCCDF check structures can reference and which describes how to evaluate compliance with recommendations in an automated way. The language that serves this purpose within SCAP is OVAL.

3.2.2 Open Vulnerability Assessment Language

The Open Vulnerability Assessment Language (OVAL) [Baker2012] is an XML-based community standard to promote open and publicly available security content. OVAL checks for the presence of vulnerabilities or desired configuration on a computer system. It defines three XML schemas:

OVAL System Characteristics: it represents configuration information of systems for testing;

OVAL Definition: it encodes the check logic to test for a specific state of the configuration (vulnerability, policy-compliance, patch level, etc.); and

OVAL Results: it reports the results of the assessment, i.e., the output of a comparison of an OVAL Definition against an OVAL System Characteristics instance.

OVAL allows to define how to check for misconfigurations by means of the following constructs: definitions, tests, objects, and states. An OVAL definition defines a boolean combination of tests. Each test defines an evaluation over an object and (optionally) a state. The OVAL object represents the configuration information that has to be collected from a system and then evaluated against the expected values defined within the state. The OVAL test can require to assess if the object exists in the system under analysis and/or how many of the collected objects satisfy the state. For each platform supported by OVAL, a schema extension defining tests, objects, and states detailing the properties to examine have to be provided. This either requires tool ven-

dors supporting the OVAL to constantly update the language interpreter, or leads to a fragmented market where tools only support a subset of the language.

3.2.3 Common Platform Enumeration

The Common Platform Enumeration (CPE) [Buttner2009; Cheikes2011; Parmelee2011] is an XML-based standard for the specification of structured names (CPE names) for identifying IT systems, software, platforms, and packages. It consists of four modular specifications: CPE Naming, CPE Language, CPE Dictionary and CPE Matching.

CPE Naming [Cheikes2011] is the base specification defining the format of CPE names, which are represented as URIs. Each name consists of the prefix `cpe:` followed by up to seven different parts used to compose consistent and unique names. The parts are: platform (one of [h]ardware, [o]perating system, or [a]pplication), vendor, product name, version, update level, edition, and language associated with the specific platform. As an example, `cpe:/o:microsoft:windows_2000::sp4:pro` represents an operating system developed by Microsoft (Windows 2000, service pack 4, professional edition).

To identify more complex platform types, the CPE Language offers boolean operators to combine different names, e.g. the AND operator allows to identify a platform with a particular operating system AND a certain application. In this way the CPE Language enables the CPE name for the operating system to be combined with the CPE name for the application.

CPE names are collected within the CPE Dictionary. Its purpose is to provide a source of all known CPE names as well as to bind descriptive prose and diagnostic tests to a CPE name. These metadata include a title, notes, and an automated check to determine if a given platform matches the CPE name. The automated checks can be expressed in OVAL.

Finally, the CPE Matching specification [Parmelee2011] includes an algorithm to establish if two names are equal, if one of the names represents a subset of the systems represented by the other, or if the names represent disjoint sets of systems.

3.2.4 Analysis of SCAP Specifications

SCAP represents a comprehensive effort to standardize the representation of security knowledge in order to foster the collaboration of security practitioners and tool interoperability. As explained below, however, several factors limit its applicability to distributed environments, in particular with regard to fine-grained targets such as software libraries, Web applications or Web services. Furthermore, SCAP specifications do not leverage standards and technologies in the area of system and configuration

management, in order to, for instance, separate check logic and information about configuration retrieval.

The CPE specifications are promising candidates to express the target systems of configuration checks. However, while the CPE Naming and CPE Matching specifications allow the definition and comparison of single software components according to properties such as vendor or product name, the CPE Language specification does not meet (RL3) with regard to the need to express relationships among software components. It supports the specification of a complex platform through a logical condition over several CPE Names, but the semantics of their relationship is not explicitly defined. The typical interpretation used in many CVE entries is that a complex platform condition is met as soon as all software components are installed on the same machine. This interpretation, however, is in many cases not sufficient to state that a vulnerability exists. The vulnerability described by [CVE-2003-0042], for instance, is only exploitable if Tomcat actually uses a given JDK version, the mere presence of both components on the same system is not sufficient. This interpretation is even more misleading if vulnerabilities are caused by combinations of client-side and server-side components, e.g., [CVE-2012-0287]. A special kind of relationship is the composition of software components, e.g., in the case of Java libraries. Today, a vendor of an application that embeds a vulnerable library cannot encode such information in a standard machine-readable format, as CPE is insufficient to detect the use of a given library in an application. For instance, the recent disclosure of the so-called *heartbleed* vulnerability [CVE-2014-0160] in the OpenSSL library, forced vendors to publish lengthy prose security advisories, with no machine-readable counterpart, providing the list of the affected products (e.g., Oracle published a list of more than 300 products [Oracle2014] distinguish vulnerable from non-vulnerable ones).

The XCCDF and OVAL languages, combined together, allow to specify unambiguous executable configuration checks that can be structured in checklist documents that support prioritization, scoring and human-consumable reporting of results. Moreover, several open source as well as proprietary interpreter implementations exist for both languages [Ovaldi; McAfee; OpenSCAP; OpenVAS]. Although they fulfill some important requirements such as (RL6), (RL7) and (RL9), they do not cope well enough with all of them.

With regard to (RL1), it is difficult, sometimes impossible, to write configuration checks for fine-grained system components independently from their computing environment. The reason is that generic OVAL objects from the so-called *independent schema* (e.g., `textfilecontent54_object`) are relative to a machine's file system, which varies from one system to another. In the case of a JEE web application, for instance, the filesystem location of the deployment descriptor (containing its configuration parameters) depends on the Servlet container and may not even be stored on the filesystem, if the web application is, e.g., persisted in a database. The definition of container-

specific objects (e.g., `spwebapplication_object` for Microsoft Sharepoint), on the other hand, restricts the use of checks to dedicated environments. Requirement (RL2) is not fulfilled as OVAL requires the extension of several schemas to support new software components.

With regard to (RL3), (RL4) and (RL5) it is impossible in XCCDF/OVAL to specify a target for checks that look at distributed components, since the execution of a set of OVAL definitions and their tests are meant to be executed on a single machine. Finally, OVAL does not clearly separate the check logic from the retrieval of the actual configuration values (RL8), herewith missing to leverage industry efforts in the area of IT service and application management. A configuration item can be retrieved by several means and potentially from different sources (the actual component, or a configuration store with copies). The mixture of these concerns makes the authoring of checks difficult and error prone, as one cannot focus only on the check logic (e.g., the session configuration of a deployment descriptor), but also needs to care for the retrieval of configuration values (e.g., the identification of the file path depending on installation directories and environment variables).

3.3 Configuration Validation Language

The configuration validation language allows the definition of security configuration checks for collections of potentially distributed software components and addresses the requirements devised in Section 3.1. Since OVAL already fulfills some of these requirements, it is to a good extent based on OVAL concepts. According to SCAP design goals, in fact, OVAL supports standardized, unambiguous, and exchangeable representations of configuration checks (RL6) as well as variables for parametrization (RL7). However, a significant limitation is that OVAL checks (like CPE) work on the granularity of individual machines (computer systems), which hinders their applicability to distributed systems.

This section introduces all the constructs the language and defines the extensions we carried out over the OVAL standard. As such, the presentation will mainly focus on the parts of OVAL which are extended by our proposal, providing a formal description of their abstract syntax. Figure 3.2 shows the main concepts of the configuration validation language. The concepts are organized into three main areas. The Check and Target areas concern the definition of the configuration checks and of the affected software components, respectively, the System area contains elements corresponding to actual configurations and components of a managed domain.

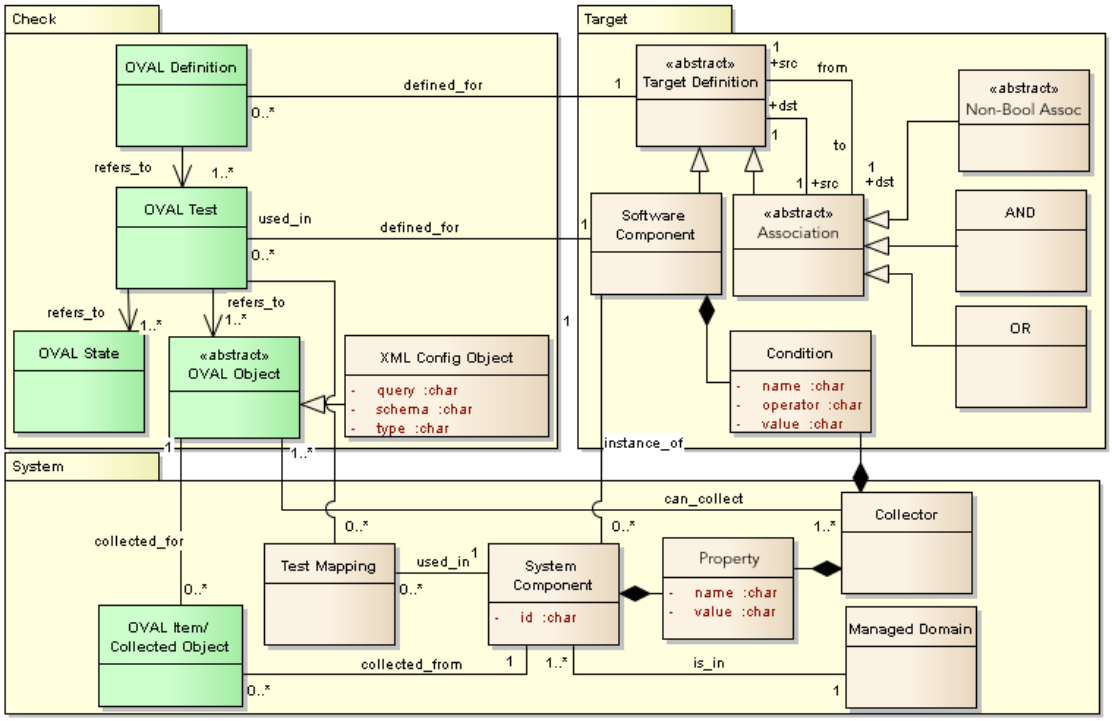


Figure 3.2: Configuration validation language class diagram

3.3.1 Check Area

The *Check* area (top left of Figure 3.2) concerns the definition of checks in the form of tests comparing an expected with an actual configuration value. This area largely relies on the OVAL standard [Baker2012]. The concepts we borrow and extend are shown in Figure 3.2 and prefixed with "OVAL". In a nutshell, an OVAL definition is characterized by a boolean combination of tests and a test defines an evaluation involving an object (possibly containing a set of other objects) and zero or more states.

We introduce a new OVAL object, called `XML Config Object`, that (i) is generic enough to apply to a wide range of configurations of different software components (RL1), (ii) flexibly adapts to arbitrary (XML-based) configuration languages without requiring changes in the interpreter (RL2) and (iii) is independent from the location where configurations are stored (RL8). The `XML Config Object` is characterized by three attributes: `type` denoting a type of configuration relevant for a software component, `schema` denoting the format (i.e., XML grammar) in which the configurations are represented, and `query` expressing how to identify the to-be-checked object within the configuration. This object applies to XML-based configurations, however the same approach can be used to define analogous objects for different common representations (e.g., key-value).

The `XML Config Object` is inspired from the standard `XML File Object` from the OVAL independent schema, which can be analogously used to check any XML-based configuration. However, while the standard object requires the filesystem path of the to-be-checked configuration file, our object is independent from both the location and the mechanism used to store the configuration. To provide these pieces of information we will later introduce dedicated modular language constructs, namely *targets* and *collectors*.

Example 3.2: Object, state, and test for http-only flag

The `XML Config Object` can be used to specify any of the test conditions that are introduced informally in Example 3.1, as they all refer to XML-based configurations.

Consider, for instance, the *http-only flag* test which is part of the SANS *secure cookie* recommendation. In the excerpt below, the type tag (line 2) indicates that the configuration we consider is a web application deployment descriptor, the schema tag (line 3) refers to the location of the schema that defines the syntax of deployment descriptors and the Xpath query (line 4) points to the value of the `http-only` tag.

```
1 <xmlconfiguration_object id="oval:sans.security:obj:1">
2   <type>deployment descriptor</type>
3   <schema>http://java.sun.com/xml/ns/j2ee</schema>
4   <query>//session-config/cookie-config/http-only/text()</query>
5 </xmlconfiguration_object>
6 <xmlconfiguration_state id="oval:sans.security:ste:1">
7   <value_of operation="equals" entity_check="all">true</value_of>
8 </xmlconfiguration_state>
9 <xmlconfiguration_test id="oval:sans.security:tst:1">
10  <object object_ref="oval:sans.security:obj:1" />
11  <state state_ref="oval:sans.security:ste:1" />
12 </xmlconfiguration_test>
```

By creating a new object and modifying only the query element, the *secure flag* and *session timeout* recommendations mentioned in Example 3.1 can be specified as well. Moreover, by also modifying the type and schema, this object can be used for any other XML based configuration.

The expected value for the configuration is specified in an OVAL state of type `xmlconfiguration_state` stating that `true` is the expected value for the `http-only` tag (line 7). This state is coupled with its corresponding object within the OVAL test `xmlconfiguration_test` (lines 9–12).

OVAL definitions are boolean expressions where the atoms are OVAL tests. As we do not change the evaluation semantics of OVAL definitions (i.e., the computation of OVAL results), we do not need to provide here a formal description of their internal structure. In the remainder of this section we will, however, need to refer to OVAL tests and definitions in order to associate them with their corresponding target systems

within a distributed environment. Hence, in the following definition, we only model the relation that associates OVAL tests with the OVAL definition(s) they appear in.

Definition 1 (OVAL Test and Definition)**.** *Let $\mathcal{T}$ denote the domain of all possible OVAL tests and $\mathcal{D}$ that of all OVAL definitions. The finite relation $OD \subseteq \mathcal{D} \times \mathcal{T}$ associates definitions with the tests they are composed of.*

For each OVAL definition $d \in \mathcal{D}$ we denote the set of associated OVAL tests as OD_d. Formally $OD_d = \{t \in \mathcal{T} \mid \langle d, t \rangle \in OD\}$.

Example 3.3: OVAL definitions

In this example, we introduce one OVAL definition for each of the informal checks that are described in Example 3.1.

We denote with the symbol $\texttt{sans} \in \mathcal{D}$ the OVAL definition that checks for the SANS recommendations. It comprises one OVAL test for each recommendation, i.e, $OD_{\texttt{sans}} = \{t_{http-only}, t_{secure-flag}, t_{session}\}$. Similarly, the $\texttt{acme}_\texttt{A}$ definition, checking the compliance with customer A's access control policy, has one test for authentication and one for authorization: $OD_{\texttt{acme}_\texttt{A}} = \{t_{authc}, t_{authz}\}$. The following excerpt exemplifies how such definitions are expressed according to the OVAL syntax.

```
1 <definition id="oval:sans.security:def:1" class="compliance">
2   <criteria operator="AND">
3    <criterion test_ref="oval:sans.security:tst:1" comment="HttpOnly
      flag"/>
4    <criterion test_ref="oval:sans.security:tst:2" comment="Secure flag
      "/>
5    <criterion test_ref="oval:sans.security:tst:3" comment="Session
      timeout"/>
6   </criteria>
7 </definition>
8 <definition id="oval:acme.cust-a.ac:def:1" class="compliance">
9   <criteria operator="AND">
10   <criterion test_ref="oval:acme.cust-a.ac:tst:1" comment="ACME
      customer A authentication"/>
11   <criterion test_ref="oval:acme.cust-a.ac:tst:2" comment="ACME
      customer A authorization"/>
12  </criteria>
13 </definition>
```

According to OVAL, a definition is a boolean combination of tests. As SANS requires all recommendations to be followed, all the tests results are put in AND with each other in order to raise an alarm whenever any of the recommendations is not followed (lines 1–7). Likewise, the access control compliance check for customer A requires both authentication and authorization settings to comply with the policy (lines 8–13).

Name	Description
`product`	Product name, e.g., Struts
`vendor`	Product vendor, e.g., Apache
`release`	Product release, e.g., 2.3.1.1
`sup_spec`	Supported specification
`req_spec`	Required specification
`unc_path`	UNC path for shared location
`ctx_root`	JEE web application context root
`ip_jmx`	IP address of JMX endpoint
`port_jmx`	Port number of JMX endpoint

Table 3.3: Example Software Component Properties

The $t_{http-only}$ test (line 3) is described in Example 3.2. All other tests can be analogously defined.

3.3.2 Target Area

The *Target* area (top right of Figure 3.2) allows to specify targets for configuration checks. A *target definition* is an abstract concept representing either a software component or an association between a pair of either software components or other target definitions. A *software component* is characterized by a set of conditions on specific properties of deployed software instances such as those listed in Table 3.3. An *association* defines a relationship between software components. We distinguish three kinds of associations. Static associations, i.e., "composed of", which allow to represent the internal structure of a software. Run-time associations, i.e., "deployed in" and "communicates with", which allow to define relations among software components running in a landscape. Finally, boolean associations (*and*, *or*) combine either static or dynamic associations. Dynamic and boolean associations can be nested whereas the static ones can only be applied to software components. These types of associations, combined with the possibility to nest them, allow to specify arbitrarily complex target definition expressions. An example target definition may, for instance, specify that a given software component communicates with a second component which is in turn deployed into a third one.

The above description is formalized by the following definitions.

Definition 2 (Condition and Software Component)**.** *A software component is a symbol that identifies a set of conditions. Given sets $\mathcal{P}$ and $\mathcal{V}$ denoting respectively the domain of all properties (cf. Table 3.3) and that of all constant values that can be taken by such properties, a condition is a triple $C = \langle p, \theta, v \rangle$, where:*

Name	Description
$\wedge$	*And*: boolean conjunction
$\vee$	*Or*: boolean disjunction
`depl_in`	*Deployed in*: models a component installed in another
`comp_of`	*Composed of*: represents the internal structure of applications (e.g. linked libraries)
`comm_with`	*Communicates with*: represents network communication
`instr_set`	*Instruction set*: for either compiled (x86, x64) or interpreted (Java Runtime) binaries

Table 3.4: Example Software Component Associations

- $p \in \mathcal{P}$ *is a property,*
- $\theta \in \{=, <, >, \geq, \leq\}$ *is a comparison operator,*
- $v \in \mathcal{V}$ *is a value for the property.*

Let $\mathcal{C}$ denote the domain of all possible conditions and $\mathcal{S}$ that of all software components. The finite relation $SC \subseteq \mathcal{S} \times \mathcal{C}$ associates software components with the conditions they identify.

For each software components $s \in \mathcal{S}$ we denote the set of associated conditions as SC_s. Formally $SC_s = \{C \in \mathcal{C} \mid \langle s, C\rangle \in SC\}$.

A target definition is either a software component or a pair of target definitions related by an association. As formally stated in the next definition, this corresponds to a binary tree where internal nodes are labeled by associations and leaf nodes by software components.

Definition 3 (Association and Target Definition)**.** *Let $\mathcal{A}$ be the set of all possible associations among software components. Some examples of such associations are listed in Table 3.4.*

Let $\mathcal{TD}$ be the set of all target definitions, then:

1. *If $s \in \mathcal{S}$ is a software component, then $\langle\langle\rangle, s, \langle\rangle\rangle \in \mathcal{TD}$ is a target definition;*
2. *If $TD_l, TD_r \in \mathcal{TD}$ are target definitions and $a \in \mathcal{A}$ is an association, then $\langle TD_l, a, TD_r\rangle \in \mathcal{TD}$ is a target definition;*
3. *Nothing else is a target definition.*

The function λ, mapping every target definition to the set of software components it is made of, is inductively defined as follows:

$$\lambda : \mathcal{TD} \to 2^{\mathcal{S}}$$
$$\langle\langle\rangle, s, \langle\rangle\rangle \mapsto \{s\},$$
$$\langle TD_l, a, TD_r\rangle \mapsto \lambda(TD_l) \cup \lambda(TD_r).$$

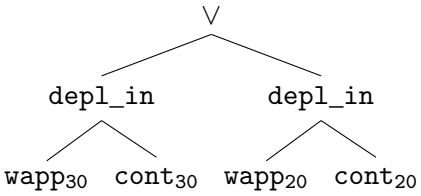


Figure 3.5: Target definition $TD_{\texttt{sans}}$ for the SANS recommendation.

Example 3.4: Software component and target definition for SANS

The SANS recommendation applies to Java Web Applications developed according to one of the releases of the Servlet specification and deployed in a web application container supporting such a specification. In particular some of the recommendations in Example 3.1 are specific to the release 3.0, whereas others apply to previous ones as well.

According to Definition 2, a software component for the web application container can be written as the set containing a single condition referring to the supported specification. For instance software components $\texttt{cont}_{20}$ and $\texttt{cont}_{30}$ refer to web application containers complying with, respectively, releases 2.0 and 3.0:

$$SC_{\texttt{cont}_{20}} = \{\langle \texttt{sup\_spec}, \geq, \texttt{Servlet\_2.0}\rangle\},$$
$$SC_{\texttt{cont}_{30}} = \{\langle \texttt{sup\_spec}, \geq, \texttt{Servlet\_3.0}\rangle\}.$$

As the recommendation applies to all web applications therein deployed, software components $\texttt{wapp}_{20}$ and $\texttt{wapp}_{30}$ for web applications just refer to an empty set of conditions:

$$SC_{\texttt{wapp}_{20}} = SC_{\texttt{wapp}_{30}} = \emptyset.$$

Finally, the target definition for the SANS recommendations, according to Definition 3, is $TD_{\texttt{sans}}$ which combines the above software components with an *or* association, as depicted in Figure 3.5.

We extend the OVAL standard by associating each OVAL definition with a target definition, i.e., a declarative intensional description of its targets (RL3). Such targets are all the collections of distributed instances satisfying the conditions of the software components in the target definition. To allow expressing checks over such distributed targets (RL5) we also associate each OVAL test contained in the OVAL definition to a particular software component appearing in the target definition. We name the resulting new artifact *check definition*. Note that this artifact is not represented by a single class in Figure 3.2 but it involves several of the concepts therein presented and formalized above. Definitions 1 and 3 provide the building blocks for the check definition.

Definition 4 (Check Definition)**.** *A check definition is a tuple* $CD = \langle d, TD, \tau\rangle$ *where*

1. $d \in \mathcal{D}$ *is an OVAL definition,*
2. $TD \in \mathcal{TD}$ *is a target definition,*
3. $\tau : \mathcal{T} \rightarrow \mathcal{S}$ *is a function that maps every OVAL test included in the definition* d *into the software component which it applies to, defined for the target definition* TD*. Hence,* $\tau(t) = s \Rightarrow t \in OD_d \wedge s \in \lambda(TD)$*.*

Example 3.5: Check definition for SANS

Given $OD_{\texttt{sans}}$ and $TD_{\texttt{sans}}$, defined in Examples 3.3 and 3.4 respectively, the check definition for SANS recommendations is

$$CD_{\texttt{sans}} = \langle \texttt{sans}, TD_{\texttt{sans}}, \tau_{\texttt{sans}} \rangle,$$

where the *http only* and *secure flag* tests refer to web applications deployed in containers that support the 3.0 Servlet specification, whereas the *session timeout* test refers to any JEE web application (i.e., from version 2.0 of the Servlet specification, when it was first officially released):

$$\tau_{\texttt{sans}} = \{t_{secure-flag} : \texttt{wapp30}, t_{http-only} : \texttt{wapp30}, t_{session} : \texttt{wapp20}\}.$$

3.3.3 System Area

The *System* area (bottom of Figure 3.2) contains the concepts characterizing systems deployed in a distributed infrastructure and the related concrete configuration checks.

A *system component* represents a single deployment of a software component in a distributed environment. As we aim at checking its configurations, a system component is constituted by an assignment of values to the particular set of properties required to retrieve its configuration through a specific collection mechanism.

Definition 5 (System Component)**.** *A system component* $\sigma : \mathcal{P} \rightarrow_{\perp} \mathcal{V}$ *is a partial mapping from properties to constant values.*

A check definition (Definition 4) associates an oval definition with a target definition, specifying the set of inter-related software components which it applies to. In general, many collections of system components will satisfy the target definition. We associate to each such collections a new construct named *system test*. Analogously to how system components are instances of the software components contained in a target definition, system tests are instances of a check definition. While, in a check definition, OVAL tests are mapped to abstract software components, in a system test they are associated to the corresponding concrete system components.

Definition 6 (System Test)**.** *A system test is a triple* $ST = \langle \Sigma, d, TM \rangle$ *where*

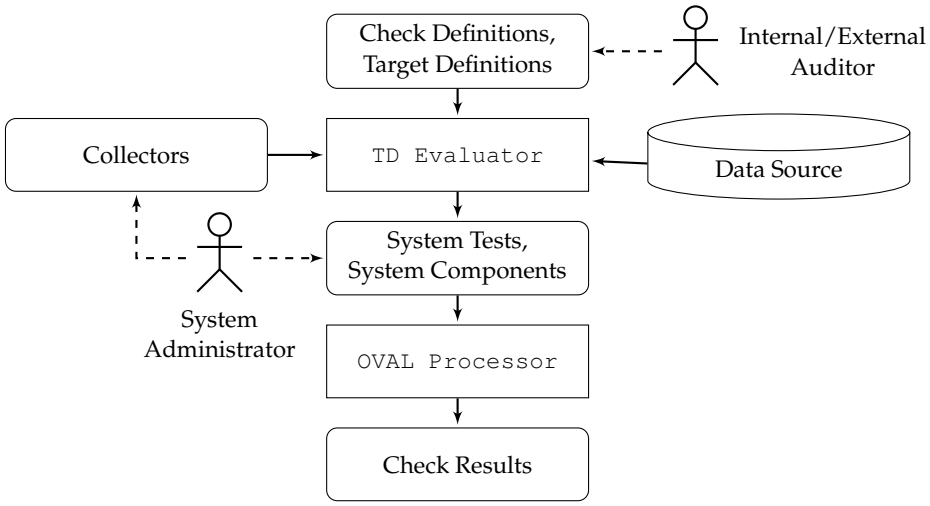


Figure 3.6: Language interpretation flow.

1. Σ *is a set of system components,*
2. *d is an OVAL definition,* OD_d *being its associated tests,*
3. $TM \subseteq \mathcal{T} \times \Sigma$ *is a relation, which we call test mapping, defining which tests of the definition apply to which system components. Hence,* $\langle t, \sigma \rangle \in TM \Rightarrow t \in OD_d \wedge \sigma \in \Sigma$.

Check definitions, respectively target definitions, are system-independent (e.g., referring to any JEE application server), whereas system tests, respectively system components, are system-specific (e.g., referring to a particular instance of Apache Tomcat listening on specific IP address/port). Hence, as outlined in Figure 3.6, the latter can be automatically derived from the former when given a complete description of a system infrastructure. This task is performed by `TD Evaluator` module.

Target definitions represent queries over check targets which can be specified by external and internal authors (from the perspective of an end-user organization), independently from any particular system infrastructure. Embedded in check definitions, they allow to express generic configuration checks (RL3) either for known vulnerabilities affecting software components, as in Scenario (S1), or for best practices of single or multiple software components, as in Scenario (S2). System components are the responses to the target definition queries which are embedded in system tests defining which tests have to be executed on which concrete target. The system tests can also be directly provided by system administrators, thereby bypassing the TD evaluation phase, in case of checks for specific instances of software components (RL4), as in scenario (S3). To produce system tests, the `TD Evaluator` relies on a *data source*: an authoritative source of information about the software components installed in a system infrastructure. The data source is the interpretation structure used to evaluate target

definition queries. As it is not part of the configuration language, but rather of the evaluation procedure implemented by the `TD Evaluator`, its formal definition will be introduced in the next section.

An additional input to the `TD Evaluator` is the set of *collectors*, that specify which properties of system components are necessary to collect the to-be-checked configurations (OVAL objects). Recall that we explicitly left out this piece of information when defining our OVAL object in Section 3.3.1. As such, check authors only need to care for the check logic whereas system administrators, having the knowledge of the system infrastructure, separately configure the available collection mechanisms (RL8). For instance, one collector may specify that for all JEE web applications (software component) it is possible to collect the deployment descriptor (object) by querying a specific Mbean for the web application's context root, through the JMX interface of its application server. Hence, in order to collect the object, one needs to retrieve the values associated to the following properties: (i) the context root, (ii) the IP address of the application server and (iii) the port of the JMX service.

Definition 7 (Collector)**.** *A collector is a tuple $K = \langle CS, PS, O \rangle$ where: $CS \subseteq \mathcal{C}$ is a set of software component conditions, $PS \subseteq \mathcal{P}$ is the set of properties required for collecting an OVAL object, and O is a query selecting the concerned OVAL objects contained in OVAL tests. We assume that a procedure exists to determine whether any OVAL test $t \in \mathcal{T}$ embeds an object matching to O, written $t \models O$[1]. The set of all collectors is $\mathcal{K}$.*

Example 3.6: Collectors, system components and system tests

A collector for web applications deployment descriptors has to define the set of attributes for retrieving the deployment descriptor of the web application installed in the landscape.

Several alternatives are viable, e.g., accessing a shared file system via the Universal Naming Convention (UNC) or relying on the JMX interface of Tomcat. In particular, these two alternatives can be encoded as the two collectors K_{unc} and K_{jmx} with same set of conditions, as they apply to the same software component, and different sets of properties:

$$K_{unc} = \langle \{\langle \texttt{req\_spec}, \geq, \texttt{Servlet\_2.0} \rangle\}, \{\texttt{unc\_path}\}, O_{wapp} \rangle,$$
$$K_{jmx} = \langle \{\langle \texttt{req\_spec}, \geq, \texttt{Servlet\_2.0} \rangle\}, \{\texttt{ctx\_root}, \texttt{ip\_jmx}, \texttt{port\_jmx}\}, O_{wapp} \rangle.$$

The O_{wapp} element is the following Xpath query that determines the applicability of the collector to specific OVAL objects:

```
1 boolean(//xmlconfiguration_object[type="deployment descriptor" and
      schema="http://java.sun.com/xml/ns/j2ee"])
```

[1]For the standard XML serialization of OVAL tests and objects any query expressed in XPath or XQuery fulfills this requirement.

In particular, it matches to all the `XML Config Object`s that refer to a JEE deployment descriptor, represented according to the syntax defined in the XSD schema namespace `http://java.sun.com/xml/ns/j2ee`.

The check definition $CD_{\texttt{sans}} = \langle \texttt{sans}, TD_{\texttt{sans}}, \tau_{\texttt{sans}} \rangle$ defined in Example 3.5 originates several system tests, one for each set of system components installed in the managed domain fulfilling the target definition $TD_{\texttt{sans}}$. Suppose that there are two such sets of system components $\Sigma_a = \{\sigma_{w_a}\}$ and $\Sigma_b = \{\sigma_{w_b}\}$, each one containing a single system component corresponding to an instance of JEE web application that satisfies the target definition $TD_{\texttt{sans}}$. Also, suppose that for the first instance the K_{jmx} collector was applicable, whereas K_{unc} could be used for the second instance (the use of collectors to determine system components' properties is further detailed and exemplified in Section 3.4). Then, the two system components would provide values to different sets of properties, thereby reflecting the different collection mechanisms:

$$\sigma_{w_a} = \{\texttt{ctx\_root} : \texttt{cust} - \texttt{a}, \texttt{ip\_jmx} : 1.1.1.129, \texttt{port\_jmx} : 8059\},$$
$$\sigma_{w_b} = \{\texttt{unc\_path} : \backslash\backslash 1.1.1.130\backslash \texttt{path}\backslash \texttt{to}\backslash \texttt{web.xml}\}.$$

Recall that (Example 3.3) the OVAL definition `sans` refers to the tests $OD_{\texttt{sans}} = \{t_{http-only}, t_{secure-flag}, t_{session}\}$. The corresponding system tests for, respectively, Σ_a and Σ_b are then the following:

$$ST_a = \langle \{\sigma_{w_a}\}, \texttt{sans}, \{\langle t_{http-only}, \sigma_{w_a}\rangle, \langle t_{secure-flag}, \sigma_{w_a}\rangle, \langle t_{session}, \sigma_{w_a}\rangle\}\rangle,$$
$$ST_b = \langle \{\sigma_{w_b}\}, \texttt{sans}, \{\langle t_{session}, \sigma_{w_b}\rangle\}\rangle.$$

Note that no system components for the web application containers (which are mentioned in the target definition $TD_{\texttt{sans}}$) are included in the system tests, as no test applies to them in the check definition (cf. $\tau_{\texttt{sans}}$ defined in Example 3.5).

Consider now the OVAL definition $\texttt{acme}_{\texttt{A}}$, which references the set of tests $OD_{\texttt{acme}_{\texttt{A}}} = \{t_{authc}, t_{authz}\}$. The fact that these tests are specific to customer A's instance of the DEx web application can be expressed by the system test:

$$ST'_a = \langle \{\sigma_{t_a}, \sigma_{w_a}\}, \texttt{acme}_{\texttt{A}}, \{\langle t_{authc}, \sigma_{t_a}\rangle, \langle t_{authz}, \sigma_{w_a}\rangle\}\rangle,$$

where σ_{t_a} denotes the system component corresponding to customer A's tomcat instance from which the authentication configuration has to be retrieved, to be then checked by the test t_{authc}.

System tests are finally processed by the `OVAL Processor` module that interprets the OVAL definitions therein contained and collects the objects defined for each system component. The collected objects are named *OVAL item*s. By comparing such items with the expected state, according to the test, a boolean *check result* is produced. Differently from the OVAL standard, our items may derive from different systems, however this does not affect the evaluation algorithm defined in the standard and which we rely on to compute check results. Each check result is the outcome of a single system test. As

more system tests can be originated from a check definition, a single check definition yields, in general, a set of check results. In the next section we describe how system tests are instantiated from check definitions.

3.4 Language Interpretation

A key step of the workflow in Figure 3.6 is the generation of the system tests based on the information contained in the data source. The evaluation of system tests is then performed according to the rules already specified by the OVAL standard [Baker2012], which we do not restate here. Instead, in this section, we focus on the first part of the workflow. We first formally define the interpretation of target definitions with respect to a data source, which provides information about the properties of software components deployed within a managed domain, and we then describe how this leads to the generation of system tests.

Informally, a data source can be seen as a particular instantiation of software component properties (cf. Definition 2) and target definition associations (cf. Definition 3) for a managed domain. Specifically, we hereby restrict to the set of properties (respectively associations) reported in Table 3.3 (respectively Table 3.4). We assume a single data source to provide information about several aspects of the managed domain, ranging from the properties of installed software (e.g. product names and vendors), or the internal structure of applications (e.g. linked libraries), up to architectural details on the deployment or the network interaction among different pieces of software. Since such information is often scattered over several repositories within an organization (e.g., configuration management databases, dependency management systems), the data source is a federated set of views over these repositories, which constitute the interface to our language.

Let $\mathcal{I}$ be the domain of instances of software components, namely software component identifiers, containing one unique symbol for each software component installed in a given managed domain. The data source then maps every software component identifier to the actual values of its properties and links it to the other software component identifiers it is associated with.

Definition 8 (Data Source). *A data source is the pair $DS = \langle \pi, \alpha \rangle$ where:*

- *the partial function $\pi : \mathcal{P} \times \mathcal{I} \rightarrow_{\perp} \mathcal{V}$, assigns values to the properties of software component identifiers. By $\pi_p : \mathcal{I} \rightarrow_{\perp} \mathcal{V}$ we denote its currying for a property $p \in \mathcal{P}$;*
- *the function $\alpha : \mathcal{A} \rightarrow 2^{\mathcal{I} \times \mathcal{I}}$ maps each association $a \in \mathcal{A}$ to the relation $\alpha_a \subseteq \mathcal{I} \times \mathcal{I}$.*

i	$\pi_{\texttt{vendor}}$	$\pi_{\texttt{product}}$	$\pi_{\texttt{release}}$	$\pi_{\texttt{sup\_spec}}$	$\pi_{\texttt{unc\_path}}$
a	Apache	HTTPd	2.4.7	$\perp$	$\perp$
l	OpenLDAP	OpenLDAP	2.4.30	$\perp$	$\perp$
t_1	Apache	Tomcat	7.0.18	Servlet3.0	$\perp$
t_2	Apache	Tomcat	6.0.25	Servlet2.5	$\perp$
w_a	ACME	DEx	1.0	$\perp$	$\perp$
w_b	ACME	DEx	1.0	$\perp$	\\1.1.1.130\path\ to\cust-b\web.xml
w_c	ACME	Dex	1.0	$\perp$	\\1.1.1.130\path\ to\cust-c\web.xml

(a) Instance of π for properties {vendor, product, release, sup_spec, unc_path}.

i	$\pi_{\texttt{ctx\_root}}$	$\pi_{\texttt{ip\_jmx}}$	$\pi_{\texttt{port\_jmx}}$
a	$\perp$	$\perp$	$\perp$
l	$\perp$	$\perp$	$\perp$
t_1	$\perp$	$\perp$	$\perp$
t_2	$\perp$	$\perp$	$\perp$
w_a	cust-a	1.1.1.129	8059
w_b	cust-b	$\perp$	$\perp$
w_c	cust-c	$\perp$	$\perp$

(b) Instance of π for properties {ctx_root, ip_jmx, port_jmx}.

$\alpha_{\texttt{comm\_with}}$		$\alpha_{\texttt{depl\_in}}$	
a	t_1	w_a	t_1
a	t_2	w_b	t_2
t_1	l	w_c	t_2
t_2	l		

(c) Instance of α for associations comm_with and depl_in.

Figure 3.7: Example data source instance for ACME.

Example 3.7: Data source

Figure 3.7 depicts a tabular representation of the data source $DS_{\texttt{acme}}$ for the example ACME system infrastructure. For the sake of conciseness, only a subset of the properties listed in Table 3.3 and associations of Table 3.4 are considered.

The software component identifiers t_1 and t_2 correspond respectively to the Tomcat instances running on machines 1.1.1.129 and 1.1.1.130, whereas a denotes the Apache reverse proxy in the DMZ and l the OpenLDAP server storing the user accounts. Finally, w_a, w_b and w_c correspond to the instances of the DEx web application dedicated to customers A, B and C respectively.

Note that the version of the Tomcat instance t_2 is significantly older than that of t_1, which is also reflected in that t_2 supports and older release of the JEE Servlet specification.

3.4.1 Target definition interpretation.

The conditions associated to a software component can be seen as a simple conjunctive query ranging over properties of software deployed within a managed domain. The

data source provides the necessary views on the managed domain to answer such a query. The answer consists of the set of software component identifiers matching to a set of software componentconditions. If it has no conditions, the answer is the entire domain of software component identifiers $\mathcal{I}$. This evaluation is performed by the data source interpretation of software components, given by the mapping $\lceil\cdot\rceil_{DS} : 2^{\mathcal{C}} \to 2^{\mathcal{I}}$:

$$\begin{aligned} \lceil\emptyset\rceil_{DS} &= \mathcal{I} \\ \lceil\{\langle p, \theta, v\rangle\} \cup X\rceil_{DS} &= \{i \in \mathcal{I} \mid \pi_p(i)\ \theta\ v\} \cap \lceil X\rceil_{DS}. \end{aligned} \tag{3.1}$$

A target definition $TD \in \mathcal{TD}$ is instead a more complex selection predicate (cf. Definition 3) and there can be several sets of software component identifiers which satisfy it. The interpretation of TD over a data source DS , $[\![\cdot]\!]_{DS} : \mathcal{TD} \to 2^{2^{\mathcal{I}}}$, provides all such sets for every target definition, as defined in (3.2).

$$\begin{aligned} [\![\langle\langle\rangle, s, \langle\rangle\rangle]\!]_{DS} &= \{\{x\} \mid x \in \lceil SC_s\rceil_{DS}\} \\ [\![\langle TD_l, a, TD_r\rangle]\!]_{DS} &= \begin{cases} [\![TD_l]\!]_{DS} \cup [\![TD_r]\!]_{DS} & \text{if } a = \vee \\ \{X \cup Y \mid X \in [\![TD_l]\!]_{DS}, Y \in [\![TD_r]\!]_{DS}\} & \text{if } a = \wedge \\ \{X \cup Y \mid X \in [\![TD_l]\!]_{DS}, Y \in [\![TD_r]\!]_{DS}, X \times Y \subseteq \alpha_a\} & \text{otherwise} \end{cases} \end{aligned} \tag{3.2}$$

If the target definition is a simple software component s, then the function returns the result of applying (3.1) to the corresponding set of conditions SC_s. Otherwise, the target definition is an association between two target definitions, which are first interpreted recursively. The sub-results are then combined differently depending on the association:

- $\vee$ (boolean disjunction) yields the union of sub-results, as all the sets from either sub-result have to be considered;
- $\wedge$ (boolean conjunction) yields the set made by the pairwise union of all the sets found in the respective sub-results;
- any other association a behaves like $\wedge$, except the pairs of sets are filtered by retaining only those for which all elements of one set are associated with those of the other set by the relation α_a in the data source.

Note that both (3.1) and (3.2) depend on the data source DS as it assigns values to instance properties and associations.

Similarly, according to (3.3), the interpretation function $[\![\cdot]\!]_{DS} : \mathcal{TD} \to 2^{\mathcal{I}\times\mathcal{S}}$ maps every target definition to a relation $IS \subseteq \mathcal{I} \times \mathcal{S}$ associating each software component to all the software component identifiers that instantiate it (note that different software components may instantiate the same software component identifier).

$$\begin{aligned} [\![\langle\langle\rangle, s, \langle\rangle\rangle]\!]_{DS} &= \{\langle i, s\rangle \mid i \in \lceil SC_s\rceil_{DS}\}, \\ [\![\langle TD_l, a, TD_r\rangle]\!]_{DS} &= [\![TD_l]\!]_{DS} \cup [\![TD_r]\!]_{DS}. \end{aligned} \tag{3.3}$$

We are now in a position to define the evaluation function $[\![\cdot]\!]_{DS}$ of a target definition $TD \in \mathcal{TD}$ over the data source DS, that maps it to the pair $\langle I^*, IS \rangle$, where I^* is a powerset of software component identifiers and IS a relation associating every $i \in I \in I^*$ to software components $s \in \lambda(TD)$. As expressed in (3.4), the definition of $[\![\cdot]\!]_{DS}$ relies on the aforementioned recursive interpretation functions of all the elements within the target definition expression.

$$[\![TD]\!]_{DS} = \langle [\![TD]\!]_{DS}, [\![TD]\!]_{DS} \rangle. \tag{3.4}$$

Example 3.8: TD interpretation

In this example we compute the interpretation of the target definition $TD_{\texttt{sans}}$, introduced in Example 3.4, with respect to the data source $DS_{\texttt{acme}}$, shown in Figure 3.7 (Example 3.7).

According to (3.4), we need to determine

$$[\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}} = \langle I^*_{\texttt{sans}}, IS_{\texttt{sans}} \rangle = \langle [\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}}, [\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}} \rangle.$$

In order to obtain $[\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}}$, according to (3.2), we start from the base cases, i.e., the terms involving software components $\texttt{wapp}_{20}, \texttt{wapp}_{30}, \texttt{cont}_{20}$ and $\texttt{cont}_{30}$. Being software components with empty sets of conditions, the first two terms instantiate all the elements in $\mathcal{I}$: $[\![TD_{\texttt{wapp}_{20}}]\!]_{DS_{\texttt{acme}}} = [\![\langle\langle\rangle, \texttt{wapp}_{20}, \langle\rangle\rangle]\!]_{DS_{\texttt{acme}}} = \{\{i\} \mid i \in \lceil SC_{\texttt{wapp}_{20}} \rceil_{DS_{\texttt{acme}}} = \lceil \emptyset \rceil_{DS_{\texttt{acme}}}\} = \{\{i\} \mid i \in \mathcal{I}\} = \{\{w_a\}, \{w_b\}, \{w_c\}, \{t_1\}, \ldots\} = [\![TD_{\texttt{wapp}_{30}}]\!]_{DS_{\texttt{acme}}}$. The third, respectively fourth, term yields instead only the JEE containers supporting version 2.0, respectively 3.0, of the Servlet specification: $[\![TD_{\texttt{cont}_{20}}]\!]_{DS_{\texttt{acme}}} = \{\{t_1\}, \{t_2\}\}$ and $[\![TD_{\texttt{cont}_{30}}]\!]_{DS_{\texttt{acme}}} = \{\{t_1\}\}$.

The first and third terms are combined through the `depl_in` association yielding

$$\begin{aligned}
&[\![TD_{20}]\!]_{DS_{\texttt{acme}}} = [\![\langle TD_{\texttt{wapp}_{20}}, \texttt{depl_in}, TD_{\texttt{cont}_{20}} \rangle]\!]_{DS_{\texttt{acme}}} \\
&= \{X \cup Y \mid X \in [\![TD_{\texttt{wapp}_{20}}]\!]_{DS_{\texttt{acme}}}, Y \in [\![TD_{\texttt{cont}_{20}}]\!]_{DS_{\texttt{acme}}}, X \times Y \subseteq \alpha_{\texttt{depl_in}}\} \\
&= \{\{w_a, t_1\}, \{w_b, t_2\}, \{w_c, t_2\}\}
\end{aligned}$$

and, by analogous reasoning, the second and fourth terms produce $[\![TD_{30}]\!]_{DS_{\texttt{acme}}} = [\![\langle TD_{\texttt{wapp}_{30}}, \texttt{depl\_in}, TD_{\texttt{cont}_{30}} \rangle]\!]_{DS_{\texttt{acme}}} = \{\{w_a, t_1\}\}$.

Finally, the last two partial results are combined in the interpretation of the topmost term $I^*_{\texttt{sans}} = [\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}} = [\![\langle TD_{20}, \vee, TD_{30} \rangle]\!]_{DS_{\texttt{acme}}} = [\![TD_{20}]\!]_{DS_{\texttt{acme}}} \cup [\![TD_{30}]\!]_{DS_{\texttt{acme}}} = \{\{w_a, t_1\}, \{w_b, t_2\}, \{w_c, t_2\}\}$. Figure 3.8 depicts the results of all recursive steps on the tree structure of the $TD_{\texttt{sans}}$ expression.

Similarly, by applying (3.3), we obtain $IS_{\texttt{sans}} = [\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}} = \{\langle w_a, \texttt{wapp}_{30} \rangle, \langle w_a, \texttt{wapp}_{20} \rangle, \langle w_b, \texttt{wapp}_{20} \rangle, \langle w_c, \texttt{wapp}_{20} \rangle, \langle t_1, \texttt{cont}_{30} \rangle, \langle t_1, \texttt{cont}_{20} \rangle, \langle t_2, \texttt{cont}_{20} \rangle\}$.

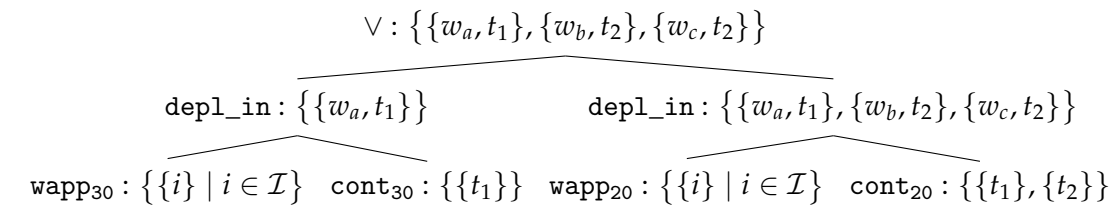


Figure 3.8: Recursive computation of $[\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}}$.

3.4.2 Generation of system tests.

As last step, the `TD Evaluator` needs to identify one or more system tests, mapping each OVAL test to the system component carrying the information about how to collect the object.

A check definition $CD = \langle d, TD, \tau \rangle$ is defined for the target definition TD, being interpreted over a data source resulting in a pair $[\![TD]\!]_{DS} = \langle I^*, IS \rangle$. Every $I \in I^*$ is a set of software component identifiers satisfying the TD expression. Therefore one system test has to be created for every such set I. When the `TD Evaluator` processes a check definition, it must identify a *matching collector* K, among the set $\mathcal{K}$ of all the ones defined for a given managed domain. This has to be done for every software component identifier $i \in I$, as the collector provides the set of properties that represent the information needed to collect the to-be-checked configurations for specific OVAL objects from i. For this reason, every $K \in \mathcal{K}$ (cf. Definition 7) contains a set of conditions CS to identify matching software component identifier and a query O, matching to the OVAL objects it applies to. The conditions required to determine whether a collector matches to a software component identifier are given by the following definition.

Definition 9 (Matching Collector)**.** *For a check definition $CD = \langle d, TD, \tau \rangle$, let $[\![TD]\!]_{DS} = \langle I^*, IS \rangle$ be an interpretation of TD over DS.*

We then say that a collector $K = \langle CS, PS, O \rangle$ matches to the software component identifier $i \in I \in I^*$, iff the following three conditions hold

1. *$i \in \lceil CS \rceil_{DS}$, i.e., i is indeed an instance matching all collector's conditions CS;*
2. *$\forall p \in PS, \pi_p(i) \neq \bot$, i.e., all the properties required by the collector are defined for i in the datasource;*
3. *$\forall t \in OD_d$ s.t. $\langle i, \tau(t) \rangle \in IS, t \models O$, i.e., all the OVAL objects to be collected from i match the collector's Xpath query.*

We now have all the ingredients that are necessary to derive the system tests that have to be executed in order to check a given check definition. This is expressed by the algorithm presented in Figure 3.9. Given the interpretation $[\![TD]\!]_{DS} = \langle I^*, IS \rangle$ of a target definition within a check definition $CD = \langle d, TD, \tau \rangle$, we associate each $I \in I^*$ to a

Input: Check definition $CD = \langle d, TD, \tau \rangle$
Input: Data Source $DS = \langle \pi, \alpha \rangle$, with $\pi : \mathcal{P} \times \mathcal{I} \to_{\perp} \mathcal{V}$ and $\alpha : \mathcal{A} \to 2^{\mathcal{I} \times \mathcal{I}}$
Output: Set of system tests STS

1 $STS \leftarrow \emptyset$;
2 $\langle I^*, IS \rangle \leftarrow [\![TD]\!]_{DS}$;
3 **foreach** $I \in I^*$ **do**
4 $\quad \Sigma_I \leftarrow \emptyset$;
5 $\quad TM_I \leftarrow \emptyset$;
6 $\quad$ **foreach** $i \in I$ **do**
7 $\quad\quad$ Choose $K = \langle CS, PS, O \rangle \in \mathcal{K}$ s.t. K matches to i;
8 $\quad\quad \sigma_i \leftarrow \left(p \mapsto \begin{cases} \pi(p, i) & \text{if } p \in PS \\ \perp & \text{otherwise.} \end{cases} \right)$;
9 $\quad\quad T_i \leftarrow \{ t \in OD_d \mid \langle i, \tau(t) \rangle \in IS \}$;
10 $\quad\quad \Sigma_I \leftarrow \Sigma_I \cup \{\sigma_i\}$;
11 $\quad\quad TM_I \leftarrow TM_I \cup \{ \langle t, \sigma_i \rangle \mid t \in T_i \}$;
12 $\quad STS \leftarrow STS \cup \{ \langle \Sigma_I, d, TM_I \rangle \}$
13 **return** STS;

Figure 3.9: Evaluation algorithm: from check definition to system tests.

system test $\langle \Sigma_I, d, TM_I \rangle$. Every element $\sigma_i \in \Sigma_I$ is a system component, i.e., a mapping assigning values to properties of the software component identifier i that will allow to collect configuration information from it. The relation TM_I associates instead every test $t \in OD_d$ to one or more system components $\sigma_i \in \Sigma_I$. To build the set Σ_I we make use of the collectors. For every $i \in I$ we first (line 7) look for a matching collector K, according to Definition 9, that carries a set of properties PS. These properties specify what information is necessary in order to collect the configuration data from the software component identifier i (e.g., IP address and port of the JMX service). We then fetch the values of such properties from the data source DS to obtain the system component σ_i (line 8). To know which tests are to be executed on which system components, we track back all the tests that were mapped, in the check definition, to the software component that instantiated i (line 9) and we associate each of them with σ_i in the TM_I relation (line 11).

Example 3.9: System tests generation

Let us consider the check definition $CD_{\texttt{sans}} = \langle \texttt{sans}, TD_{\texttt{sans}}, \tau_{\texttt{sans}} \rangle$, introduced in Example 3.5, and the data source interpretation of its target definition $[\![TD_{\texttt{sans}}]\!]_{DS_{\texttt{acme}}} = \langle I^*_{\texttt{sans}}, IS_{\texttt{sans}} \rangle$, which has been derived in Example 3.8.

Three sets of software component identifiers satisfy the target definition, namely $I^*_{\texttt{sans}} = \{I_a = \{w_a, t_1\}, I_b = \{w_b, t_2\}, I_c = \{w_c, t_2\}\}$, hence three system tests will be created. Among those, we shall only discuss, for brevity, the system tests ST_{I_a} and ST_{I_b}, related to I_a and I_b respectively, as I_c is analogous to I_b.

According to Definition 9 the collector K_{jmx} matches to the software component identifier w_a (and not to w_b), as

1. $w_a \in \lceil\{\langle \texttt{req\_spec}, \geq, \texttt{Servlet\_2.0}\rangle\}\rceil_{DS'_{\texttt{acme}}}$,
2. $\pi_{\texttt{ctx\_root}}(w_a)$, $\pi_{\texttt{port\_jmx}}(w_a)$, $\pi_{\texttt{port\_jmx}}(w_a)$ are all defined in $DS'_{\texttt{acme}}$ (while this is not the case for w_b), and
3. it is true that $t_{http-only} \models O_{wapp}$, $t_{secure-flag} \models O_{wapp}$ and $t_{session} \models O_{wapp}$.

From analogous reasoning it follows that K_{unc} matches to w_b (and not to w_a).

By applying Algorithm 3.9 we finally derive, as anticipated in Example 3.6, that:

$$ST_a = \langle\{\sigma_{w_a}\}, \texttt{sans}, \{\langle t_{http-only}, \sigma_{w_a}\rangle, \langle t_{secure-flag}, \sigma_{w_a}\rangle, \langle t_{session}, \sigma_{w_a}\rangle\}\rangle,$$
$$ST_b = \langle\{\sigma_{w_b}\}, \texttt{sans}, \{\langle t_{session}, \sigma_{w_b}\rangle\}\rangle.$$

Note, in particular, that not all the tests of the OVAL definition are included in the test mapping of ST_b. This is a consequence of the fact that $\langle w_b, \texttt{wapp}_{30}\rangle \notin IS_{\texttt{sans}}$, therefore all the tests that are specified for the release 3.0 of the Servlet specification (i.e., all $t \in OD_{\texttt{sans}}$ such that $\tau_{\texttt{sans}}(t) = \texttt{wapp}_{30}$) are excluded from the test mapping (lines 9 and 11 of Algorithm 3.9).

3.5 Implementation

This section introduces COAS (Configuration Assessment as a Service): a prototype for the automated validation of configuration settings in distributed information systems. The tool implements the language and approach defined in Sections 3.3 and 3.4.

3.5.1 Language Implementation

Section 3.3 introduces the configuration validation language without providing a concrete syntax. As we now aim at implementing configuration validation, we need to bind the language abstract structures to concrete machine-readable constructs that can be authored by users and interpreted by a tool. Being the configuration language based on OVAL, which is an XML language, we chose to use an XML representation too.

The first extension we proposed concerns the definition of the new OVAL `XML Config Object` which customizes a base OVAL object according to the extensibility rules of the OVAL standard. The XML schema of this object is included in Appendix A.1 and its use has already been shown in Example 3.2. The second extension is the in-

troduction of the check definitions, which combine standard OVAL definitions with information about the software components they apply to, i.e., the target definition. Incorporating these additional concepts in OVAL would require forbidden modifications to the OVAL schemas. In order to maximize the compatibility with the standard, we chose to avoid this option and to specify check and target definitions in independent XML documents whose grammar is reported in Appendix A.2.

Example 3.10: Concrete check and target definitions for SANS

The following snippet reports the XML representation of the target and check definitions for the SANS recommendations, specified in Examples 3.4 and 3.5 respectively.

```
1 <target_definition id="td:sans.security:def:1">
2   <association name="or">
3     <association name="deployed_in">
4       <software_component id="sc:wapp20" />
5       <software_component id="sc:cont20">
6         <condition property="supported_specification" operator="
      greater_eq" value="Servlet_2.0" />
7       </software_component>
8     </association>
9     <association name="deployed_in">
10      <software_component id="sc:wapp30" />
11      <software_component id="sc:cont30">
12        <condition property="supported_specification" operator="
      greater_eq" value="Servlet_3.0" />
13      </software_component>
14    </association>
15  </association>
16 </target_definition>
17 <check_definition id="cd:sans.security:check:1"
18                   od_ref="oval:sans.security:def:1"
19                   td_ref="td:sans.security:def:1">
20   <target_mapping sc_ref="sc:wapp30">
21     <test test_ref="oval:sans.security:tst:1" comment="HttpOnly flag"
      />
22     <test test_ref="oval:sans.security:tst:2" comment="Secure flag" />
23   </target_mapping>
24   <target_mapping sc_ref="sc:wapp20">
25     <test test_ref="oval:sans.security:tst:3" comment="Session timeout
      " />
26   </target_mapping>
27 </check_definition>
```

Lines 1 to 16 represent the target definition $TD_{\texttt{sans}}$. The check definition $CD_{\texttt{sans}}$ is instead encoded in lines 17 to 27. Here (line 18), the OVAL definition for SANS is referenced (cf. Example 3.3), as well as (line 19) the above-specified target definition.

The interpretation of check definitions requires interpreting their target definitions, in order to identify instances of the software components which they apply to. This step requires the specification of a set of collectors $\mathcal{K}$, which are language artifacts specific to a given managed domain that carry information on how to collect the configuration information from the actual systems. The XSD grammar for specifying a set of collectors is defined in Appendix A.3.

Example 3.11: Concrete JMX collector

In Example 3.6 the collector K_{jmx} has been introduced, whose purpose is to collect the deployment descriptor of JEE web applications via the JMX protocol. The following snippet represents its XML representation.

```
1 <collectors>
2   <collector id="oval:jmx:col:1" type="com.sap.coas.collector.spi.jmx.
      j2ee.J2EEJmxCollector">
3     <description>This collector will access JMX</description>
4     <platform>
5       <condition property="supported_specification" operator="
      greater_eq" value="Servlet_2.0" />
6     </platform>
7     <oval_objects>
8       boolean(//xmlconfiguration_object[type="deployment descriptor"
      and schema="http://java.sun.com/xml/ns/j2ee"])
9     </oval_objects>
10    <parameters>
11      <parameter name="jmx.conn.host" />
12      <parameter name="jmx.conn.port" />
13      <parameter name="jmx.j2ee.contextRoot" />
14    </parameters>
15  </collector>
16 </collectors>
```

Lines 4–6 and 7–9 determine the applicability conditions of the collector to, respectively, specific software component instances and OVAL objects. Lines 10–14 encode instead the parameters required by the collector. Note, moreover, the collector's `type` attribute specified in line 2. It points to the Java class implementing the behaviour specific to the collector, which, in this example, corresponds to fetching a web application's deployment descriptor from an application server that supports JMX.

The artifacts resulting from the interpretation of target definitions are system tests which specify the targets of each OVAL test, i.e., which system the to-be-checked configurations shall be collected from. System tests can also be provided directly if the check targets are known. As they map OVAL tests to target systems, their corresponding XML constructs, defined in Appendix A.4, are named `target-mapping`s.

We finally enrich the configuration validation language with the concept of *checklist*.

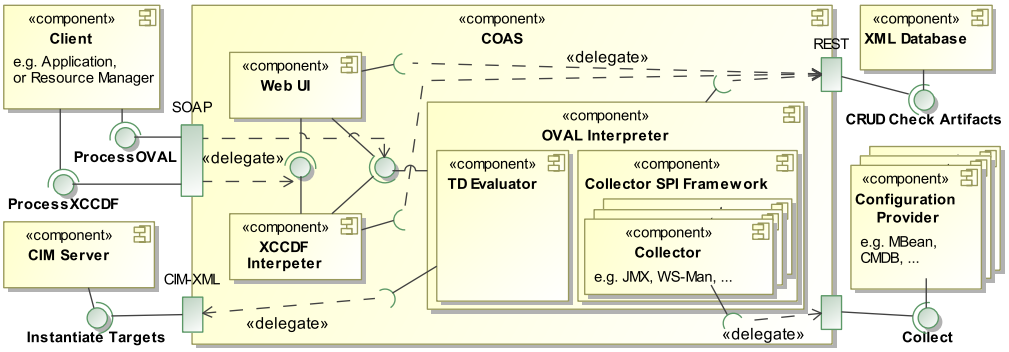


Figure 3.10: COAS Component Diagram

A checklist defines the set of check definitions to be executed and allows to organize them in groups. As groups can be included in groups themselves, a checklist allows to define an arbitrary complex hierarchy of check definitions (RL9). Since this feature is supported by the XCCDF standard (described in Section 3.2) we borrow entirely both syntax and semantics of XCCDF checklists from the standard specification [Ziring2008]. The concept of checklist was not introduced in Section 3.3, where only the extensions we carried out on OVAL, and which constitute the core contribution of this chapter, were discussed. However, as mentioned in requirement (RL9), checklists are important from a practical standpoint, in that they allow users to structure and prioritize checks. Hence, they are introduced here were the implementation and use of the configuration validation tool are discussed.

3.5.2 Tool Implementation

COAS is implemented as a JEE web application organized in different modules, as shown in the component diagram depicted in Figure 3.10. The tool can be consumed as a Web Service or through the COAS Web User Interface. This is represented by the `Client` and `WebUI` components, respectively. As such, configuration validation can be automatically triggered, e.g., periodically or upon some event occurring, or manually executed. The fact that the tool is available as a service is a key feature as it allows the validation of configuration settings of distributed systems within a single check or checklist. Moreover, being application independent, it provides a centralized approach for configuration validation that can be integrated into legacy tools, hereby establishing consistency among tools that are run by different people and at different application lifecycle phases within an organization.

The `XCCDF Interpreter` component implements the parsing of XCCDF checklists as prescribed in [Ziring2008]. The XCCDF rules that have an associated check definition trigger the invocation of the `OVAL Interpreter` component. Every check

definition references both an OVAL definition and a target definition. The former is interpreted according to the standard specification [Baker2012], while the latter is handled by the `TD Evaluator` component.

The `TD Evaluator` implements the algorithm presented in Figure 3.9 (cf. Section 3.4). It processes input target definitions, represented in XML as described previously in this section, and produces output system tests.

The external `CIM Server` component implements the data source, which is required to instantiate target definitions according to a specific IT infrastructure. The Common Information Model (CIM) [DMTF2000] is an information model developed by the Distributed Management Task Force (DMTF) to support the integrated management of large IT infrastructures comprising systems, software, users, networks and more. In order to conveniently explore the content of a CIM instance, the CIM Query Language (CQL) has been defined [DMTF2007]. In a nutshell, the CQL is a subset of SQL-92 with some extensions specific to CIM. The `CIM Server` component represents any software product capable of storing CIM instances and, in particular, that implements a CQL query engine. Several configuration management products fulfill this requirement: both commercial (e.g., BMC Atrium, IBM Tivoli, HP Universal CMDB, SAP Solution Manager) and open source ones (e.g., Open Group OpenPegasus, Sun WBEM Services, IBM SBLIM). Moreover, we assume that the data model of the `CIM Server` contains a *CIM class* named `software_component` having one attribute for each property $p \in \mathcal{P}$ and one *CIM (self-)association* for each association $a \in \mathcal{A}$. As such, the data source structure of Definition 8 is implemented by simple CQL queries. Each π_p maps to a *select* query for instances of the `software_component` *CIM class* projected on p. Each α_a, instead, is a query for instances of the *CIM association* a.

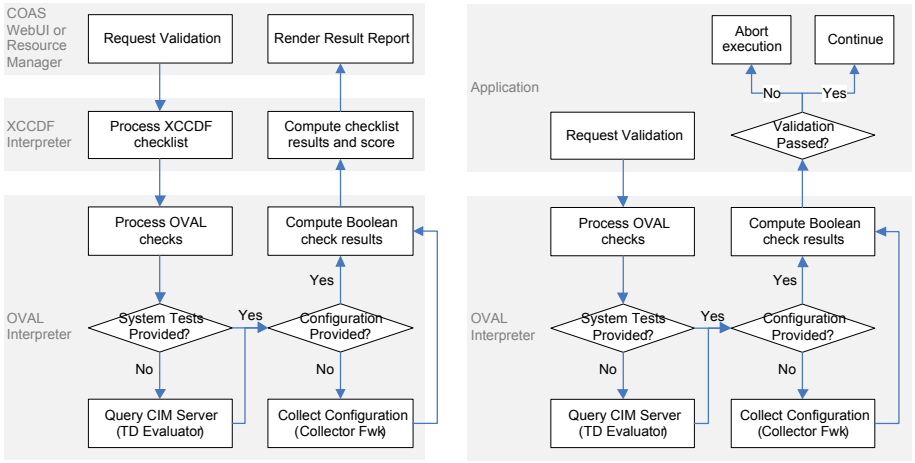
Example 3.12: Data source CQL implementation

The following snippet shows two CQL queries that are issued to the CIM server as part of the interpretation of the target definition $TD_{\texttt{sans}}$ computed in Example 3.8.

```
1 SELECT OBJECTPATH(software_component)
2 FROM software_component SC
3 WHERE SC.sup_spec >= "Servlet_2.0";
4
5 SELECT A.Antecedent, A.Dependent
6 FROM deployed_in A;
```

The first one (lines 1–3) allows to compute the term $[\![TD_{\texttt{wapp}_{20}}]\!]_{DS_{\texttt{acme}}}$. The `OBJECTPATH` operator is a CQL operator that returns the unique identifier of matching instances.

The second query (lines 5–6) is used when filtering for the membership to the relation $\alpha_{\texttt{depl\_in}}$, e.g., when computing the term $[\![TD_{20}]\!]_{DS_{\texttt{acme}}}$. Note that each CIM association



(a) Scenarios (S1) (S2) and (S3) (b) Scenario (S4)

Figure 3.11: COAS workflow per scenario (invariant flow for the OVAL interpreter).

always has two attributes, namely `Antecedent` and `Dependent` that hold the identifiers the endpoint instances (in this example software components).

The `Collector` modules are implementations of the collector Service Provider Interface (SPI), which provides an extensible library of collection mechanisms for configuration data. We implemented a small library of collector modules that leverage common application or operating system remote management protocols, such as JMX, SMB, SSH. New modules could be easily added to deal with specific technologies, e.g., SNMP for network equipment or WS-Man for web services. As anticipated in Example 3.11, every collector module has a corresponding collector specification that determines the software components and OVAL objects it applies to, as well as the parameters that are needed for the collection and whose values are fetched from the data source.

The external `XML database` is used by several COAS components to store and access the variety of XML artifacts (e.g., checklists, checks, results) generated in the course of a COAS execution. For instance, it allows the COAS Web UI to fetch, and render to the user, the results of every past execution of the tool.

Figure 3.11 shows the execution flow of COAS in the context of the different scenarios introduced in Section 3.1. Figure 3.11a shows how COAS addresses scenarios (S1), (S2) and (S3) or, in fact, any combination of the three. In this case COAS receives the request to process an XCCDF checklist including an arbitrary collection of vulnerability, best practice or compliance checks. Figure 3.11b shows the usage of COAS for the scenario (S4). In this case COAS is directly invoked by an application to evaluate a collection of checks at runtime. Depending on the check results, the application may decide to suppress critical functionalities.

In both Figures 3.11a and 3.11b, the flow diagram of the `OVAL interpreter` is the same and it depends on which artifacts were submitted. First the interpreter establishes if system tests are provided. This is the case of policy compliance checks (cf. scenario (S3)), whereby the target systems are known *a priori*. If they are not available (e.g., vulnerability and best-practice checks), COAS retrieves the information of which systems have to be validated. This is done by the `TD Evaluator` that relies on the `CIM Server` as data source. Once the target systems are identified, COAS checks whether the configuration settings have already been provided. This is the case if the client that invoked COAS manages the configuration information itself, like, for instance, a configuration management system or an application that aims at checking its own configuration. If this is not the case, the `Collector` framework is used to retrieve configuration settings from a configuration provider, such as a remote file system, a JMX endpoint or even a CMDB storing replicated configuration items. Finally, the `OVAL interpreter` computes the OVAL results by comparing the collected configuration items with the expected OVAL states according to the OVAL specification. In case of compliance assessment, a true result means that the compliance with the expected state is ensured. In case of vulnerability assessment, a true result value states that a vulnerability is present.

3.6 Related Work

Existing standards and tools for configuration validation comprise several specifications out of the Security Content Automation Protocol (SCAP), namely CPE, XCCDF and OVAL, as well as vulnerability and patch scanners that work on the basis of proprietary languages to express vulnerability checks.

In Section 3.2 we introduced SCAP specifications and we highlighted their shortcomings with regard to expressing configuration checks for distributed systems. Our target definitions are inspired from CPE Names in that we allow to express conditions over properties of software components. However we explicitly allow for expressing associations that carry specific semantics, e.g., `depl_in` to express that one component is installed within another, which is not possible with the CPE Language. Standard OVAL definitions are implicitly scoped by a single machine or operating system. In contrast, we allow to specify generic check targets in terms of conditions that span over multiple distributed software components. Finally, standard OVAL objects often require system-specific information that depend on how and where to-be-checked configurations are stored. To increase the separation of concerns and provide the flexibility to choose among different mechanisms to access the configurations, we introduced the concept of collector which we made independent from the language fragment that encodes the check logic. As such, check authors remain agnostic with respect to potential configuration sources, which are instead cared for by system administrators who are in charge of (and have the knowledge for) specifying suitable collectors.

A category of software products called authenticated vulnerability and patch scanners also perform configuration validation. Unlike unauthenticated or network vulnerability scanners that probe a target host over the network, these products require valid credentials for the host under test in order to gather information relevant for given configuration checks. As representative vulnerability scanner, we consider Nessus [Nessus]: a widely adopted tool coming with a proprietary syntax for the definition of so-called audit checks. The same considerations apply as well to open-source alternatives like [OpenVAS]. Users can either write custom checks or subscribe to a commercial feed to receive compliance checks tailored for a variety of standards and regulations. Having comparable expressiveness, checks written in Nessus' proprietary language can be transformed into SCAP content. SCAP and Nessus' language also have in common that they focus on operating systems, which makes it difficult to specify checks on a more fine-granular level, i.e., for objects which cannot be easily identified relative to the operating system. Analogously to standard generic OVAL objects, Nessus' so-called *custom items* for Windows and Unix require the specification of file paths. The so-called *built-in checks* hide the configuration source from the check's author, but instead of making the source customizable, it is hard-coded. In contrast, our collectors allow to both decouple the configuration source from the check and to customize it for a given system infrastructure. Checks considering distributed system components are not supported at all (RL5). Nessus does also not allow to condition the applicability of the check on the basis of component properties (e.g., release level) or component relationships (RL3) but only on the basis of hard-coded keywords such as `Unix`.

So far we reviewed industry-adopted standards and products for configuration validation; in the remainder of this section we discuss instead related research work.

Researchers have proposed several approaches to assess the overall security level of systems by analyzing and reasoning about the potential combination of individual vulnerabilities (exploits) by an adversary [Chen2008], [Ou2005]. Though referring to SCAP specifications, these approaches do not look into the vulnerability specification itself, but use the language and related tools merely for the discovery of individual vulnerabilities.

In [Montanari2011], the problem of distributed configuration validation is tackled from the scalability perspective. The authors propose an algorithm to dispatch the evaluation of configuration checks to the nodes of a distributed infrastructure which guarantees resiliency and scalability properties. As we focus mainly on the design of the configuration language to maximize the integration with widely-accepted industry standards and tools, our contribution is complementary to theirs, where the attention is put on decentralizing the evaluation algorithm. They model both the system infrastructure and the configurations as RDF triples and they use Datalog-like rules to express configuration checks. The structure of our data source can be also seen as a constrained

RDF graph[2], which we mapped to a CIM model since, unlike RDF, CIM is supported by most existing CMDB products. Our check definitions have a richer structure than Datalog rules, which allows to tailor the language expressiveness to the configuration validation task (e.g., the specification of the checks' logic is clearly separated from that of checks' targets).

The work that most closely relates to ours is that of [Barrere2012]. The authors agree with our concerns about the lack of expressiveness of OVAL when considering configuration checks that apply to distributed systems. In particular, we share the requirement (RL3) of expressing conditions not only on individual components' properties, but on their relationships too. Accordingly, they propose to incorporate so-called relationships into the language, which closely correspond to our associations. However, unlike us, they view the components of a distributed system as individual network hosts, whereas we consider fine-grained software instances. As such, their language can express relationships between different network nodes (e.g., "communicates with") but not between software components within the same node (e.g., "deployed in"). They provide an algorithm to interpret extended OVAL checks which relies on the assumption that the Cfengine tool [CFEngine] — a distributed agent-based system for the management of autonomic networks — is installed on every node of the system infrastructure and serves both for configuration collection and target resolution. In contrast, by relying on existing management standards and technologies, our approach is agent-less and not tied to a single collection mechanism, as we argue that the flexibility to choose among several ones is crucial to achieve a better integration with current configuration management processes.

3.7 Discussion

A key assumption underlying our proposal is the availability of a federated data source that provides information on a variety of different aspects of an IT system: from the internal architecture of applications to the network reachability of distributed components. Unfortunately, in practice this information is often scattered over multiple repositories, such as vendor-specific management systems, network administration tools, dependency management systems, and encoded in different formats. Although all these tools can be viewed as partial instances of the ITIL's configuration management database concept, which we used as reference for our data source, a single and fully integrated repository is yet unlikely to be available out of the box. In this case, additional effort is required to create and maintain a federated data source. We have nevertheless reasons to believe that this effort will decrease with the evolution of configuration

[2]The function $\alpha : \mathcal{A} \rightarrow 2^{\mathcal{I} \times \mathcal{I}}$ defines a pseudograph where $\mathcal{A}$ is the set of edges and $\mathcal{I}$ is that of vertices. An RDF graph is also a pseudograph with the difference that the set of vertices includes not only instance identifiers (URLs) but also literals and that edge labels are chosen from the set of vertices [Gutierrez2003].

management systems. In fact, configuration data federation is to a large extent an engineering and standardization problem which has recently started to receive the interest of the IT industry, as proved by emerging standards such as the DMTF's Configuration Management Database Federation (CMDBf) specification [DMTF2010]. More conceptual are instead the issues connected to integrating the information coming from federated configuration data sources into a single data model. In Section 3.5.2, we bypassed this problem by fixing the data model of the data source *a priori*. In reality, the data model would be instead fixed by the context and in general there would be multiple data sources. We argue, however, that this can be treated as an ortogonal problem, which in fact has already been studied in literature on data integration [Ullman1997; Lenzerini2002]. For instance, by applying a *global as a view* approach, we could define our integrated data model as a view over multiple data sources.

The parallel with database techniques is not only superficial. Target definitions within check definitions are effectively queries whose answers, namely system components within system tests, are computed from the data source that act as the database instance. This suggests the possibility of pushing the evaluation algorithm (cf. Figure 3.9) entirely to the data source instead of computing it externally based on the result of multiple individual queries that implement functions π and α. In the case of a CIM-based data source (CMDB), this would translate to compiling target definitions to CIM queries which, once executed, would directly return the same result of our current evaluation algorithm.

The technique proposed in this chapter does not depend on the type of the to-be-checked configurations, because checks isolate syntactic features of configuration settings which are then directly compared to the expected states through a pre-determined set of comparison operators (e.g., boolean, string, or integer comparison). As such, it can be applied to check the correctness of the configuration of virtually any kind of software functionality: from security features, as discussed and exemplified here, to, e.g., quality of service parameters or application-specific functional requirements. On the other hand, the adoption of a purely syntactical approach can sometimes limit the expressiveness of check conditions, especially when checking configurations for which the gap between syntax and semantics is substantial. Consider, for instance, the OVAL definition $\texttt{acme}_{\texttt{A}}$, introduced in Example 3.3, which contains the test t_{authz} checking for the compliance of the authorization configuration of the DEx web application dedicated to ACME's customer A. Expressed in its simplest form, such a test would compare the XML configuration of the web application with the value mandated by the policy, i.e., the snippet presented in Figure 2.3 of Chapter 2. However, because of the flexibility of the configuration language, there exist several other alternative ways of expressing the same policy: for instance a security constraint that names two HTTP methods can be equivalently expressed as two security constraints applying to one method each. Clearly, a better formulation of the configuration check should treat all the alterna-

tives as equal since they encode the same semantics, but authoring such a check is not trivial, because it requires to incorporate the rules to interpret syntactic configuration constructs into the check's logic. Even harder would be to express weaker semantic conditions, such as to determine whether a discrepancy in the configuration yields a globally more or less permissive policy. Testing this kind of conditions constitutes a crucial step towards the semantic assessment of misconfigurations, i.e., evaluating the security impact of unexpected configuration settings in order to plan and prioritize remediation actions. The next chapter will contribute to solve this problem by proposing a formalization of the access control configuration of web applications.

3.8 Synthesis

This chapter presented a declarative language and a corresponding interpreter to unambiguously specify and execute syntactic checks to detect misconfiguration issues in distributed systems, e.g., situations where configuration settings do not comply with high-level policies, or expose the system to known vulnerabilities, or do not follow security best-practices. Our contribution builds on the SCAP specifications and, specifically, extends the OVAL standard to allow the specification of security checks for fine-grained components in a distributed environment and to separate the checks' logic from the configuration retrieval. Our configuration validation language has been adopted and validated in the scope of the PoSecCo project to support the Work Package 4 activities involved in the bottom-up configuration analysis approach (cf. Section 1.5) [Ponta2012]. Furthermore, the extension of the OVAL language as well as its interpretation semantics have been published in the proceedings of an international security conference [Casalino2012a].

A prototype implementation has been presented to illustrate the feasibility of our approach at the example of different configuration validation scenarios, using a CIM-based configuration management database for resolving target definitions, and relying on existing system management protocols, such as JMX, for the collection of configuration settings. This prototype became the core of the PoSecCo's focal prototype "Audit Interface" [Bettan2012], which has been evaluated on realistic scenarios by the two project's use-case partners: an auditor and a service provider. The results [Demetz2013] showed that the prototype helped to improve the coverage of the system under analysis and to reduce the time required by configuration validation activities. Moreover, the design and implementation of the COAS tool have been published in the proceedings of an international workshop on security [Casalino2012b].

Nothing endures but change.

—Heraclitus, in: Diogenes Lærtius, "Lives of the Philosophers"

4

Formalization and Change Impact Analysis of JEE Authorizations

▷ *Configuration validation techniques that are based exclusively on syntax are limited by the lack of semantic awareness. This is especially the case for expressive configuration languages, such as access control rulesets, for which it is hard to syntactically check interesting semantic conditions, e.g., testing for inclusion or equivalence of configurations with respect to permissiveness.*

In this chapter we tackle this problem for the access control configuration language of the JEE (Jave Enterprise Edition) framework, one of the most widespread web application frameworks currently available. We provide a denotational semantics for this language, on top of which we define a procedure, which we prove correct, to compare access control configurations with respect to their permissiveness. Finally, we implement our model and evaluate it with respect to the operational semantics of existing JEE container implementations through automated software testing. The findings include not only positive results supporting the correctness of our semantics, but also evidence of discrepancies that led to the discovery of a previously unknown implementation error in the Apache Tomcat JEE container. ◁

Chapter Outline

As argued in Chapter 1, human error is among the topmost causes of security misconfiguration and it is often the consequence of system administrators failing to predict the impact of configuration changes. Although syntactic configuration validation is a promising and widely-applicable approach for tackling this issue through configuration change monitoring, it suffers, as pointed out in the conclusion of Chapter 3, from the lack of semantic assessment capabilities, requiring human intervention to detect false positives and prioritize remediation actions. We argue that this shortcoming can be mitigated by developing automated techniques for semantic-aware configuration comparison which: (i) support administrators in anticipating the impact of configuration changes and (ii) complement syntactic validation with richer change analysis capabilities. Among all possible configurable security mechanisms, we focus on access control, which has both substantially expressive (non trivial) corresponding configuration languages and several well-studied formalizations.

More specifically, this chapter will restrict to the analysis of access control policies for Web applications, the security of which has become more and more important as a consequence of their increasing pervasiveness. For example, the failure to restrict URL accesses and security misconfiguration are considered as top ten Web application security risks by [OWASP2010; OWASP2013]. Furthermore, although the change impact analysis of access control policies already attracted researchers' attention, existing approaches are not suitable to cope with some peculiarities of the authorization rules for the Web, most notably the specification of patterns over hierarchical resources (URLs).

One of the most common frameworks for enterprise Web applications is the Java Platform, Enterprise Edition (also abbreviated as Java EE or JEE). The front-end of JEE Web applications is constituted of so-called *Web Components*, handling clients' HTTP requests and computing responses. The interface between the Web Components and the application server, which provides their execution environment, is standardized in the Java EE Servlet Specification [Coward2003]. This document establishes a contract between application server implementations on one side and Web applications on the other, prescribing, among others, a number of mechanisms to deal with security in JEE Web applications. Such mechanisms belong to two categories: *programmatic security* and *declarative security*. While the former describes functionalities which developers can use through an API within their applications' code, the latter refers to the enforcement of security properties (such as HTTP-based access control) achieved not through dedicated source code in the application, but through the declarative specification of security configurations. In this case the enforcement of security at runtime is transparent to the Web application's developer. In this chapter we focus on declarative security, which defines the syntax of access control configurations and informally describes their semantics. As depicted on the right-hand side of Figure 4.1, each Web application that is deployed within a JEE application server is bundled with a configuration file, the so-called *deployment descriptor*, where security and several other aspects of the runtime

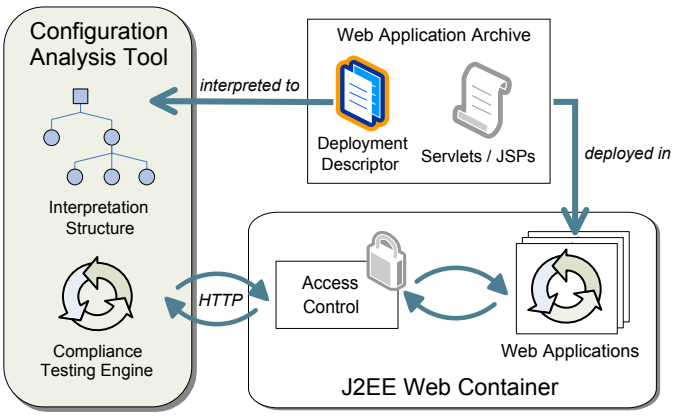


Figure 4.1: JEE framework (on the right) and chapter's contribution (on the left).

environment are configured.

Unfortunately, the declarative security semantics of the JEE Servlet Specification is defined in prose, which can lead to errors due to misinterpretation. Such errors, as shown by documented misconfiguration vulnerabilities [CVE-2010-0738; Polyakov2011], are among the causes of serious security issues. This motivates the need for provably correct formal tools with which system administrators can verify security properties of configurations. In line with related work on declarative access control languages [Bryans2005; Kolovski2007; Ni2009; Ahn2010; Ramli2011], we argue that it is important to equip JEE authorization configurations with formal semantics. This constitutes both an unambiguous reference for the JEE specification and a basic building block to support automated reasoning tasks, such as configuration change impact analysis, which we discuss in this chapter.

The rest of the chapter is organized as follows. Section 4.1 details the declarative security mechanisms offered by the JEE framework and the language of so-called *security constraints*, which allows to configure access control. In Section 4.2 we provide an interpretation structure for security constraints, upon which we define the formal semantics of corresponding access control policies (Section 4.3) and a comparison algorithm that is compatible with the partial order of permissiveness on policies (Section 4.4). Together with a prototype implementation, Section 4.5 compares our semantics with existing Web container implementations. The motivation for this experiment is twofold: on the one hand we empirically verify that the formal semantics complies with the informal one in the JEE Servlet Specification and, on the other hand, we are able to find cases where containers do not behave as expected. Experiments run on Tomcat and Glassfish application servers have led to the discovery of implementation errors. The left-hand side of Figure 4.1 shows how the different parts of our contribution interact with the JEE framework. Section 4.6 compares our approach to related work on XACML, access

$\langle ac \rangle$ ::= '`*`' | '`<`' $\langle rl \rangle$ '`>`'

$\langle rl \rangle$ ::= $\langle empty \rangle$ | `role` '`,`' $\langle rl \rangle$

$\langle up \rangle$ ::= $\langle empty \rangle$ | `part` | '`*`' | `part` '`/`' $\langle up \rangle$

$\langle upl \rangle$ ::= $\langle up \rangle$ | $\langle up \rangle$ '`,`' $\langle upl \rangle$

$\langle ml \rangle$ ::= `method` | `method` '`,`' $\langle ml \rangle$

$\langle wrc \rangle$ ::= '`{`' $\langle upl \rangle$ '`}`' '`[`' $\langle empty \rangle$ '`]`' | '`{`' $\langle upl \rangle$ '`}`' '`[`' $\langle ml \rangle$ '`]`'

$\langle wrcl \rangle$ ::= $\langle wrc \rangle$ | $\langle wrc \rangle$ '`,`' $\langle wrcl \rangle$

$\langle sc \rangle$ ::= $\langle wrcl \rangle$ | $\langle wrcl \rangle$ $\langle ac \rangle$

$\langle scl \rangle$ ::= $\langle sc \rangle$ | $\langle sc \rangle$ '`\n`' $\langle scl \rangle$

Figure 4.2: Shorthand syntax for security constraints.

control frameworks for Web-services and other security analysis tools for JEE applications. Section 4.7 discusses our proposal and outlines some further technical perspectives. Finally, Section 4.8 concludes the chapter with a summary of our contribution and results.

4.1 Security Constraints

The security-related fragment of the deployment descriptor is composed by the *security constraints* XML tags. For the sake of conciseness, we provide in Figure 4.2 a BNF grammar modeled from the XML grammar defined by the Servlet specification.

In a web application the to-be-protected resources are identified by URLs accessible via HTTP methods. Hence, the language offers a construct to specify URL patterns ($\langle up \rangle$), which are sequences of strings (URL parts) separated by `'/'` and possibly terminated by a wildcard (`'/*'`). Patterns ending with such a wildcard identify the entire hierarchy of URLs sharing the same prefix.

A web resource collection ($\langle wrc \rangle$) consists then of a list of URL patterns ($\langle upl \rangle$) followed by a (possibly empty) list of HTTP methods ($\langle ml \rangle$).

In a security constraint ($\langle sc \rangle$), access control is configured by associating web resource collections with up to one authorization constraint ($\langle ac \rangle$), that is the set of roles allowed to access the mentioned resources. The special role name `'*'` is a shorthand for all the roles defined inside the deployment descriptor. The initial non-terminal symbol $\langle scl \rangle$ represents a list of security constraints.

Example 4.1: Security constraints for ACME

The following snippet shows the example security contraints for one of ACME's customers, introduced in Figure 2.3 of Chapter 2, encoded according to the shorthand syntax of Figure 4.2.

```
1 {/manager/*} [DELETE, PUT] <>
2 {/manager/*} [] <dex-mgr>
3 {/partner/*} [] <dex-mgr, dex-tp>
```

The security constraint of line 2 grants the role `dex-mgr` access to the `/manager/*` URL pattern with any HTTP method. At the same time, it specifies that no other role can access URLs that match this pattern. Similarly, the constraint of line 3 ensures that only the members of either `dex-mgr` or `dex-tp` roles are granted access to the `/partner/*` URL pattern, with, again, any HTTP method.

The constraint of line 1 ensures instead that HTTP methods `DELETE` and `PUT`, not being implemented by the web application, are never accessible within the manager console. This is expressed by including an empty list of roles in the constraint.

Any other URL that does not match these two patterns is unconstrained. For instance, the root `/` is accessible by any, possibly unauthenticated, user.

This example will be used throughout the rest of the chapter, with abbreviated identifiers (e.g., `m` for `manager`, `p` for `partner`, etc.).

According to the informal semantics from [Coward2003], in order to have access granted, a user must be a member of *at least one of the roles* named in the security constraint (or implied by `'*'`) that matches to her/his HTTP request. An empty authorization constraint means that *nobody* can access the resources, whereas access is granted to *any* (possibly unauthenticated) user in case the authorization constraint is omitted. Unauthenticated access is also allowed by default to any unconstrained resources. It's worth noting that an intuitively insignificant syntactic difference, such as omitting the authorization constraint instead of specifying an empty one, corresponds to a major discrepancy in semantics, respectively *allow all* or *deny all* behaviours are obtained.

In case the same URL pattern and HTTP method occur in different security constraints, their authorization constraints have to be composed. If two non-empty authorization constraints are composed, the result is the *union* of the two sets of allowed roles. If one of the two allows unauthenticated access, the composition also does, conceptually resulting again in a *union*. In contrast, if one of the sets of roles is empty, their composition is empty. Constraints on more specific URL patterns (e.g., `/a/b`) always override more general ones (e.g., `/a/*`).

If some HTTP methods are explicitly mentioned in a web resource collection, all the other methods are unconstrained, whereas, if none is named, every method is implicitly constrained. Verb tampering attacks [CVE-2010-0738; Polyakov2011] exploit this

behaviour to bypass the access control check in vulnerable web applications that (i) handle requests on unimplemented methods (e.g., `HEAD`) as ordinary ones (e.g., `GET`) instead of correctly returning an appropriate HTTP error to the client, and (ii) exhibit a badly configured deployment descriptor that constrains only the implemented methods.

The peculiar handling of unconstrained methods, combined with the fact that most specific constraints take precedence, leads to particularly counterintuitive behaviours, as illustrated by the following example.

Example 4.2: Combination of security constraints

Assume that one of ACME's customers, say customer A, decides to restrict the access control policy of its dedicated instance of the DEx service in order to forbid trading partners to submit EDI documents and let them only access the service to either retrieve or delete the document they are recipient of.

To enforce this new policy, as the EDI exchange web service is available as a RESTful API at the URL `/partner/edi/` (cf. Section 2.2), ACME's system administrators decide to restrict the access rights of the `dex-tp` role for this location to the HTTP methods `GET` and `DELETE` only.

To do so, they might modify the security constraints configuration of Example 4.1 as follows:

```
1 {/manager/*} [DELETE, PUT] <>
2 {/manager/*} [] <dex-mgr>
3 {/partner/*} [] <dex-mgr>
4 {/partner/edi/*} [GET, DELETE] <dex-tp>
```

Note that, while lines 1 and 2 remained unchanged, lines 3 and 4 seemingly make the configuration more restrictive, as `dex-tp` has now access only to a subset of HTTP methods on a more specific URL pattern. However, with this new constraint, HTTP requests such as `(/partner/edi/123, PUT)` and `(/partner/edi/123, POST)` are granted to anyone, even unauthenticated users! Moreover, `dex-mgr` users cannot access any more any URL matching to `/partner/edi/*` with methods `GET` and `DELETE`.

This is the case because `/partner/edi/*` is more specific than `/partner/*`, hence line 3 is overridden by line 4. However the latter does not define behaviour for the `PUT` and `POST` methods, so the default allow policy is applied.

A better formulation, which does not override the behavior imposed by line 3, requires including the additional constraint: `{/partner/edi/*} [] <dex-mgr>`.

4.2 Interpretation Structure

In this section we define a mathematical structure, named Web Access Control Tree (WACT), that encodes authorization rules on hierarchical resources and we describe how security constraints can be interpreted into such a structure. This step provides the foundations for the definition of a denotational semantics of JEE access control configurations.

A function $[\![\cdot]\!]_{LIT}$ is defined for each non-terminal symbol $\langle lit \rangle$ in the grammar given in Figure 4.2. These functions derive from case analysis on the structure of the language. Terminal symbols are interpreted within an associated domain of semantic objects. For instance, role literals in `'role'` are interpreted by the function $[\![\cdot]\!]_R$: `'role'` $\rightarrow \mathcal{R}$ in elements of the roles domain $\mathcal{R}$. Likewise $[\![\cdot]\!]_M$ maps `'method'` literals in the domain of HTTP methods $\mathcal{M}$, and $[\![\cdot]\!]_S$ interprets URL `'part'`s in an infinite domain of strings $\mathcal{S}$. The final interpretation function is $[\![\cdot]\!]_{SCL}$, that is, the interpretation of initial symbol of the grammar.

4.2.1 Authorization Constraints

The interpretation function of authorization constraints is given by the function $[\![\cdot]\!]_{AC}$ defined in (4.1). This function maps every authorization constraint $\langle ac \rangle$ to an element of the powerset of the role domain. The function $[\![\cdot]\!]_{RL}$, defined by (4.2), *folds* roles into a set.

$$[\![\langle ac \rangle]\!]_{AC} = \begin{cases} \mathcal{R} & \text{if } \langle ac \rangle = \texttt{'*'}, \\ [\![\langle rl \rangle]\!]_{RL} & \text{if } \langle ac \rangle = \texttt{'<'}\,\langle rl \rangle\,\texttt{'>'}. \end{cases} \tag{4.1}$$

$$[\![\langle rl \rangle]\!]_{RL} = \begin{cases} \emptyset & \text{if } \langle rl \rangle = \langle empty \rangle, \\ \{[\![\texttt{'role'}]\!]_R\} \cup [\![\langle rl \rangle']\!]_{RL} & \text{if } \langle rl \rangle = \texttt{'role'}\;\texttt{','}\;\langle rl \rangle'. \end{cases} \tag{4.2}$$

Fold, also known as reduce or accumulate, is a standard high-order functional operation on containers. It has an intuitive meaning: for instance, according to (4.2), the syntactic role list "`<dex-mgr>`" (line 2 of Example 4.1) is turned into the subset of $\mathcal{R} = \{\texttt{M}, \texttt{P}\}$ that contains only one symbol for the role `dex-mgr`: $[\![\texttt{<dex-mgr>}]\!]_{RL} = \{\texttt{M}\}$. We use capital letters to denote semantic role symbols (i.e., `M`, `P` for `dex-mgr` and `dex-tp` respectively) in order to distinguish them from the lowercase identifiers of URL parts (i.e., `m`, `p` for `manager` and `partner` respectively).

Other similar fold functions that interpet syntactic lists into sets of semantic objects are used throughout this section, namely $[\![\cdot]\!]_{UPL}$, $[\![\cdot]\!]_{ML}$ and $[\![\cdot]\!]_{WRCL}$ for URLs, methods and web resource collections respectively. Their definitions rest on the same principle and hence are not reported here.

In order to capture the semantics of authorization constraints, a partial order $\leq_R$ between sets of roles is defined. To take the case of unauthenticated users into account,

the symbol $\top$ is added. The role lattice $\mathcal{L}$ is the complete lattice given by the powerset of the role domain, ordered by set inclusion, and containing the additional element $\top \notin 2^{\mathcal{R}}$.

Definition 10 (Role Lattice)**.** *The (complete) role lattice is defined by the partially-ordered set* $\mathcal{L} = \langle 2^{\mathcal{R}} \cup \{\top\}, \leq_R \rangle$*, where* $R_A \leq_R R_B$ *iff* $R_B = \top$ *or* $R_A \subseteq R_B$*.*

The top element $\top$ semantically corresponds to the default *allow all* authorization constraint implicitly associated with any non-constrained web resource. In contrast, the bottom element $\emptyset$ represents a *deny all* authorization constraint.

Equation (4.3) formally captures the composition rules of different authorization constraints mentioned in Section 4.1. The operator $\otimes : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ performs composition by relying on the least upper bound lattice operator ($\bigsqcup$).

$$R_A \otimes R_B = \begin{cases} \emptyset & \text{if } R_A = \emptyset \text{ or } R_B = \emptyset, \\ R_A \bigsqcup R_B & \text{otherwise.} \end{cases} \tag{4.3}$$

To illustrate the behaviour of this composition rule, we consider the following equalities: $\{r_1, r_2\} \otimes \top = \top$, $\{r_1, r_2\} \otimes \{r_3, r_1\} = \{r_1, r_2, r_3\}$, $\{r_1, r_2\} \otimes \emptyset = \emptyset$ and $\top \otimes \emptyset = \emptyset$. Note that this is consistent with the behaviour described in Example 4.1 for the composition of the security constraints: `{/manager/*} [DELETE, PUT] <>` and `{/manager/*} [] <dex-mgr>`. For methods `DELETE` and `PUT`, the composition between the empty set of roles of the first constraint and the non-empty one of the second yields an empty set: $\emptyset \otimes \{\texttt{M}\} = \emptyset$. This means that any URL matching the pattern `/manager/*` will not be accessible via these two methods by any user (denial takes precedence); not even by the members of the `dex-mgr` role.

4.2.2 Web Resource Collections

The resources being subject to access control in a web application are URLs. The URL hierarchy must be taken into account while evaluating access control requests, since a URL pattern ending with a wildcard matches every URL sharing its prefix. We therefore interpret URL patterns as a tree, where each node is a prefix-ordered sequence of symbols.

Definition 11 (URL)**.** *A URL* $u \in \mathcal{U}$ *is a (possibly empty) sequence of symbols each one belonging to* $\mathcal{S}$*, and ending with at most one symbol belonging to the set* $\mathcal{E} = \{\epsilon, *\}$[1]*, where* $\mathcal{S} \cap \mathcal{E} = \emptyset$*:*

1. $u = \langle\rangle$ *is an (empty) URL;*
2. $u = \langle s_0, \ldots, s_n \rangle$*, with* $n > 0$ *and* $s_0, \ldots, s_n \in \mathcal{S}$*, is a URL;*

[1]The symbol ϵ is used to differentiate files from folders, e.g., between `/a/` and `/a`.

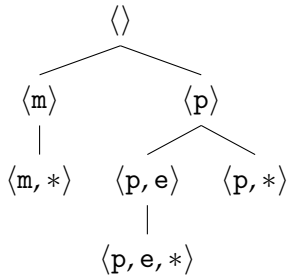


Figure 4.3: URL tree of ACME's DEx web application.

3. $u = \langle s_0, \ldots, s_n, s_e \rangle$, *with* $n > 0, s_0, \ldots, s_n \in \mathcal{S}$ *and* $s_e \in \mathcal{E} = \{\epsilon, *\}$, *is a URL.*

For a given URL $u = \langle s_0, \ldots, s_n \rangle$ its *length*, written $|u|$, equals $n + 1$; the length of the empty URL being 0. The l-long *prefix* of u, written $u^{\leq l}$ is the sequence $\langle s_0, \ldots, s_{l-1} \rangle$, with $u^{\leq 0} = \langle\rangle$. The i^{th} symbol s_i of u is written u_i. Equality of URLs is defined in the traditional way. The URL concatenation operator $\oplus : \mathcal{U} \times \mathcal{U} \to \mathcal{U}$ is defined as follows:

$$u \oplus v = \begin{cases} \langle u_0, \ldots, u_{|u|}, v_0, \ldots, v_{|v|} \rangle & \text{if } u_{|u|} \in \mathcal{S}, \\ \text{undefined} & \text{if } u_{|u|} \in \mathcal{E}. \end{cases} \tag{4.4}$$

For instance, let $\mathcal{S} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $u = \langle \mathtt{a}, \mathtt{b} \rangle$. The following equalities hold: $|u| = 2$, $u^{\leq 1} = \langle \mathtt{a} \rangle$, $v = u \oplus \langle \mathtt{c} \rangle = \langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$, $v_2 = \mathtt{c}$, $w = u \oplus \langle \epsilon \rangle = \langle \mathtt{a}, \mathtt{b}, \epsilon \rangle$. Finally $w \oplus \langle \mathtt{c} \rangle$ is not defined. This is indeed consistent with reality: since the URL w represents the file `/a/b`, and not the directory `/a/b/`, any further concatenation is meaningless.

Definition 12 (URL Tree)**.** *A URL tree is a non empty, finite, partially ordered set* $\langle U, \prec \rangle$ *such that:*

1. $U \subseteq \mathcal{U}$*;*
2. U *is prefix-closed, i.e.,* $u \in U$ *and* $|u| > 0 \Rightarrow u^{\leq |u|-1} \in U$*, in particular the empty URL* $\langle\rangle$ *always belongs to* U*;*
3. $\prec$ *is the weak partial order defined as* $u \prec v$ *iff* $|u| \leq |v|$ *and* $u = v^{\leq |u|}$*.*

The set $\mathcal{U}^*$ *denotes all the possible URL trees.*

Proposition 1. *The relation* $\prec$ *is indeed a partial order for* $\mathcal{U}$*. Moreover, for any URL tree* $\langle U, \prec \rangle$ *the set of predecessors of any of its elements* $u{\downarrow} = \{p \mid p \prec u\}$ *is well-ordered.*

Proof. See Appendix B.1. □

Proposition 1 ensures that a URL tree is indeed a tree according to the set-theoretic definition. Figure 4.3 depicts the URL tree corresponding to the interpretation of all the URL patterns in Example 4.1.

Every URL pattern $\langle up \rangle$ is interpreted as a URL through the function $[\![\cdot]\!]_{UP}$ recursively defined in (4.5). Intuitively, a URL is simply a sequence of identifiers (`'part'`s) separated by the `'/'` character. For instance, the concrete URL pattern `/p/e/*` is turned into the sequence $[\![\texttt{\{/p/e/*\}}]\!]_{UP} = \langle \texttt{p}, \texttt{e}, * \rangle$.

$$[\![\langle up \rangle]\!]_{UP} = \begin{cases} \langle \epsilon \rangle & \text{if } \langle up \rangle = \langle empty \rangle, \\ \langle * \rangle & \text{if } \langle up \rangle = \texttt{'*'}, \\ \langle [\![\texttt{'part'}]\!]_S \rangle & \text{if } \langle up \rangle = \texttt{'part'}, \\ \langle [\![\texttt{'part'}]\!]_S \rangle \oplus [\![\langle up \rangle']\!]_{UP} & \text{if } \langle up \rangle = \texttt{'part'}\ \texttt{'/'}\ \langle up \rangle'. \end{cases} \tag{4.5}$$

The combination of URL patterns and HTTP methods into web resource collections is done by performing the cartesian product of the two sets by means of the function $[\![\cdot]\!]_{WRC}$ defined in (4.6). This definition is consistent with the Servlet specification, since it states that naming no methods means that every method is constrained.

$$[\![\langle wrc \rangle]\!]_{WRC} = \begin{cases} [\![\langle upl \rangle]\!]_{UPL} \times \mathcal{M} & \text{if } \langle wrc \rangle = \langle upl \rangle, \\ [\![\langle upl \rangle]\!]_{UPL} \times [\![\langle ml \rangle]\!]_{ML} & \text{if } \langle wrc \rangle = \langle upl \rangle \langle ml \rangle. \end{cases} \tag{4.6}$$

For instance, if we consider the constraint on line 4 from Example 4.2, then $[\![\texttt{\{/p/e/*\}, [GET, DELETE]}]\!]_{WRC}$ equals the set with two elements:

$$\{\langle \langle \texttt{p}, \texttt{e}, * \rangle, \texttt{GET} \rangle, \langle \langle \texttt{p}, \texttt{e}, * \rangle, \texttt{DELETE} \rangle\}.$$

4.2.3 Security Constraints

Security constraints describe which roles are allowed to access to the nodes of a URL tree. To encode this information, we enrich a URL tree with a labeling function that maps nodes of the tree, i.e., URLs, and HTTP methods to a respective set of authorized roles. We name the resulting structure *Web application Access Control Tree*.

Definition 13 (Web application Access Control Tree)**.** *A Web application Access Control Tree (WACT) is a pair $\langle U, \rho \rangle$, where $U \in \mathcal{U}^*$ is a URL tree as defined in Definition 12 and $\rho : \mathcal{U} \times \mathcal{M} \to \mathcal{L}$ is a partial function giving the set of roles allowed to access a pair $\langle u, m \rangle$. The set of all WACTs is $\mathcal{T}$.*

A security constraint is interpreted as a WACT through the function $[\![\cdot]\!]_{SC}$ which maps any constraint $\langle sc \rangle$ to the WACT $[\![\langle sc \rangle]\!]_{SC} = \langle U, \rho \rangle$.

$$U = \{w \in \mathcal{U} \mid w \prec u \wedge \langle u, \cdot \rangle \in [\![\langle wrcl \rangle]\!]_{WRCL}\} \tag{4.7}$$

$$\rho(u, m) = \begin{cases} \top & \text{if } \langle sc \rangle = \langle wrcl \rangle \wedge \langle u, m \rangle \in [\![\langle wrcl \rangle]\!]_{WRCL}, \\ [\![\langle ac \rangle]\!]_{AC} & \text{if } \langle sc \rangle = \langle wrcl \rangle \langle ac \rangle \wedge \langle u, m \rangle \in [\![\langle wrcl \rangle]\!]_{WRCL}, \\ \text{undefined} & \text{if } \langle u, m \rangle \notin [\![\langle wrcl \rangle]\!]_{WRCL}. \end{cases} \tag{4.8}$$

As defined in (4.7), $U \in \mathcal{U}^*$ is given by the prefix-closure of every URL in the web resource collections, for instance $\{\langle\rangle, \langle \mathtt{p}\rangle, \langle \mathtt{p}, \mathtt{e}\rangle, \langle \mathtt{p}, \mathtt{e}, *\rangle\}$ is the prefix closure of $\langle \mathtt{p}, \mathtt{e}, *\rangle$. The function ρ, according to (4.8), is defined only for the URL/method pairs contained in the interpretation of the web resource collections. It maps all such pairs:

- to the $\top$ element of the role lattice, in case no authorization constraints are specified;
- to the set of roles given by interpreting the authorization constraints $[\![\langle ac\rangle]\!]_{AC}$, otherwise.

Several trees obtained from $[\![\cdot]\!]_{SC}$ have to be combined when a web applications' deployment descriptor contains more than one security constraint. The union of two WACTs $\langle U_1, \rho_1\rangle \dot\cup \langle U_2, \rho_2\rangle$ is the tree $\langle U_1 \cup U_2, \rho_U\rangle$ where ρ_U is defined by (4.9). In the case where both trees define a set of roles for a common pair $\langle u, m\rangle$, the corresponding role sets are merged by using the operator $\otimes$ defined by (4.3).

$$\rho_U(u,m) = \begin{cases} \rho_1(u,m) \otimes \rho_2(u,m) & \text{if } \langle u,m\rangle \in \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) \\ \rho_1(u,m) & \text{if } \langle u,m\rangle \in \mathrm{dom}(\rho_1) \setminus \mathrm{dom}(\rho_2) \\ \rho_2(u,m) & \text{if } \langle u,m\rangle \in \mathrm{dom}(\rho_2) \setminus \mathrm{dom}(\rho_1) \end{cases} \tag{4.9}$$

Finally, Equation (4.10) folds all the security constraints from a deployment descriptor ($\langle scl\rangle$) to produce a single WACT.

$$[\![\langle scl\rangle]\!]_{SCL} = \begin{cases} [\![\langle sc\rangle]\!]_{SC} & \text{if } \langle scl\rangle = \langle sc\rangle, \\ [\![\langle sc\rangle]\!]_{SC} \dot\cup [\![\langle scl\rangle']\!]_{SCL} & \text{if } \langle scl\rangle = \langle sc\rangle\langle scl\rangle'. \end{cases} \tag{4.10}$$

For instance, the two security constraints `{ /a, /a/b } [GET] <x>` and `{ /a/b } [GET, POST] <y>` turn into the WACTs $t_1 = \langle U_1, \rho_1\rangle$ and $t_2 = \langle U_2, \rho_2\rangle$ respectively, with $\rho_1(\langle \mathtt{a}\rangle, \mathtt{GET}) = \{\mathtt{x}\}$, $\rho_1(\langle \mathtt{a}, \mathtt{b}\rangle, \mathtt{GET}) = \{\mathtt{x}\}$, $\rho_2(\langle \mathtt{a}, \mathtt{b}\rangle, \mathtt{GET}) = \{\mathtt{y}\}$ and $\rho_2(\langle \mathtt{a}, \mathtt{b}\rangle, \mathtt{POST}) = \{\mathtt{y}\}$. Their union is the WACT $t_1 \dot\cup t_2 = \langle U_1 \cup U_2, \rho\rangle$, with $\rho(\langle \mathtt{a}, \mathtt{b}\rangle, \mathtt{GET}) = \{\mathtt{x}, \mathtt{y}\}$, $\rho(\langle \mathtt{a}, \mathtt{b}\rangle, \mathtt{POST}) = \{\mathtt{y}\}$ and $\rho(\langle \mathtt{a}\rangle, \mathtt{GET}) = \{\mathtt{x}\}$.

4.3 Access Control Semantics

According to the Servlet specification [Coward2003, Section 12.8.3], when a container receives a request, it shall determine the applicable security constraints and enforce the role-based access control policy which results from their interpretation.

Each request is a triple $\langle u, m, R\rangle \in \mathcal{U} \times \mathcal{M} \times \mathcal{L}$ composed by (i) a URL identifying the requested resource, (ii) a HTTP method and (iii) an element of the role lattice representing either the set of roles assigned to the user who submitted the request or an unauthenticated request in case $R = \top$. For any such request, enforcing access control

requires determining whether to accept or deny the request, based on the policy expressed by the given security constraints. Formally, this is equivalent to implementing the function

$$\delta : \mathcal{U} \times \mathcal{M} \times \mathcal{L} \to \{0, 1\}, \tag{4.11}$$

where every request is mapped to either 0 (deny) or 1 (allow).

The rules to determine the behaviour of δ are informally described in [Coward2003, Section 12.8.3] and may be summarized as "*constraints on most specific URL patterns take precedence*". In the remainder of this section we show how the hierarchical structure of WACTs can be conveniently exploited to define δ.

Recall that the partial mapping ρ in the WACT associates URLs and method with the corresponding sets of granted roles. However, a request may refer to a URL which is not mapped by ρ, in which case the most specific constraint shall apply. In order to capture this behaviour, for a URL tree U, we denote the set of $*$-predecessors of every URL $u \in U$ as $u\downarrow_*$. The elements of this set are all the immediate successors of the ancestors of u, ending with the symbol $* \in \mathcal{E}$. Formally,

$$u\downarrow_* = \{w \oplus \langle * \rangle \mid w \in U \wedge w \prec u \wedge w \oplus \langle * \rangle \in U\}.$$

Note that this set may be empty, in case u does not have any ancestor that satisfies such a property. For instance, if $U = \{\langle \mathtt{a} \rangle, \langle \mathtt{a}, \mathtt{b} \rangle, \langle \mathtt{a}, \mathtt{b}, * \rangle, \langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle, \langle \mathtt{a}, \mathtt{b}, \mathtt{c}, * \rangle, \langle \mathtt{a}, \mathtt{d} \rangle\}$, then $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle \downarrow_* = \{\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, * \rangle, \langle \mathtt{a}, \mathtt{b}, * \rangle\}$, but $\langle \mathtt{a}, \mathtt{d} \rangle \downarrow_* = \emptyset$.

The $*$-predecessors of a URL u are all and the only URL patterns that match u. Among them, we shall consider the most specific one, which is the closest one to u in the hierarchy or, equivalently, the one having maximum length. Let max denote the function that maps any set of URLs to the subset of them having maximum length. The next proposition guarantees that in any (non empty) set of $*$-predecessors there is a unique maximum element.

Proposition 2. *Given a URL Tree $U \in \mathcal{U}^*$ and a URL $u \in U$, the set $u\downarrow_*$ of $*$-predecessors of u has at most one maximum element and it has exactly one element iff $u\downarrow_*$ is not empty.*

Proof. See Appendix B.1. □

We are now in a position to formally express the aforementioned *most specific applies* behaviour. This is done in the following definition, where the function ρ is extended to the entire (infinite) domain of all possible URLs.

Definition 14 (Effective Roles)**.** *Given a WACT $t = \langle U, \rho \rangle$ the set of effective roles for each couple $\langle u, m \rangle \in \mathcal{U} \times \mathcal{M}$ is given by the function $\hat{\rho} : \mathcal{U} \times \mathcal{M} \to \mathcal{L}$*

$$\hat{\rho}(u, m) = \begin{cases} \rho(u, m) & \text{if } \langle u, m \rangle \in \mathrm{dom}(\rho) \\ \rho(w, m) & \text{else if } \max(u\downarrow_*) = \{w\} \wedge \langle w, m \rangle \in \mathrm{dom}(\rho) \\ \top & \text{otherwise} \end{cases} \tag{4.12}$$

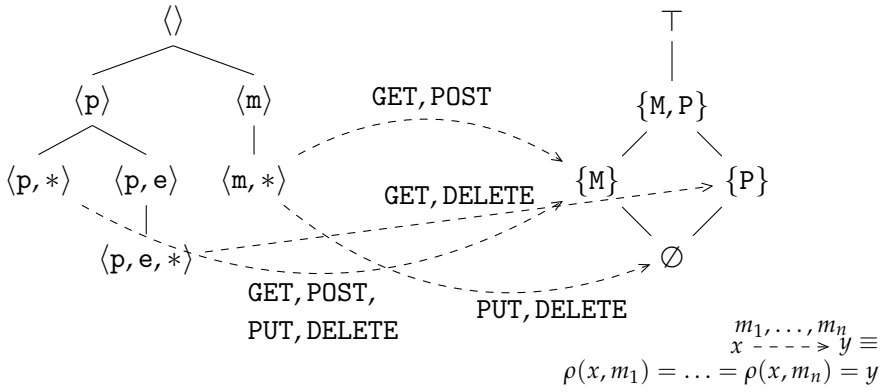


Figure 4.4: WACT obtained from Example 4.2.

From Proposition 2 it follows that $\max(u_* \downarrow)$ always contains at most one element, thus (4.12) is well-defined.

The decision function δ is defined straightforwardly from the set of effective roles: access to $\langle u, m \rangle$ is granted either if the user is unauthenticated and the resource accessible to unauthenticated users or if the user endorses at least one role in the set of effective roles associated with $\langle u, m \rangle$.

Definition 15 (Decision Function)**.** *Every WACT $\langle U, \rho \rangle \in \mathcal{T}$ has a corresponding access control decision function $\delta : \mathcal{U} \times \mathcal{M} \times \mathcal{L} \to \{0, 1\}$ defined as follows:*

$$\begin{aligned} \delta(u, m, \top) &= 1 \quad \textit{if} \quad \hat{\rho}(u, m) = \top \\ \delta(u, m, R) &= 1 \quad \textit{if} \quad \hat{\rho}(u, m) \bigsqcap R \neq \emptyset \\ \delta(u, m, R) &= 0 \quad \textit{otherwise.} \end{aligned} \tag{4.13}$$

Example 4.3: Effective roles and decision function

Let us consider the following security constraints, which we introduced in Example 4.2.

```
1 {/manager/*} [DELETE, PUT] <>
2 {/manager/*} [] <dex-mgr>
3 {/partner/*} [] <dex-mgr>
4 {/partner/edi/*} [GET, DELETE] <dex-tp>
```

When interpreted by $[\![\cdot]\!]_{SCL}$, this set of constraints yields the WACT $\langle U, \rho \rangle$ represented in Figure 4.4. The URL tree U is depicted on the left, while on the right appears the role lattice $\mathcal{L}$. The partial function ρ is represented by a set of labeled dashed arrows that map nodes of the tree and HTTP methods to elements in the role lattice. For each of the following example requests, we compute the set of effective roles and the corresponding access control decisions:

- $\hat{\rho}(\langle \mathtt{p}, \mathtt{x}, \mathtt{y} \rangle, \mathtt{GET}) = \hat{\rho}(\langle \mathtt{p} \rangle, \mathtt{GET}) = \{\mathtt{M}\}$ because the constraint on line 3 applies. Hence $\delta(\langle \mathtt{p}, \mathtt{x}, \mathtt{y} \rangle, \mathtt{GET}, \{\mathtt{M}\}) = \delta(\langle \mathtt{p}, \mathtt{x}, \mathtt{y} \rangle, \mathtt{GET}, \{\mathtt{M}, \mathtt{P}\}) = 1$, but $\delta(\langle \mathtt{p}, \mathtt{x}, \mathtt{y} \rangle, \mathtt{GET}, \{\mathtt{P}\}) = 0$;
- $\hat{\rho}(\langle \mathtt{m}, \mathtt{z} \rangle, \mathtt{GET}) = \hat{\rho}(\langle \mathtt{m} \rangle, \mathtt{GET}) = \{\mathtt{M}\}$ because the constraint on line 2 applies, but $\hat{\rho}(\langle \mathtt{m}, \mathtt{z} \rangle, \mathtt{DELETE}) = \emptyset$ because of the constraint on line 1. Hence $\delta(\langle \mathtt{m}, \mathtt{z} \rangle, \mathtt{GET}, \{\mathtt{M}\}) = 1$, but $\delta(\langle \mathtt{m}, \mathtt{z} \rangle, \mathtt{DELETE}, \{\mathtt{M}\}) = 0$;
- $\hat{\rho}(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{GET}) = \{\mathtt{P}\}$ because the constraint on line 4 applies, hence $\delta(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{GET}, \{\mathtt{P}\}) = 1$. However, $\hat{\rho}(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{POST}) = \top$ because no security constraints apply, therefore $\delta(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{POST}, \{\mathtt{P}\}) = \delta(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{POST}, \top) = 1$.

4.4 Change Impact Analysis

In the previous section we defined the semantics of security constraints in terms of the permissiveness of their corresponding access control policy, i.e., the space of all granted (respectively denied) permissions. We now extend the above reasoning to measure the impact of syntactic changes in security constraints in terms of semantic changes in permissiveness. Such a characterization is intuitively useful to support the management of configuration changes. For instance, one may wish to verify that the introduction of a new security constraint leads to a more restrictive policy without yielding undesired side-effects. Another example is refactoring: as an access control configuration evolves in time, more and more rules may be introduced. At some point it may be worth rewriting the whole set of constraints into a clearer and maybe shorter one, but it must be ensured that the new policy behaves exactly the same as the old one.

In order to tackle this problem, we define a relation between pairs of WACTs and show that it corresponds to a partial order on permissiveness that is compatible with the semantics of access control decisions.

Intuitively, a WACT t_1 is less permissive than t_2, written $t_1 \leq_T t_2$ if for any URL in any of the two trees and for any method, the set of effective roles of t_1 is included in that of t_2.

Definition 16 (Order of Permissiveness)**.** *For any pair of WACTs $t_1 = \langle U_1, \rho_1 \rangle$ and $t_2 = \langle U_2, \rho_2 \rangle$, we define the relation $\leq_T$ as follows:*

$$t_1 \leq_T t_2 \text{ iff } \forall u \in U_1 \cup U_2, m \in \mathcal{M},\ \hat{\rho}_1(u, m) \leq_R \hat{\rho}_2(u, m). \tag{4.14}$$

The next proposition ensures the correctness of this definition, by stating that $t_1 \leq_T t_2$ is equivalent to δ_1 granting access to all possible (infinite) requests only if δ_2 does.

Proposition 3. *The relation $\leq_T$ is a partial order of permissiveness. That is, for WACTs $t_1, t_2 \in \mathcal{T}$ and corresponding decision functions δ_1, δ_2,*

$$t_1 \leq_T t_2 \Leftrightarrow \forall \langle u, m, R \rangle \in \mathcal{U} \times \mathcal{M} \times \mathcal{L},\ \delta_1(u, m, R) = 1 \Rightarrow \delta_2(u, m, R) = 1.$$

Input: WACTs $t_1 = \langle U_1, \rho_1 \rangle$ and $t_2 = \langle U_2, \rho_2 \rangle$
Output: $t_1 \leq_T t_2$
foreach $u \in U_1 \cup U_2$ **do**
  foreach $m \in \mathcal{M}$ **do**
    if $\hat{\rho}_1(u, m) \not\leq_R \hat{\rho}_2(u, m)$ **then**
      return false;
return true;

Figure 4.5: WACT permissiveness comparison.

Proof. See Appendix B.1. □

Proposition 3 gives an effective method to check whether a configuration is semantically more permissive than another. It is sufficient to verify if inclusion of roles holds for each node in the WACT. If $u \notin U_1 \cup U_2$, then $\hat{\rho}_{t_1}(u, m) = \hat{\rho}_{t_2}(u, m) = \top$ by (4.12), thus only a *finite* set of URLs have to be checked.

The algorithm reported in Figure 4.5 performs this computation. As the cardinality of the set of HTTP methods $\mathcal{M}$ is finite and constant and $\hat{\rho}$ can be precomputed for each WACT, the algorithm runs linearly in the number of URL prefixes. As such, it is suitable for interactive applications. For instance, we can envisage a configuration editing environment equipped with static analysis capabilities based on WACT, where the user is informed instantaneously about the impact on permissiveness of each change in the configuration.

Example 4.4: WACT comparison

Let us compare WACTs t_1 and t_2, obtained from the Examples 4.1 and 4.2 respectively. The first tree, $t_1 = \langle U_1, \rho_1 \rangle$, is the result of interpreting the security constraints contained in the next snippet.

```
1 {/manager/*} [DELETE, PUT] <>
2 {/manager/*} [] <dex-mgr>
3 {/partner/*} [] <dex-mgr, dex-tp>
```

The second tree, $t_2 = \langle U_2, \rho_2 \rangle$, corresponds instead to the following configuration.

```
4 {/manager/*} [DELETE, PUT] <>
5 {/manager/*} [] <dex-mgr>
6 {/partner/*} [] <dex-mgr>
7 {/partner/edi/*} [GET, DELETE] <dex-tp>
```

On the one hand $\hat{\rho}_1(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{GET}) = \{\mathtt{M}, \mathtt{P}\}$, because the constraint of line 3 applies, and $\hat{\rho}_2(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{GET}) = \{\mathtt{P}\}$, because of the new constraint of line 7. Hence, $\hat{\rho}_2(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{GET}) \leq_R \hat{\rho}_1(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{GET})$. On the other hand, while $\hat{\rho}_1(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{POST}) = \{\mathtt{M}, \mathtt{P}\}$, we have that $\hat{\rho}_2(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{POST}) = \top$, because no constraint applies. Hence $\hat{\rho}_1(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{POST}) \leq_R \hat{\rho}_2(\langle \mathtt{p}, \mathtt{e} \rangle, \mathtt{POST})$.

Therefore, according to Definition 16, neither $t_1 \leq_T t_2$ nor $t_2 \leq_T t_1$ hold. As a consequence, by Proposition 3, we conclude that the change in the configuration did not yield a more restrictive policy, in contrast to the intuitive expectation of ACME's administrators. The new policy is in fact, at the same time, both more permissive and more restrictive, hence not comparable according to $\leq_T$.

4.5 Implementation and Evaluation

The implementation of the WACT model, defined in Section 4.2, rests on a *trie* data structure, that is a prefix-ordered tree where the descendants of every node share a common prefix, which constitutes a natural representation of URLs. Our prototype contains an implementation of the $[\![\cdot]\!]_{SCL}$ interpretation function, compiling security constraints into a WACT, as well as the decision function δ and the algorithm to compute the partial order $\leq_T$ described in Sections 4.3 and 4.4 respectively.

To validate the correctness of our interpretation of the JEE Servlet Specification, we conducted an experiment aimed at comparing our formal semantics to the operational one of different JEE application servers that implement the specification, through automated software testing.

The flow diagram shown in Figure 4.6 illustrates the testing procedure. We first generate a set of different configurations of security constraints (1) by exploring all the main combinations of constructs allowed by the grammar (cf. Figure 4.2). This is done by the GENSCSET procedure which can be found in Appendix C.1. Based on the grammar, this procedure generates security constraint configurations from the finite input sets of URLs $\mathcal{U}$, HTTP methods $\mathcal{M}$ and roles $\mathcal{R}$, ignoring repetitions due to the mutual re-ordering of the XML constructs. Next, for every configuration c, we instrument the JEE container under scrutiny by deploying a web application that includes c within its deployment descriptor (2a). At the same time, the configuration is interpreted according to the formal semantics (2b), yielding a WACT t with a corresponding decision function δ. For every triple $\langle u, m, R \rangle$, where $u \in \mathcal{U}$ is a URL, $m \in \mathcal{M}$ a method and R is an element of the role lattice $\mathcal{L} = 2^{\mathcal{R}} \cup \{\top\}$, we then issue an HTTP request to the application server (3) for the pair u, m from a user that is granted (4) precisely all the roles in R, or from an unauthenticated user in case $R = \top$. Finally, we use the decision function associated with the WACT as an oracle to test the behaviour of the container: if the HTTP code of the server response is not consistent with the value of $\delta(u, m, R)$,

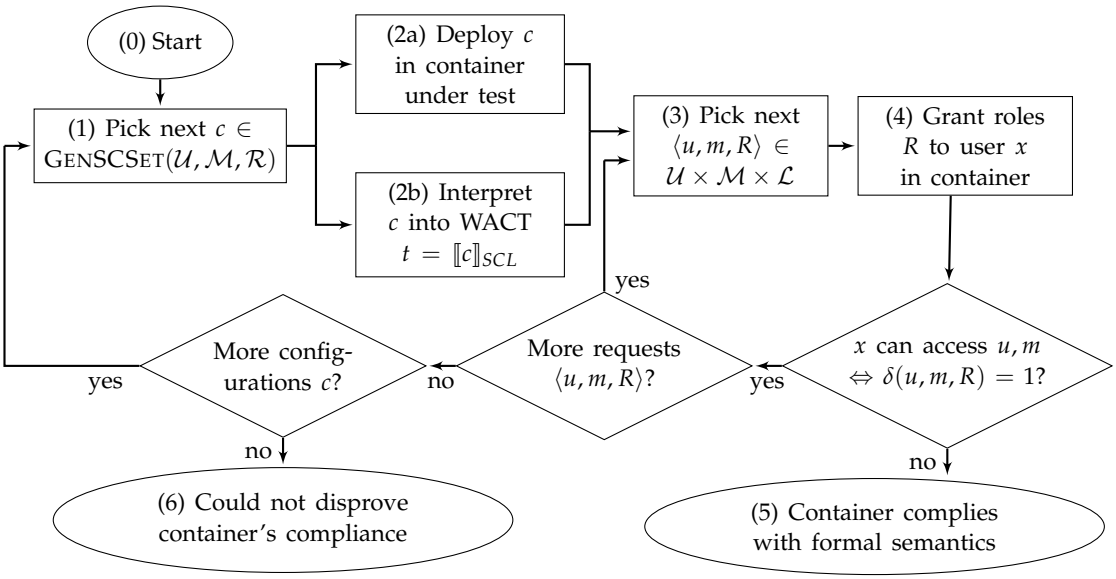


Figure 4.6: Compliance test for JEE containers.

we conclude that the container is not compliant with the formal semantics (5). Note that this could mean that either the container's implementation or our model misinterpreted the specification. If we could not find any discrepancy, we conclude that the test case was not able to disprove the container's compliance (6).

We conducted our experiments on Apache Tomcat versions 6.0.35 and 7.0.29 and Oracle Glassfish version 3.1.2, which are popular JEE application servers implementing the Servlet specification that are widely employed in productive environments. As reported in Table 4.7, we varied the values of the URL, method and role domains (input to the GenSCSet procedure) in order to generate different test cases (one for each line in the table) that explore interesting corner-cases of the language, e.g., overlapping URL patterns with or without wildcards.

The results provide evidence that the implementations did not comply with the formal specification for several tested configurations: Table 4.7 shows, for each test case, the number of configurations generated by GenSCSet and for how many of those the containers' behaviour was not as expected. For what concerns Glassfish, we noticed that all the configurations producing a misbehaviour follow a common pattern, where one or more constraints apply to the context root "/" while other constraints are at the same time defined over the URL pattern "/$\star$". An example of such configurations is given by the pair of security constraints in Figure 4.8a, in which case Glassfish grants any user access to the URL "/", while it should be denied being the constraint of line 2 more specific. Note that this faulty behaviour was not verified in Tomcat, which,

Domains			No. of generated configurations	No. of discrepancies	
$\mathcal{U}$	$\mathcal{M}$	$\mathcal{R}$		Tomcat	Glassfish
{/*,/a}	{GET,POST}	{r1,r2}	9260	0	0
{/*,/a/*}	{GET,POST}	{r1,r2}	9260	152	0
{/,/*,/a}	∅	{r1}	16383	0	5080

Table 4.7: Test results for Apache Tomcat v6.0.35 and Oracle Glassfish v3.1.2

however, did not appear fully compliant to the specification either. More precisely, we found discrepancies in Tomcat's interpretation of all the configurations where at least two different constraints appear and (i) they are defined over overlapping URL patterns ending in "*", (ii) they name (or imply) different methods and (iii) one of them contains a more specific URL pattern. An example is given by the two security constraint reported in Figure 4.8b. In this case `GET` access to the URL "`/a`" should be granted to anyone, as the more specific constraint on line 2 applies, but only mentions the `POST` method. Tomcat, in contrast, was found to deny access to unauthenticated users, whereas removing the seemingly unrelated constraint on line 1 would reinstate the expected behaviour.

Further investigations revealed that the behaviour of Glassfish is a consequence of additional rules included in the JAAC (Java Authorization Contract for Containers) specification [Monzillo2013], prescribing how security constraints are translated into so-called JAAC policies that are then enforced by a JAAC-compliant container. This specification states (Section 3.1.3.2: "Translating security-constraint Elements") that the URL pattern "/" constitutes a special case in that it is always overridden by the path-prefix pattern "/*" and the empty string pattern "" shall be used instead to specify a security constraint applying only to requests that exactly match to the context root. We argue that this exception should be made explicit in the standard Servlet specification too, in order to uniform the behaviour of those JEE containers, such as Tomcat, that do not implement the JAAC specification with those that do support it, like Glassfish.

Following up on the discrepancies that we found in Tomcat's behaviour led instead to the discovery of an implementation error due to an incorrect initialization of part of the data structure used to represent security constraints in the application server. **We reported the issue to the developers who issued a patch that fixes the error** [ASF2012] and that has been included in the Tomcat official distribution as of release 7.0.30.

```
1  {/*} []
2  {/} [] <>
```

(a) Example discrepancy in Glassfish 3.1.2.

```
1  {/*} [] <r2>
2  {/*, /a/*} [POST] <r1>
```

(b) Example discrepancy in Tomcat 6.0.35 and 7.0.29.

Figure 4.8: Example configurations for which JEE containers do not comply with the formal semantics.

4.6 Related Work

Many proposals dealing with the formalization of industry standards can be found in literature. A prominent example, concerning access control, is given by XACML [OASIS2003], a standard for specifying and enforcing access control policies. Because of its generality and high degree of expressiveness, it is able to capture a broad class of access control requirements. However, XACML is quite a complex policy language with informal evaluation semantics, so the development of tools complementing testing with formal verification of XACML is difficult. To tackle this issue, different formal semantics have been given to core concepts of XACML using for instance process algebra [Bryans2005], description logics [Kolovski2007], answer set programming [Ahn2010], specific algebraic variety [Ni2009] or *ad hoc* compositional semantics [Ramli2011].

It is tempting to translate JEE security constraints into XACML and then rely on cited works to benefit from a formal semantics. Unfortunately, some of the selected subsets of the XACML language are incomparable and it seems there is no consensual agreement on its formal semantics, see related work of [Ramli2011] for discussion and examples. Moreover, we argue that a direct semantics for JEE security constraints from its specification without intermediate rewriting provides valuable insights to the policy developers.

Instead of working on a concrete language like XACML suffering from a lack of formal foundations, researchers have proposed access control languages with formal semantics. Several models have been proposed for specific domain of web services. For instance, in [Bertino2006] the authors provided a model with identity attributes and service negotiation capabilities as key features. Attribute-based models remove the subject identification constraint in access control by allowing to specify *who* can access a resource by means of attributes the subject must have [Yuan2005; Crampton2012a]. Such an approach is particularly well suited to open environments where the set of all subjects cannot be known in advance. Those works are valuable as both sources of inspiration for new features and theoretical foundations for next versions of the JEE standard.

In this chapter we considered another challenge: in order to provide formal verification tools for concrete problems of querying and comparison, we do not design a language from scratch and give its formal semantics *a priori*, instead we analyse an existing language and give its semantics *a posteriori*. As the semantics of JEE security constraints is quite specific, it is not clear whether the language can be translated into another one or not. For instance, the Malgrave System [Fisler2005] is a powerful change impact assessment tool based on a restricted sub-language of XACML. However, hierarchical resources such as URLs, which are the core of JEE security constraints and very common in web oriented models, are not supported.

Related work on JEE access control configuration analysis [Naumovich2004; Sun2008] share some of our motivations concerning, e.g., the likelihood that configuration authors are prone to commit mistakes, which leads to the need of automated analysis tools. However, these approaches rather focus on checking the consistency of *programmatic* access control with respect to the implementation of JEE components of the business tier [Naumovich2004] or both business and web tiers [Sun2008], in order, e.g., to detect accesses to EJB fields or methods inconsistent with the access control policy. Our work focusing on *declarative* security is complementary: our formalization supports other reasoning tasks, such as the comparison of different configurations.

4.7 Discussion

In this chapter we considered the version 2.4 of the Servlet specification [Coward2003]. However, recently, a new major release (3.x) has been released [Chan2013]. In this latest revision, configuration authors are allowed to explicitly *omit*, i.e., deny access to selected HTTP methods in a security constraint. Intuitively, assuming the set of HTTP methods to be finite, we argue that there exists an equivalent rewriting for such configurations towards the ones considered in this chapter, where selective negation on methods is implemented through complement. Although this suffices to interpret the security constraints of the new specification directly on our model without loss of generality, it would be nevertheless interesting to extend the model to incorporate explicit prohibitions. Another useful extension would be to allow to explicitly state the default access policy in the input configuration language, in order to cope with different access control systems for the web that do not exhibit an implicit allow-by-default behaviour. For instance, the Apache web server allows to specify, for each authorization rule in the configuration, whether the default policy shall be allow or deny. These extensions would go towards developing a formal role-based access control model for hierarchical resources tailored to web applications. Having such a common formal interpretation structure would allow, for example, to automatically compare or translate access control configurations among different web application frameworks.

In Section 4.5 we proposed to validate our model against existing JEE containers by comparing the respective interpretations of automatically-generated configurations. We tested about 35000 configurations combining security constraints on overlapping URL patterns with and without wildcards. Of course, as the entire space of possible configurations is infinite, our testing methodology cannot be exhaustive. However, larger experiments could be performed to gain increasing assurance on both the model's and containers' correctness. To do so, as the number of generated configurations grows exponentially with the size of URL, method and role domains, we believe that some heuristic has to be developed to select smaller but interesting subsets of configurations for testing. Alternatively, we could introduce additional assumptions on the behaviour of JEE containers, under which a finite number of tests would be enough to guarantee full coverage. For instance, in our experiment we only assumed that JEE containers are insensitive to the mutual ordering of XML tags in the configuration. However, introducing additional hypotheses of regularity, e.g., assuming that the behaviour of the system under test can be described inductively with respect to the hierarchy of URLs, could allow to obtain 100% coverage by testing only a finite set of base cases.

The technique we proposed in this chapter performs static analysis of concrete configurations expressed in a specific language. By following a comparable approach, it is possible to provide formal semantics for the access control configurations of a large variety of common components of distributed information systems, e.g., web, mail, directory, database servers, but also network devices like firewalls. Although such access control systems differ substantially in the structure of the resources they handle and in the expressiveness of access rules, in many cases they ultimately abstract to implementations of particular decision functions [Crampton2012b; Ramli2011], such as the function δ that we introduced for WACTs in Section 4.3 (Equation 4.11). The problem of modeling the interactions among the behaviour of several such access control systems cooperating in the same environment is not trivial and yet largely unexplored. In the next chapter we will tackle this problem in order to leverage such interactions to support the refactoring of distributed access control policies.

4.8 Synthesis

In this chapter we proposed a formal framework able to effectively capture the semantics of the declarative security fragment of the JEE Servlet Specification and efficiently supporting the comparison of policies with respect to their permissiveness.

We equipped the language of security constraints, defined in the Servlet specification, with a formal, set-theoretic interpretation structure. We highlighted key capabilities of this structure, namely answering to access control requests and comparing the permissiveness of security constraints. Such tools can help system administrators to

increase the security of web applications by analyzing the impact of misconfigurations or to prevent security vulnerabilities due to uninformed configuration changes.

In order to validate our interpretation of the Servlet specification, we compared the behaviour of two major existing JEE container implementations with an oracle based on our formal semantics. We showed that generating test configurations from a relatively small number of resources (up to three different URLs and up to two methods and roles) was sufficient to detect discrepancies. Since we could not find any configuration for which our interpretation disagrees at the same time with both the containers under test, we concluded that the formal semantics is correct for the tested configurations. This was supported by the discovery of an bug in Tomcat and of an inconsistency between different JEE specifications (namely Servlet versus JAAC) implemented in Glassfish.

This contribution has been partially integrated in the PoSecCo's focal prototype "Audit Interface" [Bettan2012] in order to provide semantics-aware assessment capabilities. Moreover, it has been published in the proceedings of an international conference [Casalino2012c], in a book chapter [Basile2013a] and in another PoSecCo deliverable [Basile2013b].

Predictability: Does the flap of a butterfly's wings in Brazil set off a tornado in Texas?

—Edward Lorenz, title of paper presented at the 139th "Annual Meeting of the American Association for the Advancement of Science" (29 Dec 1979)

5

Multi-Layered Access Control Policy Refactoring

▷ *In a distributed system, a change in one component's configuration may affect other components' behaviour. While this issue has been investigated in the domains of both network and application-layer policy composition and conflict detection separately, the treatment of inter-layer interactions is still considered an open problem. Such interactions are a consequence of access control systems supporting policies that span over multiple architectural layers: Web servers, for instance, often support access control on the basis of network-layer (IP) addresses other than application-layer (HTTP) fields. The resulting flexibility comes, however, with increased management complexity and the risk of granting unnecessary privileges due to the lack of global knowledge when authoring local policies in isolation.*

To tackle this problem, we propose a technique to perform multi-layered policy refactoring, i.e., to rewrite a collection of access control policies belonging to different architectural layers such that: (i) the global permissiveness is preserved, (ii) the least privilege principle is enforced and (iii) superfluous inter-layer interactions are removed. To this end, we embed a generic access control system into a structure that keeps track of the interactions among authorization decisions taken on different layers. We then define the semantics of composition of such access control layers and show that its inverse, namely decomposition, provides (when it exists) a solution to the problem of refactoring. Finally, we provide algorithms to test for decomposability, as well as to compute (de)composition, that work on a constraint-based relational representation of access control policies. Our main theoretical result is the proof of correctness of the decomposability condition for access control layers, which leverages and extends existing results in database dependency theory, and provides novel evidence that the study of database dependencies can be fruitfully applied to help solve security problems. To assess the feasibility of our approach, we evaluate the algorithms with respect to various properties of input policies. The results show comparable performances with previous work on network security configuration analysis. ◁

Chapter Outline

In the previous chapter we focused on the management of configuration changes for a specific access control system. In this chapter we consider the additional challenges that arise in a distributed system, where a change in one component's configuration can affect the behaviour of other components.

The analysis of distributed access control policies and configurations is a largely-explored research topic. Many existing approaches are tailored to a specific category of system components that handle access control policies with a fixed and uniform semantics. For instance, network-layer conflict analysis [AlShaer2005] focuses on the interactions among distributed policies that filter on the basis of IP addresses and TCP/UDP ports; application-layer conflict analysis [Lupu1999; Davy2008a] and policy composition [Bonatti2002; Wijesekera2003] instead reason on the interaction among access control systems based on the subject-object-action paradigm. In contrast, modeling and reasoning on the interactions between such different architectural layers is still considered a rather open and challenging problem [Sloman2002; AlShaer2011].

The inter-layer overlap among access control policies is indeed more and more common in practice: for instance, many common applications, such as database, mail or Web servers, can constrain access based on clients' IP addresses, modern firewalls can inspect application-layer request fields as well. While such inter-layer relationships can be leveraged to increase the expressiveness of access control policies, it is hard to fully exploit them when authoring the different policies individually, because all other layers' policies shall be taken into account. Furthermore, it may be the case that the same behaviour expressed by the collection of access control mechanisms of all layers can be as well expressed by simpler policies where the inter-layer overlap is minimized and the separation of concerns is increased.

We characterize this problem as the **inter-layer refactoring of access control policies**, i.e., the task of finding the least permissive rewriting of a collection of policies that belong to different layers such that the global composite policy remains equivalent. Policy refactoring is a means (i) to enforce the least privilege principle in multi-layered policy-based access control systems, (ii) to reduce management overhead by simplifying local policies and (iii) to adapt to changing security capabilities of single components.

Example 5.1: Refactoring

In order to illustrate refactoring, we consider the ACME scenario introduced in Chapter 2. Any access from the Internet to ACME's network is mediated by a firewall performing network-layer filtering and it is further regulated by a Web server that acts as a reverse proxy and that filters both on the application (URLs) and network (clients' IP address) layers. If the policies of the two devices are written independently, some unnecessary privileges may be granted by either of them. For instance, the firewall policy may be granting access

to a larger portion of clients than actually allowed by the Web server policy or vice versa. Through refactoring, we aim at exploiting the knowledge of the inter-layer interactions to reduce such privileges to the minimum, by preserving the composite policy. As such, unauthorized access attempts are blocked as soon as possible, according to the least privilege principle.

Suppose now that we are interested in replacing the Web server policy by a simpler one that does not discriminate clients' IP addresses while keeping the semantics of the global composite policy unaltered. Intuitively, to do so we would need to transfer part of the Web server policy to the firewall. Whether such a decomposition is possible, depends both on the internal structure of the original policies and on the access control capabilities of the devices. For instance, if the Web server policy prevents a given IP address from accessing only some specific URLs, the firewall cannot enforce such a policy on its own unless it is capable of HTTP header inspection. In this case refactoring consists in first determining if the new desired access control layer's layout can enforce the global policy and then finding how the original policies are to be rewritten.

The structure of this chapter is the following: In Section 5.1 we define a model that captures the access control behavior of a collection of policy decision points that cooperate on different layers of the same IT infrastructure. In Section 5.2 we define composition of inter-dependent policies as the operation that, given a pair of access control decision functions, produces a composite decision function, decomposition being its inverse. In Section 5.3 we provide an intensional representation for our model based on which we devise algorithms that compute policy (de)composition and that we formally prove correct. In Section 5.4 we identify a criterion inspired from database normalization theory which characterizes precisely when policies can be decomposed and we show how it can be computed on our model. Finally, we show that our proposal is suitable to solve the policy refactoring problem. In Section 5.5 we evaluate the performance of our algorithms on synthetic policies and characterize their behaviour with respect to different statistical properties of input datasets. In Section 5.6 we review and compare our proposal with related work. Section 5.7 provides a technical discussion as well as some theoretic perspectives. Finally, Section 5.8 concludes the chapter with a synthesis of both the contribution and results.

5.1 Access Control Layers

In this section we lay the foundations of a model that captures the access control policy implemented by the collection of policy decision points that operate at different layers within the same IT infrastructure (e.g., firewalls, application servers, Web servers, database servers, etc.). In particular, we aim at characterizing each layer in terms of its access control capabilities and its interface with the other layers.

f	Meaning	$\mathrm{dom}(f)$
$\mathsf{I_s}$	IP source address	Integers range $[0, 2^{32}-1]$
$\mathsf{I_d}$	IP destination address	Integers range $[0, 2^{32}-1]$
$\mathsf{P_s}$	Port source	Integers range $[0, 2^{16}-1]$
$\mathsf{P_d}$	Port destination	Integers range $[0, 2^{16}-1]$
H	HTTP Host Header	Dot-separated strings
U	URL of HTTP Requests	Strings complying with rfc1738
$\sharp$	Singleton field	$\{\square\}$

Table 5.1: Example of Fields and Related Domains

We rely on a classic and general description of access control systems, where a logical subsystem (usually called *policy decision point*) associates, for a given policy, a unique *decision* to any possible *access control request* [Crampton2012b; Ramli2011].

Once we come to reason about the composition of layers, we need to consider relations between the different types of requests they handle. For instance, the IP and port destination fields of the requests handled by a firewall are related to the IP addresses and ports of available services (e.g., Web and application servers). Intuitively, a particular firewall, depending on its policy, either enables requests to be further processed by other policy decision points or blocks them right away. Keeping track of these relationships allows to determine how decisions taken by one layer's policy decision points influence the ones taken in other layers. In order to formalize the above concepts we start from access control request fields and types.

Definition 17 (Request Field and Field's Domain)**.** *The finite set $\mathfrak{F}$ is the universe of all request fields. Each field $f \in \mathfrak{F}$ has a corresponding domain, written $\mathrm{dom}(f)$, that is the set of all possible values f can take in a request. There exists a total order on fields, denoted $\preceq$.*

Each set of fields identifies a particular type of access control requests, as stated in the next definition.

Definition 18 (Request Type and Request Space)**.** *A request type is a finite subset of request fields $F \in 2^{\mathfrak{F}}$. The request space $\mathfrak{Q}(F)$ associated with a request type $F = \{f_i\}_{i=1}^{n}$ characterizes all the requests existing over F. It is the Cartesian product of the domains of the fields in F, taken according to $\preceq$: $\mathfrak{Q}(F) = \mathrm{dom}(f_1) \times \ldots \times \mathrm{dom}(f_n)$., with $f_i \preceq f_j$ for $i \leq j$. The empty request space is the singleton $\mathfrak{Q}(\emptyset) = \{\square\}$.*

Example 5.2: Request Fields and Types

Table 5.1 presents some example request fields which we will refer to throughout the chapter, together with their respective domains. The purpose of the special field $\sharp$, associated to

the fictitious singleton domain $\mathrm{dom}(\sharp) = \{\square\}$, is to characterize the bottom of the network stack.

Example request types are $F_{fw} = \{\mathsf{I_s}, \mathsf{I_d}, \mathsf{P_s}, \mathsf{P_d}, \mathsf{T_p}\}$, which characterizes requests handled by firewalls, or $F_{ws} = \{\mathsf{H}, \mathsf{U}\}$ representing application-layer requests for a Web server capable of filtering on the HTTP host and URL fields.

Other combinations are possible too: for instance $F_{ws} \cup \{\mathsf{I_s}\}$ represents the type of requests handled by a Web server capable of filtering on the clients' IP address, whereas $F_{fw} \cup \{\mathsf{H}, \mathsf{U}\}$ models the request type of a firewall that can inspect parts of the HTTP header.

The definition of access control request follows directly from those of request type and request space. We define two operations on requests: concatenation and projection.

Definition 19 (Access Control Request)**.** *An access control request of type $F = \{f_i\}_{i=1}^n$ is an element of the request space $\mathfrak{Q}(F)$. The requests belonging to $\mathfrak{Q}(F)$ are all the possible sequences $\langle v_1, \ldots, v_n \rangle$ with $v_i \in \mathrm{dom}(f_i)$. The ith coordinate of $q \in \mathfrak{Q}(F)$ is written $q(f_i) = v_i$.*

Given the requests q_1 and q_2 having disjoint request types F_1, F_2, their concatenation is the request $q_1 + q_2$ (also denoted $q_1 q_2$) such that, $\forall f \in F_1 \cup F_2$, $(q_1 + q_2)(f) = q_j(f)$ if $f \in F_j$ (with $j \in \{1, 2\}$).

Given a request q, its projection on some subset P of its request type is denoted by $q|_P$ and it is the restriction of the sequence q to the fields in $P = \{p_1, \ldots, p_n\}$: it is defined by $q|_P = \langle q(p_1), \ldots, q(p_n) \rangle$, with $p_i \preceq p_j$ for $i \leq j$.

We are now ready to provide a formal description of access control layers. An access control layer represents a collection of policy decision points that are all capable of processing access control requests of the same type. It conveys essentially the following three pieces of information:

- the type of access control requests that are in the layer's scope, i.e., those which the layer's decision points are deputed to express a decision for;
- how the request type handled locally relates to that of requests handled within other layers;
- which decision is taken, for every request, by any decision point in the layer according to its policy.

Definition 20 (Access Control Layer)**.** *An Access Control Layer (ACL) is a triple $\langle F, C, \delta \rangle$ where:*

- *$F \in 2^{\mathfrak{F}}$ is the layer request type;*
- *$C \in 2^{\mathfrak{F}} \setminus \{\emptyset\}$ such that $C \cap F = \emptyset$, is the (bottom) layer coupling type;*
- *$\delta : \mathfrak{Q}(C) \to (\mathfrak{Q}(F) \to D)$ is the access decision function with D the set of decisions.*

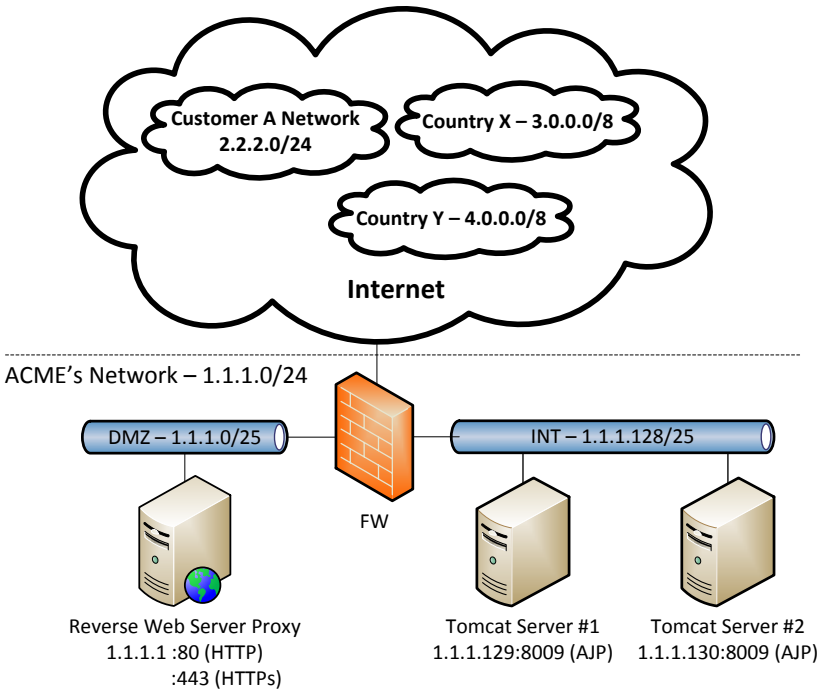


Figure 5.2: Part of ACME's network topology.

The coupling type C relates the ACL with the request type of the layer which lies below it in the stack. In particular, it specifies which fields of the lower layer requests are necessary to uniquely identify a policy decision point in the current layer. For example, a Web server would "couple" with a firewall on the destination IP and port fields, as they are both necessary and sufficient to determine which Web server the requests going through the firewall are directed to, hence its coupling type is the set $\{\mathsf{I_d}, \mathsf{P_d}\}$. Every value in the coupling space $c \in \mathfrak{Q}(C)$ identifies a policy decision point in the ACL, e.g, the pair $c_{ws,443} = \langle 1.1.1.1, 443 \rangle$ represents a Web server listening on the IP address 1.1.1.1 and port 443. The function $\delta(c) : \mathfrak{Q}(F) \to D$, mapping every request $q \in \mathfrak{Q}(F)$ to a unique decision, represents the policy of the policy decision point identified by c.

Not fixing a particular set of decisions in Definition 20 gives us some flexibility to model different aspects of reality. This is illustrated in the next example by making undefined behavior explicit, which leads to a form of partial knowledge reasoning. We argue that this eases the applicability of our approach to real world scenarios, where, even if it is not always possible to model every detail of the system, we still want to be able to drive consistent and insightful conclusions.

Example 5.3: Access Control Layers

In this example we model the system composed by the firewall and the reverse proxy of ACME's scenario (cf. Chapter 2, page 28). In Figure 5.2 we reported an excerpt of the topology of ACME's network. The firewall FW protects the access to the DMZ 1.1.1.0/25 where the reverse proxy is located, being implemented by a Web server WS listening on the IP address 1.1.1.1 and TCP ports 80 and 443.

While FW filters on network fields only, WS supports access control based on both URLs and IP addresses. Hence, the former belongs to the ACL $L_a = \langle \{\mathsf{I_s}, \mathsf{I_d}, \mathsf{P_s}, \mathsf{P_d}\}, \{\sharp\}, \delta_a \rangle$, whereas the latter belongs to $L_b = \langle \{\mathsf{I_s}, \mathsf{H}, \mathsf{U}\}, \{\mathsf{I_d}, \mathsf{P_d}\}, \delta_b \rangle$. The decision functions δ_a and δ_b, encoding the access control policies of the two devices, are reported in Tables 5.3 and 5.4 respectively. Each row in the tables maps all the requests matching to the wildcards to the decision reported in the last column. The set difference symbol "\" represents exceptions (e.g., $* \setminus$ 4.0.0.0/8 means every IP address except the subnet 4.0.0.0/8). The ellipsis "..." maps all the requests that do not match any other row to a "default" decision. These tables are such that it is never the case that a request matches more than one row, hence there is no ambiguity in this example and the mutual order of rows is irrelevant. Note also that the graph of decision functions is in principle huge or even infinite (depending on the fields' domains). We will deal with this issue in Section 5.3 where we will formalize the intuition behind the shorthand tabular notation introduced here.

Only a single policy decision point exists in the ACL L_a, namely the firewall FW. Its policy δ_a allows any client to reach the reverse proxy WS (IP address 1.1.1.1, TCP ports 80 and 443) which in turn can reach the AJP connectors of the backend application servers (IP addresses 1.1.1.129 and 1.1.1.130, TCP port 8009). The policy decision points within L_b are uniquely identified by pairs in the coupling space $\mathfrak{Q}(\{\mathsf{I_d}, \mathsf{P_d}\}) = \mathrm{dom}(\mathsf{I_d}) \times \mathrm{dom}(\mathsf{P_d})$. Only one such decision point is known within the DMZ, namely the reverse proxy WS identified by the pairs $c_{ws,80} = \langle 1.1.1.1, 80 \rangle$ and $c_{ws,443} = \langle 1.1.1.1, 443 \rangle$. Hence, $\delta_b(c_{ws,80})$ and $\delta_b(c_{ws,443})$ are both completely definite functions that represents the policy of WS for, respectively, HTTP and HTTPs requests. The HTTPs policy is the one introduced in Chapter 2: customer A denies access to any partner from country Y (4.0.0.0/8) and allows managers to connect exclusively from its own network (2.2.2.0/24), while customers B and C only accept incoming connections from country X (3.0.0.0/8). The HTTP policy instead only allows access to the root URL, where clients are properly redirected to the encrypted channel. The remaining part of the decision function δ_b maps instead any other request to $\perp \in D$: $\delta_b(c) = \mathfrak{Q}(\{\mathsf{I_s}, \mathsf{H}, \mathsf{U}\}) \mapsto \perp$ for all $c \notin \{c_{ws,80}, c_{ws,443}\}$, meaning that every other decision point yields an undefined decision for every possible request. This models situations where either the topology is only partially known or some components of the system do not fit in the ACL model, but it is nevertheless important to keep track of their presence. For instance, the pair $\langle 1.1.1.129, 8009 \rangle$ represents a Tomcat AJP connector, which does not correspond to any decision point as there is no associated access control policy; however, accounting for its existence in δ_b will allow us to avoid errors when computing the composition with the lower layer.

More layers could be added on top of L_b to incorporate other categories of policy decision points. For instance, the JEE Web applications running on ACME's infrastructure

I_s	I_d	P_s	P_d	D
1.1.1.0/25	1.1.1.0/25	$*$	$*$	1
1.1.1.128/25	1.1.1.128/25	$*$	$*$	1
$* \setminus$ 1.1.1.0/25	1.1.1.1	$*$	80	1
$* \setminus$ 1.1.1.0/25	1.1.1.1	$*$	443	1
1.1.1.1	1.1.1.129	$*$	8009	1
1.1.1.1	1.1.1.130	$*$	8009	1
	...			0

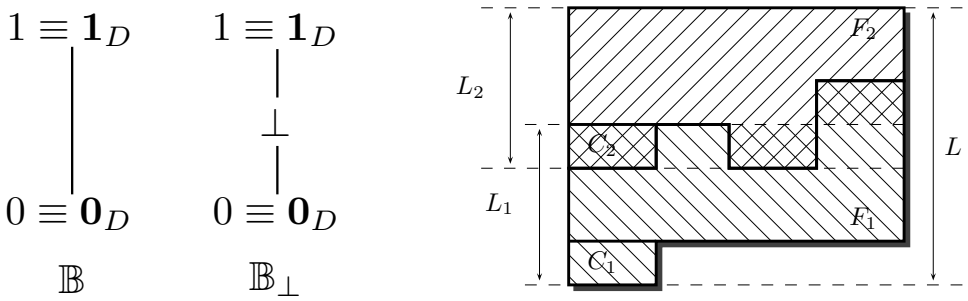
Table 5.3: Example Decision Function δ_a

I_d	P_d	I_s	H	U	D
1.1.1.1	80	$*$	$*$	/	1
			...		0
1.1.1.1	443	$* \setminus$ 4.0.0.0/8	cust-a.acme.com	/partner/$*$	1
		1.1.1.0/24	cust-a.acme.com	/manager/$*$	1
		2.2.2.0/24	cust-a.acme.com	/manager/$*$	1
		1.1.1.0/24	cust-b.acme.com	$*$	1
		3.0.0.0/8	cust-b.acme.com	$*$	1
		1.1.1.0/24	cust-c.acme.com	$*$	1
		3.0.0.0/8	cust-c.acme.com	$*$	1
			...		0
			...		$\perp$

Table 5.4: Example Decision Function δ_b

would be coupled to a specific virtual host of the Web server and would perform access control based on application-specific fields, such as users, roles and URLs. However, for the sake of conciseness, in this chapter we limit the scope of our examples to layers L_a and L_b.

Notice the use of the fictitious coupling type $\{\sharp\}$ for L_a to encode that this is the bottom layer of the stack. As the domain of $\sharp$ is a singleton, there can be only a single policy decision point in this layer (in our example, the firewall `FW`). The reason why we do not model multiple firewalls is that the interest in analyzing distributed firewall policies is about intra-layer dependencies, whereas we focus on (and reason about) inter-layer dependencies. We believe that intra-layer reasoning of network filtering policies is an orthogonal problem that, as argued in Section 5.6, has already been subject of related work.



(a) Example Decision Spaces (b) Request Types Overlap

Figure 5.5: Inter-Layer Composition

5.2 Composition and Decomposition

In this section we define composition as a binary operation between ACLs. In order to compose access control layers, we first need a way to combine the decisions yielded by their respective policies. To this end, we equip the set of decisions D with a suitable algebraic structure named *decision space*.

The standard decision space is $\mathbb{B} = \{0, 1\}$ where 0 stands for prohibition and 1 for authorization. Note that in this case the decision function $\delta : \mathfrak{Q}(F) \to \mathbb{B}$ simply tests whether some request $q \in \mathfrak{Q}(F)$ is a member of a set $Q^{Auth} \subseteq \mathfrak{Q}(F)$ of authorized queries. Many existing languages (e.g., [Ramli2011; Bruns2011; Ni2009]) assume that the decision space is larger than $\mathbb{B}$ to include for instance undefined ($\bot$) or conflicting decisions ($\top$) to cope with modular specification of authorization policies. Similarly to [Ramli2011], we equip the decision space with two operators, denoted $\sqcup$ and $\sqcap$, that generalize standard boolean disjunction and conjunction.

Definition 21 (Decision Space)**.** *A decision space is a bounded distributive lattice $\langle D, \sqcup, \sqcap \rangle$, where D is a non empty finite set of decisions and $\sqcup, \sqcap$ are respectively the least upper bound and greatest lower bound operators on D. The top and bottom elements of the lattice are denoted respectively $\mathbf{1}_D$ and $\mathbf{0}_D$.*

Where no ambiguity arises we identify a decision space with its underlying set D. Figure 5.5a shows the Hasse diagrams of the boolean decision space $\mathbb{B}$ and its extension to undefined decisions $\mathbb{B}_\bot$, which we use throughout the rest of the chapter. Note that $\sqcup, \sqcap$ behave exactly like standard boolean disjunction and conjunction for decisions 0 and 1. Undefined decisions constitute an intermediate level of permissiveness, e.g., $\bot \sqcap 1 = \bot$, but $\bot \sqcap 0 = 0$. In fact, the partial order associated with the lattice on D can be interpreted as an order of permissiveness on decisions: for each $x, y \in D$, x is less permissive than y, written $x \leq y$, if and only if $x \sqcap y = x$.

The key to ACL composition is the overlap of request types, because it implies interdependency between decisions taken by different decision functions. Let $L_1 = \langle F_1, C_1, \delta_1 \rangle$ and $L_2 = \langle F_2, C_2, \delta_2 \rangle$ be two ACLs that act in composition, e.g., suppose L_2 is over L_1 in the network stack. Then, every upper layer policy decision point shall match to some lower layer request, hence the upper layer coupling type C_2 is included in the lower layer request type F_1. Furthermore, it may be the case that the two layers' request types have some fields in common (e.g., layers L_a and L_b of Example 5.3 have in common the IP source field $\mathsf{I_s}$). Figure 5.5b depicts this situation, where the double hatched areas highlight the overlap between layers.

The union of the request types of L_1 and L_2 can then be thought as the request type of a new ACL L, that we are going to define as their composition. The decision function of L needs to depend both on δ_1 and δ_2. In case of a boolean decision space, we would expect every lower layer request q_l that agrees with an upper layer request q_u to yield a composite request that is allowed if and only if *both q_l and q_u* are allowed. In the following definition we generalize this intuition by substituting the logic conjunction with the greatest lower bound operator $\sqcap$. The behaviour of $\sqcap$ is consistent with the semantics of composition not only for allow/deny decisions, but for undefined decisions too. Undefined can be interpreted as "either deny or allow", hence its composition with allow shall equal undefined (1 is the identity in the algebra of composition), whereas it shall yield deny when composed with deny (0 is the absorbing element).

Definition 22 (ACL Composition)**.** *Given the ACLs $L_1 = \langle F_1, C_1, \delta_1 \rangle$ and $L_2 = \langle F_2, C_2, \delta_2 \rangle$ such that $C_2 \subseteq F_1$ and $C_1 \cap F_2 = \emptyset$, their composition is the ACL $L_1 \otimes L_2 = \langle F_1 \cup F_2, C_1, \delta \rangle$, where δ is defined as follows:*

$$\begin{aligned} \delta : \mathfrak{Q}(C_1 \cup F_1 \cup F_2) &\to D \\ q &\mapsto \delta_1(q|_{C_1 \cup F_1}) \sqcap \delta_2(q|_{C_2 \cup F_2}). \end{aligned} \tag{5.1}$$

Example 5.4: Composition

We again consider the ACLs $L_a = \langle \{\mathsf{I_s}, \mathsf{I_d}, \mathsf{P_s}, \mathsf{P_d}\}, \{\sharp\}, \delta_a \rangle$ and $L_b = \langle \{\mathsf{I_s}, \mathsf{H}, \mathsf{U}\}, \{\mathsf{I_d}, \mathsf{P_d}\}, \delta_b \rangle$ introduced in Example 5.3. As the coupling type of L_b (resp. L_a) is included in (resp. disjoint from) the request type of L_a (resp. L_b), it follows, by Definition 22, that their composition is defined. This equals $L_a \otimes L_b = L_c = \langle \{\mathsf{I_s}, \mathsf{I_d}, \mathsf{P_s}, \mathsf{P_d}, \mathsf{H}, \mathsf{U}\}, \{\sharp\}, \delta_c \rangle$, where the graph of δ_c is represented in Table 5.6.

For instance, the request $\langle$2.2.2.1, 1.1.1.1, 12345, 443, cust-a.acme.com, /manager/$\rangle$ is authorized in L_c because L_a allows $\langle$2.2.2.1, 1.1.1.1, 12345, 80$\rangle$ and L_b allows $\langle$2.2.2.1, cust-a.acme.com, /manager/$\rangle$. However, the decision related to the request $\langle$1.1.1.1, 1.1.1.129, 12345, 8009, x, y$\rangle$ is $\perp$ for any x, y because $\langle$1.1.1.1, 1.1.1.129, 12345, 8009$\rangle$ is allowed in L_a, but there is no corresponding policy decision point in L_b.

I_s	I_d	P_s	P_d	H	U	D
$*$	1.1.1.1	$*$	80	$*$	/	1
1.1.1.0/25	1.1.1.1	$*$	443	cust-a	/manager/$*$	1
1.1.1.0/25	1.1.1.1	$*$	443	cust-b	$*$	1
1.1.1.0/25	1.1.1.1	$*$	443	cust-c	$*$	1
1.1.1.0/25	1.1.1.1	$*$	$* \setminus \{80, 443\}$	$*$	$*$	$\bot$
1.1.1.0/25	1.1.1.0/25 $\setminus$ {1.1.1.1}	$*$	$*$	$*$	$*$	$\bot$
1.1.1.1	1.1.1.129	$*$	8009	$*$	$*$	$\bot$
1.1.1.1	1.1.1.130	$*$	8009	$*$	$*$	$\bot$
1.1.1.128/25	1.1.1.128/25	$*$	$*$	$*$	$*$	$\bot$
$*\setminus$ 4.0.0.0/8	1.1.1.1	$*$	443	cust-a	/partner/$*$	1
2.2.2.0/24	1.1.1.1	$*$	443	cust-a	/manager/$*$	1
3.0.0.0/8	1.1.1.1	$*$	443	cust-b	$*$	1
3.0.0.0/8	1.1.1.1	$*$	443	cust-c	$*$	1
		...				0

Table 5.6: Example Decision Function δ_c

Given an ACL $L = \langle F, C, \delta \rangle$ and sets F_1, F_2, C_2 such that $F = F_1 \cup F_2$, $C_2 \subseteq F_1$ and $C_2 \cap F_2 = \emptyset$, decomposition is the problem of finding ACLs $L_1 = \langle F_1, C, \delta_1 \rangle$ and $L_2 = \langle F_2, C_2, \delta_2 \rangle$ such that $L_1 \otimes L_2 = L$. As the request and coupling types of the candidate decomposition are given, the problem amounts to computing δ_1 and δ_2 from δ. For instance, for every request $q' \in \mathfrak{Q}(F_1 \cup C_1)$, we want to compute $\delta_1(q')$ from all the values $\delta(q)$ corresponding to the requests $q \in \mathfrak{Q}(F)$ that concern the layer L_1, i.e., such that $q|_{F_1 \cup C_1} = q'$. The same reasoning applies symmetrically for computing δ_2.

In the case of a boolean decision space, the natural semantics we would like to assign to such an operation is that of *projection*. For instance, let $\delta : \mathfrak{Q}(F) \to \mathbb{B}$. Its projection on $P \subseteq F$ would be $\pi_P(\delta) : \mathfrak{Q}(P) \to \mathbb{B}$ such that $\pi_P(\delta)(q') = 1$ if and only if $\delta(q) = 1$ for *at least one* $q \in \mathfrak{Q}(F)$ that agrees with q' on all the fields in P. Hence, $\pi_P(\delta)$ would map each q' to the logic disjunction of all the $\delta(q)$ where $q|_{F_1 \cup C_1} = q'$. In the next definition we generalize to larger decision spaces by replacing disjunction with the least upper bound on decisions $\sqcup$.

Definition 23 (Projection)**.** *The projection of a decision function $\delta : \mathfrak{Q}(F) \to D$ over the set of fields $P \subseteq F$ is the decision function $\pi_P(\delta)$ defined as follows:*

$$\begin{aligned} \pi_P(\delta) : \mathfrak{Q}(P) &\to D \\ q &\mapsto \bigsqcup_{x \in \mathfrak{Q}(F \setminus P)} \delta(q + x). \end{aligned} \tag{5.2}$$

As stated in the next proposition, *decompositions* obtained through projection are always the *least permissive* among those that leave the composite decision function unchanged. This result guarantees the least privilege principle for policy refactoring as stated in the introduction of this chapter.

Proposition 4 (Least Privilege Decomposition)**.** *Let* $L = \langle F_1 \cup F_2, C_1, \delta \rangle$ *such that* $L = \langle F_1, C_1, \delta_1 \rangle \otimes \langle F_2, C_2, \delta_2 \rangle$*. Then,* $\forall q \in \mathfrak{Q}(F_i \cup C_i)$*,* $\pi_{F_i \cup C_i}(\delta)(q) \leq \delta_i(q)$ *for* $i \in \{1, 2\}$*.*

Proof. See Appendix B.2. □

5.3 Intensional Representation

The request spaces we considered so far are, in general, infinite or very large in cardinality. This follows from the fact that each field can have either an infinite (e.g. URLs) or a very large (e.g. IP addresses) domain of values. In order to cope with this issue we introduce a finite and compact representation for generic request spaces and decision functions. We then show how (de)composition can be computed on instances of such a representation.

5.3.1 Finite descriptors for decision functions

Every policy-based access control system provides administrators with a configuration language whose expressiveness is tailored to the domain of access control policies for such a system. For instance, the access control language of a web server will likely exploit the hierarchical order of URLs to allow constraining access on an entire subtree of resources with a single rule. As we aim at integrating policies from different domains, we want our approach to be independent from the different domain-specific policy languages. To this end, we will identify a class of languages having sufficient properties to allow defining simple and generic (de)composition procedures.

We take inspiration from constraint database theory [Revesz1995], where database relations are represented *in intenso*: each tuple is not a sequence of atomic values but a sequence of subsets of values described by constraints. We use the same idea to encode the graph of decision functions. The basic building block is a constraint language suitable to describe subsets of a field's domain, which we name *field descriptor*.

Definition 24 (Field Descriptor)**.** *Given a request field* $f \in \mathfrak{F}$*, a field descriptor for* f *is a structure* $\langle \Phi_f, [\![\cdot]\!]_f \rangle$ *where* Φ_f *is a language that allows to describe sets of elements in the domain of* f *and* $[\![\cdot]\!]_f : \Phi_f \to 2^{\mathrm{dom}(f)}$ *is an interpretation function that maps every sentence of the language to its extension, i.e., the subset of the field's domain it describes.*

Furthermore, the language Φ_f is assumed to be closed under the intersection of sentences' extensions, namely $\forall \varphi_1, \varphi_2 \in \Phi_f$, $\exists \varphi_3 \in \Phi_f$ s.t. $[\![\varphi_3]\!]_f = [\![\varphi_1]\!]_f \cap [\![\varphi_2]\!]_f$. For every such combination we call φ_3 the conjunction of φ_1, φ_2, written $\varphi_3 = \varphi_1 \wedge \varphi_2$.

Example 5.5: Field Descriptor of Integer Intervals

To illustrate the concept, we consider a field descriptor that allows to describe intervals of positive integers. A bounded interval is described by a pair of integers representing its minimum and maximum values; an unbounded one has its maximum value set to ∞. The conjunction of two intervals is the (potentially empty) interval ranging from the maximum of their lower boundaries to the minimum of their upper ones. E.g., if $\varphi_1 = [0, 100], \varphi_2 = [20, \infty]$ and $\varphi_3 = [200, 300]$, we have $\varphi_1 \wedge \varphi_2 = [20, 100]$ and $[\![\varphi_1 \wedge \varphi_3]\!] = \emptyset$, $[\![\varphi_1]\!] = \{0, 1, \ldots, 100\}$, $[\![\varphi_1 \wedge \varphi_2]\!] = \{20, 21, \ldots, 100\}$.

This descriptor is suitable for representing many of the fields introduced in Table 5.1 (i.e., $\mathsf{I_s}, \mathsf{I_d}, \mathsf{P_s}, \mathsf{P_d}$). For fields associated with large or infinite domains of strings (such as U) we could analogously define a field descriptor where the sentences in Φ are arbitrary regular expressions, and their conjunction is the intersection of the corresponding automata.

When we arrange together a collection of field descriptors, we obtain an object suitable to describe sets of requests. This is formalized in the following definition, which generalizes Definitions 18 and 19 to the language of field descriptors.

Definition 25 (Requests Descriptor)**.** *Let F be a set of fields and, for every field $f \in F$, let $\langle \Phi_f, [\![\cdot]\!]_f \rangle$ be an associated field descriptor. We then define the requests descriptors space on $F = \{f_i\}_{i=1}^n$ as the product of all the languages Φ_{f_i}: $\Psi(F) = \Phi_{f_1} \times \ldots \times \Phi_{f_n}$. Every sequence $\psi = \langle \varphi_1, \ldots, \varphi_n \rangle \in \Psi(F)$ is a Request Descriptor (RD) for the request space $\mathfrak{Q}(F)$.*

The concatenation $\psi + \psi'$ and the projection $\psi|_{P \subseteq F}$ from Definition 19 extend naturally to RDs. Moreover, if $\psi = \langle \varphi_1, \ldots, \varphi_n \rangle, \psi' = \langle \varphi'_1, \ldots, \varphi'_n \rangle$ are RDs on $\mathfrak{Q}(F)$, we define their conjunction as $\psi \wedge \psi' = \langle \varphi_1 \wedge \varphi'_1, \ldots, \varphi_n \wedge \varphi'_n \rangle$.

The extension of a RD ψ, written $[\![\psi]\!]_F$, is the product of the extension of the sentences φ_i. Formally: $[\![\psi]\!]_F = [\![\psi(f_1)]\!]_{f_1} \times \ldots \times [\![\psi(f_n)]\!]_{f_n}$.

We are now in a position to define a finite descriptor for decision functions.

Definition 26 (Decision Function Descriptor)**.** *Given a set of fields F and a decision space D, a Decision Function Descriptor (DFD) is a finite relation $\Delta \subseteq \Psi(F) \times D$ that covers the entire request space $\mathfrak{Q}(F)$.*

The extension of Δ is the decision function $\delta : \mathfrak{Q}(F) \to D$, written $\delta = \mathrm{ext}(\Delta)$, defined as follows:

$$\delta(q) = \bigsqcup \{d \mid \langle \psi, d \rangle \in \Delta \wedge q \in [\![\psi]\!]_F\}, \forall q \in \mathfrak{Q}(F). \tag{5.3}$$

Algorithm DFDCOMP($\langle F_1, C_1, \Delta_1 \rangle, \langle F_2, C_2, \Delta_2 \rangle$)

$W \leftarrow (F_1 \cap F_2) \cup C_2$;
$U \leftarrow (F_1 \cup C_1) \setminus W$;
$V \leftarrow (F_2 \cup C_2) \setminus W$;
$\Delta \leftarrow \emptyset$;
foreach $\langle \psi_1, d_1 \rangle \in \Delta_1, \langle \psi_2, d_2 \rangle \in \Delta_2$ **do**
  if $[\![\psi_1|_W \wedge \psi_2|_W]\!]_W \neq \emptyset$ **then**
    $\psi \leftarrow (\psi_1|_W \wedge \psi_2|_W) + \psi_1|_U + \psi_2|_V$;
    $\Delta \leftarrow \Delta \cup \{\langle \psi, d_1 \sqcap d_2 \rangle\}$;
return $\langle F_1 \cup F_2, C_1, \Delta \rangle$;

Algorithm DFDPROJ(Δ, P)

return $\{\langle \psi|_P, d \rangle \mid \langle \psi, d \rangle \in \Delta\}$;

Figure 5.7: ACL Composition and Projection with DFD

Equation (5.3) associates each DFD with a unique decision function (namely its extension), which can be thought of as its semantics. The DFD extension maps every request q to the least upper bound of all the decisions being associated, in the DFD, with a request descriptor that matches q. Note how, given this semantics, it is not restrictive to require the complete coverage of the entire request space. In fact, this can always be achieved by including in the DFD a *default* RD that (i) matches all the possible requests and (ii) is associated with the decision $\mathbf{0}_D$. Moreover, whenever a concrete policy language features a *deny by default* semantics (as it is typically the case, e.g., for firewalls), the translation of such policies to DFD reduces to computing decisions for all the possible overlaps among rules within the policy.

5.3.2 Computing (de)composition

Figure 5.7 defines two procedures that compute composition and projection on DFDs. The correctness of the algorithms, as stated in Proposition 5, is ensured by showing that the extension of the output DFDs equals the composition (Definition 22), respectively projection (Definition 23), of the input ones.

Proposition 5 (Correctness of DFDCOMP and DFDPROJ)**.** *For every pair* $L_1 = \langle F_1, C_1, \Delta_1 \rangle$, $L_2 = \langle F_2, C_2, \Delta_2 \rangle$, *if* DFDCOMP$(L_1, L_2) = \langle F_1 \cup F_2, C_1, \Delta \rangle$, *then we have* $\langle F_1, C_1, \text{ext}(\Delta_1) \rangle \otimes \langle F_2, C_2, \text{ext}(\Delta_2) \rangle = \langle F_1 \cup F_2, C_1, \text{ext}(\Delta) \rangle$. *Moreover, for every DFD* Δ, *if* DFDPROJ$(\Delta) = \Delta'$, *then* $\pi_P(\text{ext}(\Delta)) = \text{ext}(\Delta')$.

Proof. See Appendix B.2. □

The algorithms' complexity, for constant sets ot fields F_i, C_i ($i \in \{1,2\}$) and input DFDs of size n, is $O(n)$ for DFDPROJ and $O(n^2)$ for DFDCOMP. More precisely, if n_1 and n_2 are the respective sizes of the input DFDs, the cost of computing DFDCOMP is proportional to their product. Each of the $n_1 \times n_2$ iterations has two contributions: (i) the cost of intersecting two RDs restricted to the subset of fields W (line 6) and, if such an intersection is not empty, (ii) the cost of concatenating the RDs and updating the result DFD (lines 7-8). Note that the complexity of the former depends on how many fields are contained in the set W and the latter contributes only if the extensions of the current pair of RDs ψ_1, ψ_2 have a non-empty intersection when restricted to the fields in W. The performance of the algorithm will be, therefore, influenced by both the cardinality of W and the probability that any pair of RDs have a non-empty intersection on W.

In order to perform refactoring, we need to determine whether a decomposition is possible. This is equivalent to checking if a decision function, once projected and composed back, equals itself. This translates into testing the equivalence of $\langle F_1 \cup F_2, C_1, \Delta \rangle$ with $\langle F_1, C_1, \pi_{F_1 \cup C_1}(\Delta) \rangle \otimes \langle F_2, C_2, \pi_{F_2 \cup C_2}(\Delta) \rangle$, which, as different DFDs can have the same extension, requires to compare the (possibly infinite) extensions of their DFDs. In the next section we deal with this issue by developing an alternative criterion to test decomposability.

5.4 Decomposability and Refactoring

Through decomposition, we aim at factorizing the complexity of some layer's policy into simpler ones. This means that the request type of any of the decomposed layers shall be a strict subset of the one of the original (composite) layer. The next result shows that it is not guaranteed that such a decomposition exists for a generic access control layer.

Proposition 6. *For all F_1, F_2, C_1 where $F_1 \cup F_2$ is a request type, C_1 is a coupling type and $F_2 \not\subseteq F_1$ there exists an ACL $L = \langle F_1 \cup F_2, C_1, \delta \rangle$ that cannot be decomposed in any pair of ACLs $\langle F_1, C_1, \delta_1 \rangle$, $\langle F_2, C_2, \delta_2 \rangle$.*

Proof. See Appendix B.2. □

The last proposition can be illustrated by the following counterexample.

Example 5.6: Non-decomposability

Consider the decision function δ_c (Table 5.6) and let

$q_1 = \langle$2.2.2.1, 1.1.1.1, 12345, 443, cust-a.acme.com, /manager/$\rangle$,
$q_2 = \langle$3.0.0.1, 1.1.1.1, 12345, 443, cust-a.acme.com, /manager/$\rangle$ and
$q_3 = \langle$3.0.0.1, 1.1.1.1, 12345, 443, cust-b.acme.com, /$\rangle$.

We can immediately see that the decomposition in the pair of ACLs $\langle \{\mathsf{I_s}, \mathsf{I_d}, \mathsf{P_s}, \mathsf{P_d}\}, \{\sharp\}, \delta_1 \rangle$ and $\langle \{\mathsf{H}, \mathsf{U}\}, \{\mathsf{I_d}, \mathsf{P_d}\}, \delta_2 \rangle$, is not possible. In fact, as $\delta_c(q_2) = 0$, according to the rules of composition (Definition 22), we need either $\delta_1(\langle$3.0.0.1, 1.1.1.1, 12345, 443$\rangle) = \delta_1(x) = 0$ or $\delta_2(\langle$1.1.1.1, 443, cust-a.acme.com, /manager/$\rangle) = \delta_2(y) = 0$. On the other hand, as $\delta_c(q_1) = \delta_c(q_3) = 1$, both $\delta_1(x) = 1$ and $\delta_2(y) = 1$ must hold.

Intuitively, we see that in order to have decomposability, the decisions associated to requests that satisfy a specific inter-field dependency cannot be chosen independently one from another. The next definition formalizes this intuition.

Definition 27 (Inter-Field Dependency)**.** *For W and V non-empty and disjoint subsets of F, we say that a decision function δ satisfies the Inter-Field Dependency (IFD) condition $W \twoheadrightarrow V$, written $\delta \models W \twoheadrightarrow V$,* if and only if $\forall q, q' \in \mathfrak{Q}(F),\ q|_W = q'|_W \Rightarrow \delta(q) \sqcap \delta(q') = \delta(q|_{F\setminus V} + q'|_V) \sqcap \delta(q|_V + q'|_{F\setminus V})$.

We are now ready to state the main result of this section: IFDs precisely characterize when an ACL can be decomposed by projections without loss of permissiveness.

Theorem 1 (Decomposability)**.** *Given a generic ACL $L = \langle F_1 \cup F_2, C_1, \delta \rangle$, the following expressions are equivalent:*

- $L = \langle F_1, C_1, \pi_{F_1 \cup C_1}(\delta) \rangle \otimes \langle F_2, C_2, \pi_{F_2 \cup C_2}(\delta) \rangle$
- $\delta \models (F_1 \cap F_2) \cup C_2 \twoheadrightarrow (F_2 \setminus F_1)$.

Sketch of the proof. (See Appendix B.2 for the full proof). Let $W = (F_1 \cap F_2) \cup C_2$, $U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1$ and $V = F_2 \setminus F_1$ and let $q = wuv$ be a query. The sets W, U and V form a partition of $F_1 \cup F_2 \cup C_1$. For the first half, we need to show that $\pi_{W\cup U}(\delta)(wu) \sqcap \pi_{W\cup V}(\delta)(wv) = \delta(q)$ for all q given that $\delta \models W \twoheadrightarrow V$. The proof amounts to a sequence of equalities involving distributivity and absorption properties of lattices. One of the key equality being the following:

$$\delta(wuv) \sqcup \bigsqcup_{\substack{u' \in \mathfrak{Q}(U)\setminus\{u\} \\ v' \in \mathfrak{Q}(V)\setminus\{v\}}} \Big(\delta(wuv') \sqcap \delta(wu'v) \Big) = \delta(wuv) \tag{5.4}$$

For the second half, we suppose that $\delta(wuv) = \pi_{W\cup U}(\delta)(wu) \sqcap \pi_{W\cup V}(\delta)(wv)$ and that $\delta(wu_1v_1) \sqcap \delta(wu_2v_2) \neq \delta(wu_1v_2) \sqcap \delta(wu_2v_1)$ for some queries. The key is to be able to

$\mathsf{I_s}$	$\mathsf{I_d}$	$\mathsf{P_s}$	$\mathsf{P_d}$	D
$*$	1.1.1.1	$*$	80	1
$*\setminus$ 4.0.0.0/8	1.1.1.1	$*$	443	1
1.1.1.0/25	1.1.1.1	$*$	$*\setminus\{80,443\}$	$\perp$
1.1.1.0/25	$1.1.1.0/25\setminus\{1.1.1.1\}$	$*$	$*$	$\perp$
1.1.1.1	1.1.1.129	$*$	8009	$\perp$
1.1.1.1	1.1.1.130	$*$	8009	$\perp$
1.1.1.128/25	1.1.1.128/25	$*$	$*$	$\perp$
	...			0

Table 5.8: Example Projection $\pi_{\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_s},\mathsf{P_d}\}}(\delta_c)$

derive a series of inequalities of the form $\delta(wu_1v_2)\sqcap\delta(wu_2v_1)\leq\delta(wu_1v_1)$ leading to a contradiction with $\delta\not\models W\twoheadrightarrow V$. □

Theorem 1 gives an alternative criterion to test the decomposability of an ACL: we need to check if its decision function satisfies the IFD $(F_1\cap F_2)\cup C_2\twoheadrightarrow(F_2\setminus F_1)$, for the subsets F_1,F_2,C_2 of its request type (with $C_2\subseteq F_1$) that represent the new layers' layout we want to find a policy for.

Example 5.7: ACL Refactoring

Consider the ACL $L_c=\langle\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_s},\mathsf{P_d},\mathsf{H},\mathsf{U}\},\ \{\sharp\},\ \delta_c\rangle$ that was introduced in Example 5.4. As it is the result of the composition of ACLs $L_a=\langle\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_s},\mathsf{P_d}\},\{\sharp\},\delta_a\rangle$ and $L_b=\langle\{\mathsf{I_s},\mathsf{H},\mathsf{U}\},\{\mathsf{I_d},\mathsf{P_d}\},\delta_b\rangle$, we naturally expect it to be decomposable for the very same request types of L_a and L_b. It can be checked, by inspecting Table 5.6, that indeed $\delta_c\models\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_d}\}\twoheadrightarrow\{\mathsf{H},\mathsf{U}\}$. Hence, if we name $L'_a=\left\langle\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_s},\mathsf{P_d}\},\{\sharp\},\pi_{\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_s},\mathsf{P_d}\}}(\delta_c)\right\rangle$ and $L'_b=\left\langle\{\mathsf{I_s},\mathsf{H},\mathsf{U}\},\{\mathsf{I_d},\mathsf{P_d}\},\pi_{\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_d},\mathsf{H},\mathsf{U}\}}(\delta_c)\right\rangle$, we know by Theorem 1 that $L_c=L'_a\otimes L'_b$. This is an example of refactoring that keeps the request type unchanged.

Moreover, because of Proposition 4, we expect the decomposed policies in L'_a,L'_b to be equally or less permissive (more precisely the least possible permissive that still preserves the equivalence of the composite policy) than the original ones in L_a,L_b. Table 5.8 represents the decision function $\pi_{\{\mathsf{I_s},\mathsf{I_d},\mathsf{P_s},\mathsf{P_d}\}}(\delta_c)$. Notice that it is not in fact equal to the original decision function δ_a of the ACL L_a (cf. Table 5.3). In particular, it is *never more permissive* than the original; on the other hand it is, where possible and according to the least privilege principle, *more restrictive*. For instance, requests coming from the 4.0.0.0/8 IP network (country Y) and directed to 1.1.1.1, which were allowed by the original policy, are denied by the refactored version. This is consistent with the fact that such requests were anyway always denied in L_b (cf. Table 5.4). On the other hand, the decisions for all requests coming from 1.1.1.1 and directed to either 1.1.1.129 or 1.1.1.130 on port 8009, are refactored to $\perp$. This is a consequence of assuming partial knowledge of the L_b policy.

Since additional information is required to decide on the usefulness of those permissions, the choice is left to the user who can either trust the original policy, i.e., change $\perp$ to 1, or follow a more restrictive approach and change $\perp$ to 0. In this example the right choice is of course the former, which allows the reverse proxy to reach the backend application servers as expected.

Let us now try to refactor L_a, L_b to a pair of ACLs that have smaller request types. Suppose, for instance, to substitute the WS Web server of our scenario with one that does not discriminate requests on the basis of the IP source address; this means that the field $\mathsf{I_s}$ does not belong any more to the request type of L''_b. However, as shown in Example 5.6, we know that such a decomposition is not possible. This is because $\delta_c \not\models \{\mathsf{I_d}, \mathsf{P_d}\} \twoheadrightarrow \{\mathsf{H}, \mathsf{U}\}$. Had all the requests with $\mathsf{H} =$ cust-a.acme.com in δ_c been mapped to 0, the IFD would have been instead satisfied. In such a case we would have had a refactoring with a change in request types that simplified the decision function δ_b.

Inter-field dependencies are defined directly on the graph of a decision function, which suggests that they can be as well checked on the corresponding DFD. The IFDCHECK algorithm (Figure 5.9) computes this check and the next proposition ensures its correctness.

Proposition 7 (Correctness of IFDCHECK)**.** *Let $\Delta \subseteq \Psi(F) \times D$ be a DFD and W, V non-empty and disjoint subsets of F. Then,* IFDCHECK(Δ, W, V)=**true** $\Leftrightarrow$ ext $(\Delta) \models W \twoheadrightarrow V$.

Proof. See Appendix B.2. □

The key ideas underlying the IFDCHECK algorithm are as follows. First, we rewrite the RDs contained in Δ to make them partition the entire request space (lines 2–6) such that every request matches exactly one RD $\psi_W + \psi_U + \psi_V$. Second, for every ψ_W, we consider all the pairs ψ_U, ψ'_U and ψ_V, ψ'_V (lines 7, 8) and we compute the greatest lower bound of the decisions associated with all the pairs of requests matching respectively $\psi_W + \psi_U + \psi_V$ and $\psi_W + \psi'_U + \psi'_V$ (line 9). We finally check if the latter equals the greatest lower bound of all the pairs of requests matching $\psi_W + \psi_U + \psi'_V$ and $\psi_W + \psi'_U + \psi_V$ (lines 10–14).

To partition the request space, we iteratively use the PARTITION procedure defined in Figure 5.10 that computes a closure with respect to intersection and difference for the portion of the RDs concerning the subset of fields X. Note that here, unlike for previous algorithms, we need to compute the set of RDs describing the difference between two given RDs, formally DIFF$(\psi_1, \psi_2) = \{\psi_i^*\}$ such that $\bigcup\{[\![\psi_i^*]\!]_F\} = [\![\psi_1]\!]_F \setminus [\![\psi_2]\!]_F$. This is generally possible for the RDs composed of the field descriptors considered in this chapter (e.g., intervals of integers). Figure 5.11 shows an example execution of PARTITION on a DFD composed of two bidimensional RDs, i.e., belonging to the request space $\Psi(\{f_1, f_2\})$. The input DFD, depicted on the left, is the relation $\{\langle \psi_1, a\rangle, \langle \psi_2, b\rangle\} \subseteq \Psi(\{f_1, f_2\}) \times D$ which contains two RDs ψ_1, ψ_2 associated to decisions a and b respectively. We compute the partition of this DFD with respect to the

Algorithm IFDCHECK(Δ, W, V)

$U \leftarrow F \setminus (W \cup V)$;
foreach $\langle \psi_W, \Delta_W \rangle \in$ PARTITION(Δ, W) **do**
  $S \leftarrow \emptyset$;
  foreach $\langle \psi_U, \Delta_U \rangle \in$ PARTITION(Δ_W, U) **do**
    $S \leftarrow S \cup \{\psi_U\}$;
    $\mathcal{P}_{\psi_U} \leftarrow$ PARTITION(Δ_U, V);
  foreach $\psi_U, \psi'_U \in S$ **do**
    foreach $\langle \psi_V, \Delta_V \rangle \in \mathcal{P}_{\psi_U}, \langle \psi'_V, \Delta'_V \rangle \in \mathcal{P}_{\psi'_U}$ **do**
      $d_1 \leftarrow \bigsqcup \{d \mid \langle \cdot, d \rangle \in \Delta_V\} \sqcap \bigsqcup \{d \mid \langle \cdot, d \rangle \in \Delta'_V\}$;
      foreach $\langle \psi''_V, \Delta''_V \rangle \in \mathcal{P}_{\psi_U}, \langle \psi'''_V, \Delta'''_V \rangle \in \mathcal{P}_{\psi'_U}$ **do**
        if $[\![\psi'_V \wedge \psi''_V]\!]_V \neq \emptyset \wedge [\![\psi_V \wedge \psi'''_V]\!]_V \neq \emptyset$ **then**
          $d_2 \leftarrow \bigsqcup \{d \mid \langle \cdot, d \rangle \in \Delta''_V\} \sqcap \bigsqcup \{d \mid \langle \cdot, d \rangle \in \Delta'''_V\}$;
          if $d_1 \neq d_2$ **then**
            return false;
return true;

Figure 5.9: Inter-Field Dependency check on DFD

Algorithm PARTITION(Δ, X)

$Y \leftarrow F \setminus X$;
$\mathcal{P} \leftarrow \emptyset$;
foreach $\langle \psi, d \rangle \in \Delta$ **do**
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{\langle \psi|_X, \{\langle \psi|_Y, d \rangle\}\rangle\}$;
while $\exists \langle \psi_1, \Delta_1 \rangle, \langle \psi_2, \Delta_2 \rangle \in \mathcal{P}$ *s.t.* $[\![\psi_1 \wedge \psi_2]\!]_X \neq \emptyset$ **do**
  $\mathcal{P} \leftarrow \mathcal{P} \setminus \{\langle \psi_1, \Delta_1 \rangle, \langle \psi_2, \Delta_2 \rangle\}$;
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{\langle \psi_1 \wedge \psi_2, \Delta_1 \cup \Delta_2 \rangle\}$;
  foreach $\psi_{1\setminus 2} \in$ DIFF(ψ_1, ψ_2) **do**
    if $[\![\psi_{1\setminus 2}]\!]_X \neq \emptyset$ **then**
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{\langle \psi_{1\setminus 2}, \Delta_1 \rangle\}$;
  foreach $\psi_{2\setminus 1} \in$ DIFF(ψ_2, ψ_1) **do**
    if $[\![\psi_{2\setminus 1}]\!]_X \neq \emptyset$ **then**
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{\langle \psi_{2\setminus 1}, \Delta_2 \rangle\}$;
return $\mathcal{P}$;

Figure 5.10: DFD Partition

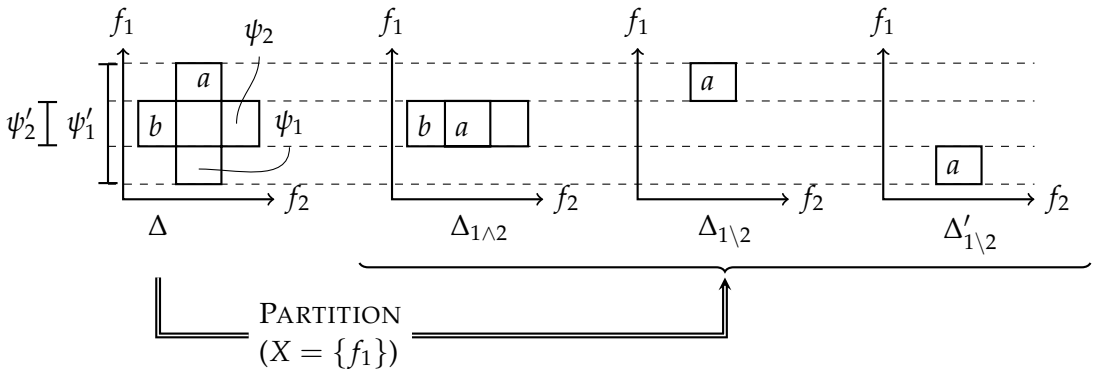


Figure 5.11: PARTITION algorithm on bidimensional DFD.

singleton $X = \{f_1\}$. First, we project all the RDs on the field f_1 (line 4 in Figure 5.10), obtaining $\psi_1' = \psi_1|_{\{f_1\}}, \psi_2' = \psi_2|_{\{f_1\}}$. Both the conjunction $\psi_1' \wedge \psi_2'$ and the difference $\text{DIFF}(\psi_1', \psi_2')$ have non empty extensions, hence they both contribute to the result. The former produces the DFD $\Delta_{1\wedge 2}$ (line 7), whereas the latter produces the pair of DFDs $\Delta_{1\setminus 2}, \Delta'_{1\setminus 2}$ (line 10). Note that $\text{DIFF}(\psi_2', \psi_1')$ instead does not yield any contribution, as the difference between ψ_2' and ψ_1' is empty. At the second iteration of the main loop (line 5), the test returns false because there are not any more pairs of RDs that have a non empty intersection on f_1; hence the algorithm terminates.

Thanks to the IFDCHECK algorithm, we now have a correct procedure to compute the refactoring of any given access control layer described in terms of a DFD. First we test for decomposability by the means of IFDCHECK and then, in case of success, we project (DFDPROJ) to obtain the desired decomposition.

5.4.1 Improving Request Space Partitioning

The generality of the PARTITION algorithm, with respect to the structure of field descriptors, makes it hard to analyze its complexity. Intuitively, a key role is played by the strategy used to choose which pair of items $\langle \psi_1, \Delta_1 \rangle, \langle \psi_2, \Delta_2 \rangle$ have to be partitioned at each iteration of the while loop (line 5), among all the candidates satisfying the condition $[\![\psi_1 \wedge \psi_2]\!]_X \neq \emptyset$. We illustrate this concept through the following example.

Example 5.8: Partition Strategy

Given an integer n, consider the set of $n/2$ concentric intervals $\{[i, n-i] \mid 1 \leq i \leq n/2 - 1\}$. By interpreting such intervals as single-field request descriptors $\psi_i = \langle [i, n-i] \rangle$, we can collect them in a DFD $\Delta = \{\langle \psi_i, d_i \rangle\} \subseteq \Psi(\{f\}) \times D$ and partition the latter with respect to

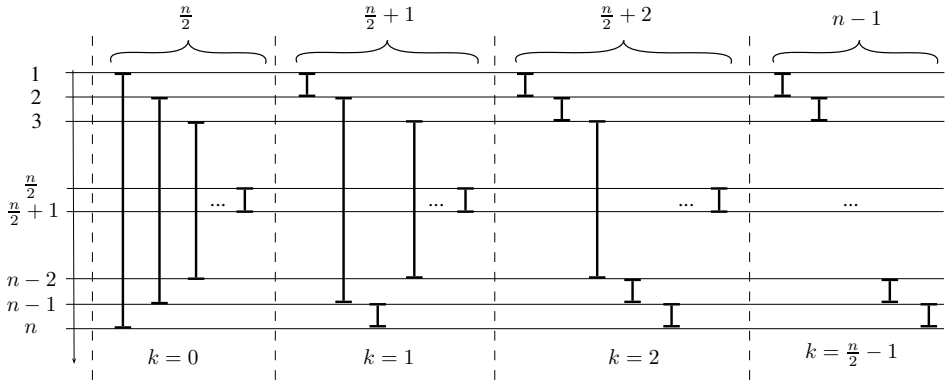


Figure 5.12: Optimal execution of PARTITION on a set of concentric intervals of integers.

the field f.

Note how, in this particular case, there is an obvious choice for the best items-selection strategy, that consists of selecting, at each iteration, the two longest intervals with non-empty intersection. Figure 5.12 depicts different iterations k of the algorithm implementing such a strategy. As each iteration reduces by two the number of intervals that intersect with all the others and increases by one the total number of intervals, the algorithm terminates after $n/2-1$ iterations, yielding $n-1$ non-intersecting intervals. Applying different strategies to the same input would necessarily require more iterations, as each step may not decrease, or may even increase, the number of mutually-intersecting intervals. It shall be noted however that this strategy, although optimal for this particular input, has in general higher overall complexity. In fact, it is not guaranteed that the number of intersecting intervals will be always reduced by two at each step. Moreover, searching for the two longest intersecting intervals requires, at each step, to consider all the 2-choices among them, depending quadratically on n.

It is indeed not trivial to determine a strategy that would be optimal in general, i.e., for every possible field descriptor and distribution of input RDs. On the other hand, it is reasonable to assume that specializing the PARTITION algorithm for a particular type of field descriptor would allow to improve it by leveraging specific properties of the chosen data structure. To verify this assumption, let us restrict to the field descriptor of intervals of positive integers (cf. Example 5.5). This descriptor is particularly interesting as it naturally encodes many common rule-based firewall-like policy languages. Hence, it is worthwhile exploring whether our algorithms can be optimized by taking advantage of its structural properties.

We observe that a straightforward approach for determining how intervals overlap with each other is to scan each point in the domain and keep track of which intervals it is part of. In fact, as intervals represent convex sets, it is sufficient to check only in the neighborhood of their endpoints. The algorithm PARTINTV, reported in Figure 5.13,

Algorithm PARTINTV(Δ, X)

$Y \leftarrow F \setminus X$;
foreach $f \in X$ **do**
  $\mathcal{Q}_f \leftarrow$ PQCREATE() ; // Creates a priority queue associated to field f.
  foreach $i \in \{1, 2, \ldots, |\Delta|\}$ **do**
    $\langle \psi, d \rangle \leftarrow \Delta[i]$; // Assuming any total order on Δ, $\Delta[i]$ denotes its i^{th} element.
    $\langle s, e \rangle \leftarrow$ PARSEINTVFD($\psi(f)$);
    PQINSERT($\mathcal{Q}_f, s, \langle$ `start`$, i \rangle$) ; // Inserts element with priority s in $\mathcal{Q}_f$.
    PQINSERT($\mathcal{Q}_f, e, \langle$ `end`$, i \rangle$);
$\mathcal{P} \leftarrow \{\langle \langle\rangle, \{1, 2, \ldots, |\Delta|\} \rangle\}$;
foreach $f \in X$ **do**
  foreach $\langle \psi, I \rangle \in$ PARTINTVSINGLEFIELD($\mathcal{Q}_f$) **do**
    $\mathcal{P}' \leftarrow \emptyset$;
    foreach $\langle \psi', I' \rangle \in \mathcal{P}$ **do**
      if $I' \cap I \neq \emptyset$ **then**
        $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\langle \psi' + \psi, I' \cap I \rangle\}$;
    $\mathcal{P} \leftarrow \mathcal{P}'$;
$\mathcal{R} \leftarrow \emptyset$;
foreach $\langle \psi, I \rangle \in \mathcal{P}$ **do**
  $\Delta' \leftarrow \emptyset$;
  foreach $i \in I$ **do**
    $\langle \psi', d \rangle \leftarrow \Delta[i]$;
    $\Delta' \leftarrow \Delta' \cup \{\langle \psi'|_Y, d \rangle\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{\langle \psi, \Delta' \rangle\}$;
return $\mathcal{R}$;

Algorithm PARTINTVSINGLEFIELD($\mathcal{Q}$)

$\langle p, \langle flag, i \rangle \rangle \leftarrow$ PQEXTRACTMIN($\mathcal{Q}$); // Extracts the element with minimum priority p from $\mathcal{Q}$.
$\mathcal{R} \leftarrow \emptyset$; $I \leftarrow \{i\}$; $s \leftarrow p$; $e \leftarrow 0$;
while $\neg$PQEMPTY($\mathcal{Q}$) **do**
  $\langle p, \langle flag, i \rangle \rangle \leftarrow$ PQEXTRACTMIN($\mathcal{Q}$);
  if $flag =$ `start` **then**
    $e \leftarrow p - 1$;
    if $s \geq 0 \wedge e \geq 0 \wedge e \geq s$ **then**
      $\psi \leftarrow \langle$ NEWINTVFD$(s, e)\rangle$; // Creates a RD for the interval $[s, e]$.
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{\langle \psi, I \rangle\}$;
    $I \leftarrow I \cup \{i\}$;
    $s \leftarrow p$;
  else
    $e \leftarrow p$;
    if $s \geq 0 \wedge e \geq 0 \wedge e \geq s$ **then**
      $\psi \leftarrow \langle$ NEWINTVFD$(s, e)\rangle$;
      $\mathcal{R} \leftarrow \mathcal{R} \cup \{\langle \psi, I \rangle\}$;
    $I \leftarrow I \setminus \{i\}$;
    $s \leftarrow p + 1$;
return $\mathcal{R}$;

Figure 5.13: DFD Partition for the field descriptor of positive integer intervals only.

precisely exploits this intuition. For each to-be-partitioned field, a priority queue is created. Then, the endpoints $\langle s, e \rangle$ of all the intervals of that field are inserted in the priority queue, together with the index i pointing to the current RD and a flag distinguishing `start` from `end` endpoints (lines 2–8). Next, each queue is processed by the PARTINTVSINGLEFIELD procedure. Here, each endpoint is extracted from the queue in ascending order and, distinguishing whether it is of type `start` or `end`, it is used to keep the set I updated with all the indexes pointing to the intervals that are overlapping with the current one identified by endpoints $\langle s, e \rangle$. Next, a single-field request descriptor ψ is created out of the current interval, it is then associated with the set I and added to the result. The second part of the PARTINTV algorithm (lines 9–16) computes the product among the results, one for each field, of PARTINTVSINGLEFIELD, by considering only the pairs $\langle \psi, I \rangle$, $\langle \psi', I' \rangle$ for which the sets of indexes I, I' have a non-empty intersection. Finally (lines 17–23), for each partitioned RD $\langle \psi, I \rangle$, the indexes $i \in I$ are resolved to the entries in the original DFD and associated to ψ.

It is straightforward to work out the complexity of the PARTINTVSINGLEFIELD algorithm as it essentially amounts to extracting all the elements out of a priority queue, that is well known to be a $O(|\mathcal{Q}| \log(|\mathcal{Q}|))$ problem (being $|\mathcal{Q}|$ the size of the queue). In our case, the size of the queue equals the total number of endpoints that are $2n$ for n intervals; hence we obtain a complexity of $O(n \log(n))$. The main loop of PARTINTV iterates $|X|$ times through the result of PARTINTVSINGLEFIELD, that is once for each field in the set X. This yields the overall computational cost of $O(|X|(n \log(n) + n^{|X|}))$, that will tend to behave like the $O(n^{|X|})$ polynomial the more the cardinality of set X will increase.

5.5 Experimental Evaluation

The algorithms presented in Sections 5.3 and 5.4 ensure that refactoring can be theoretically computed over DFDs. In this section, we ought to investigate the computational feasibility of our approach in practice. More specifically, we aim at evaluating the algorithms' performances and their sensitivity to various characteristics of input data, in order to drive conclusions about the concrete applicability of our approach.

The ideal way to carry out such an evaluation would be to make use of real policies to be given as input to the algorithms and measure the corresponding performances. Unfortunately, publicly-available security policies are a scarce resource in nature, as they constitute a private piece of information that system administrators would hardly disclose. In fact, sensitive details about the security infrastructure of an organization can be inferred from its policies, even when anonymization techniques are employed [Samak2009]. In order to evaluate the performance of our algorithms, given the lack of a sufficiently large database of real-world policies, we then chose to generate synthetic ones.

As first step in this direction, we identify what characteristics of input policies are expected to produce a relevant impact on performances. The purpose of this step is twofold. Firstly, it is needed to design a methodology that allows to control such characteristics while synthesizing datasets. Secondly, it allows to estimate the behaviour of the algorithms for an arbitrary input, e.g., real world policies, from the analysis of input's characteristics. To this end, based on the analysis of the algorithms presented in Sections 5.3 and 5.4, we formulate the following set of hypotheses.

(HP1) The *size of input policies*, i.e., the number of RDs they are made of once encoded as DFDs, is the major contributor to the cost of all the algorithms.

(HP2) The *number of fields* that are in common between two access control layers impacts the cost of composition (DFDCOMP) and inter-field dependency checking (IFDCHECK): in particular its partitioning subroutine (PARTINTV[1]), where the cardinality of the set of common fields influences the exponent of the polynomial complexity.

(HP3) The *average degree of overlap*, i.e., the average number of RDs any RD overlaps with, impacts the cost of inter-field dependency checking (IFDCHECK), but does not significantly influence the cost of partitioning (PARTINTV). In fact, an increasing probability of overlap in the input DFD (Δ) yields in average bigger DFDs (Δ_U, Δ_V) as the result of each round of partitioning in IFDCHECK, which then require more time to be processed. In contrast, PARTINTV shows no dependency on the average degree of overlap of input RDs.

(HP4) The *average degree of overlap* of the RDs of two different DFDs (on their common fields) impacts the cost of computing their composition (DFDCOMP). This follows from the analysis of the DFDCOMP algorithm, where one cost component is weighted by the probability of a pair of RDs to overlap on the common set of fields.

Before proceeding we shall precisely define what we mean by the notion of *average degree of overlap* and by that, strictly related to the former, of *overlap density*, for the set of RDs that compose a DFD. To do that, we first define the overlap relation between RDs.

Definition 28 (Overlap of Request Descriptors)**.** *We say that two RDs ψ_1, ψ_2 overlap with each other, denoted $\psi_1 \sim \psi_2$, when the extension of their conjunction is not empty, i.e., $[\![\psi_1 \wedge \psi_2]\!] \neq \emptyset$.*

We immediately see, from the previous definition, that the overlap relation is binary and symmetric. Hence, it can be equivalently interpreted as the edge relation of an undirected graph having a set of RDs as vertices. This interpretation is useful to

[1]We assume the PARTINTV algorithm being used for partitioning because, as shown in Section 5.4, it is easy to work out its complexity.

Algorithm GENINTV(n, M, ν)
Data: Number n of desired intervals
Data: Upper bound M of the domain of integers
Data: Variance coefficient ν
Result: Set of intervals S
1 Find points $\{c_i\}_1^n$ partitioning the domain $0, \ldots, M$ in n intervals of size $\frac{M}{n}$;
2 **forall the** c_i **do**
3 Generate a random integer d_i with log-normal distribution having parameters $\mu = \frac{M}{n}$ and $\sigma = \nu \times \frac{M}{n}$;
4 Add interval $[\max(0, c_i - \frac{d_i}{2}), \min(M, c_i + \frac{d_i}{2})]$ to S;
5 **return** S;

Figure 5.14: Generation of a set of random intervals of integers with controlled degree of overlap

characterize the concepts of density and average degree. We recall that, for an undirected graph $G = \langle V, E \rangle$, the density is defined as $\frac{2|E|}{|V|(|V|-1)}$ and the average degree as $\frac{2|E|}{|V|}$. Combining these quantities with the notion of overlap yields the following definition.

Definition 29 (Overlap Density and Average Degree of Overlap)**.** *Given a set of request descriptors S, we define its overlap density ρ and average degree of overlap k as follows*[2]*:*

$$\rho = \frac{2\left|\left\{\{\psi_1, \psi_2\} \in \binom{S}{2} \mid \psi_1 \sim \psi_2\right\}\right|}{|S|(|S|-1)},$$

$$k = \frac{2\left|\left\{\{\psi_1, \psi_2\} \in \binom{S}{2} \mid \psi_1 \sim \psi_2\right\}\right|}{|S|} = (|S|-1)\rho.$$

Note that, as the total number of 2-combinations in S (i.e., the cardinality of $\binom{S}{2}$) equals $\frac{|S|(|S|-1)}{2}$, the overlap density ρ corresponds to the probability that two random RDs in S overlap with each other. The average degree of overlap k is instead a measure of how many RDs are expected to overlap, in average, with one chosen randomly. In the following we will refer to either of the two quantities at convenience, knowing that the other can be obtained by simply multiplying or dividing by the factor $|S| - 1$.

In order to validate our hypotheses, we aim at designing a procedure to synthesize random DFDs by allowing to control (i) the number of therein contained RDs, (ii) the

[2]Throughout this chapter $\binom{S}{k}$ denotes the set of all the combinations (choices) of k elements in the set S, whereas $\binom{n}{k}$ is the number of combinations of n elements.

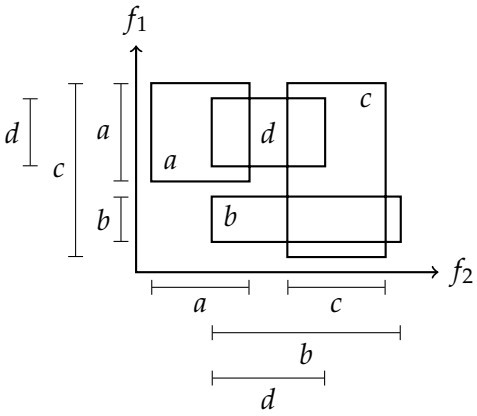
number of fields in each RD and (iii) the density of overlapping RDs. For the sake of simplicity, we will use only field descriptors for intervals of integers (cf. Example 5.5). As each interval is described by a pair of integers $[x, y]$ with $0 \leq x \leq y$, one can randomly generate $n \times m$ such pairs to obtain n RDs with m fields each. Clearly, in order to control the degree of overlap among such RDs, one has to carefully choose the probability distribution of the randomly chosen pairs of numbers. We first introduce a procedure to do so for unidimensional RDs, i.e., containing a single field with an associated domain of integers ranging from 0 to M. The algorithm GENINTV reported in Figure 5.14 generates n intervals having deterministic equally-spaced centers $\{c_i\}$ and random diameters $\{d_i\}$. The probability distribution of the diameters is log-normal (as such, only positive values are possible): its mean equals the distance between two centers and its variance can be tuned via the parameter ν. In Appendix C.2.1 we describe a procedure to empirically estimate the values of ν that allow to control the average degree of overlap k of the generated RDs for different choices of n. Being a function of both parameters, we denote such values as $\nu(k, n)$.

A (finite) multidimensional DFD is a relation $\Delta \subseteq \Psi(\{f_1, \ldots, f_m\}) \times D$ having n RDs over the set of fields $\{f_1, \ldots, f_m\}$. To randomly synthesize such a DFD, we execute GENINTV once for each field f_1 to f_m and we arrange the generated intervals in a $n \times m$ matrix. Each row (from 1 to n) of the matrix represents a m-dimensional RD to be associated with a random decision in D. While we are able to control, through the parameter $\nu(k, n)$ of the GENINTV algorithm, the average degree of overlap associated to the single dimensions (corresponding to fields f_1 to f_m), we cannot directly control the average degree of overlap of the entire m-dimensional RDs. The difference between unidimensional and multidimensional degree of overlap is illustrated in Figure 5.15. We consider four bidimensional RDs a, b, c, d (i.e., $n = 4$ and $m = 2$) and their unidimensional projections on fields f_1 and f_2 (Figure 5.15a). The overlap relation among the RDs projected on f_1, respectively on f_2, is represented by the edges of the graph in Figure 5.15b, respectively Figure 5.15c. The bidimensional RDs, as shown in Figure 5.15d, overlap with each other if and only if both projections do simultaneously overlap. The average degrees of these graphs are respectively $k_1 = 2, k_2 = 2.5$ and $k_{1,2} = 1.5$.

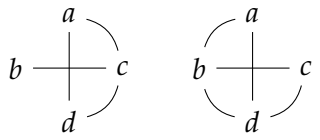
In order to control the multidimensional average degree of overlap, in Appendix C.2.2 we work out how it is related to the average degree of overlap of the single dimensions. We also show how to obtain, on the basis of the GENINTV algorithm, m sets of n unidimensional RDs having, for each set, (i) uniform probability of overlap between any pair of RDs and (ii) the same expected value of average degree of overlap k. We then argue that the expected value of average degree of overlap k_{mul} of the corresponding set of n m-dimensional RDs can be expressed as follows:

$$k_{\text{mul}} = \frac{k^m}{(n-1)^{m-1}}. \tag{5.5}$$

This follows from observing that, under the above assumptions, the overlap density of

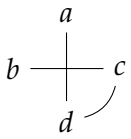


(a) Example overlapping bidimensional RDs.



(b) Overlap graph for f_1.

(c) Overlap graph for f_2.



(d) Bidimensonal overlap graph.

Figure 5.15: Unidimensional versus multidimensional degree of overlap.

the m-dimensional RDs (i.e., the probability of two random RDs to overlap with each other) equals the product of the m unidimensional overlap densities $\rho_{\text{mul}} = \rho^m$. We then substitute into the relation between ρ and k (Definition 29) to obtain (5.5).

Finally we recall that, according to Definition 26, it is necessary to ensure that the generated DFD will cover the entire request space. As already mentioned in Section 5.3, this can be easily achieved by adding a "default" RD that (i) ranges over the entire domain $[0, \ldots, M]$ and (ii) is associated to the least upper bound decision in D. As the default RD covers the entire domain, it will necessarily overlap with all the other n RDs. Hence, the overlap measures will change as follows:

$$\rho_{\text{def}} = \frac{2\left(\left|\left\{\{\psi_1, \psi_2\} \in \binom{S}{2} \mid \psi_1 \sim \psi_2\right\}\right| + n\right)}{(n+1)n} = \frac{2\left(\frac{n(n-1)}{2}\rho + n\right)}{(n+1)n} = \frac{(n-1)\rho + 2}{n+1}, \quad (5.6)$$

$$k_{\text{def}} = (n-1)\rho_{\text{def}} = (n-1)\frac{(n-1)\rho + 2}{n+1} = \frac{n-1}{n+1}(k+2). \quad (5.7)$$

5.5.1 Experiments

Figure 5.16 illustrates the workflow of an experiment that is designed to evaluate the performances of the different steps involved in the process of refactoring synthetic policies as a function of the parameters of our interest.

The first step consists in generating a pair of DFDs Δ_1, Δ_2 through the above-described GENINTV algorithm. To simplify the interpretation of results we use the same parameters n, m, k for generating both DFDs, respectively the number of RDs, the number of fields (or dimensions) and the expected average degree of overlap. Next,

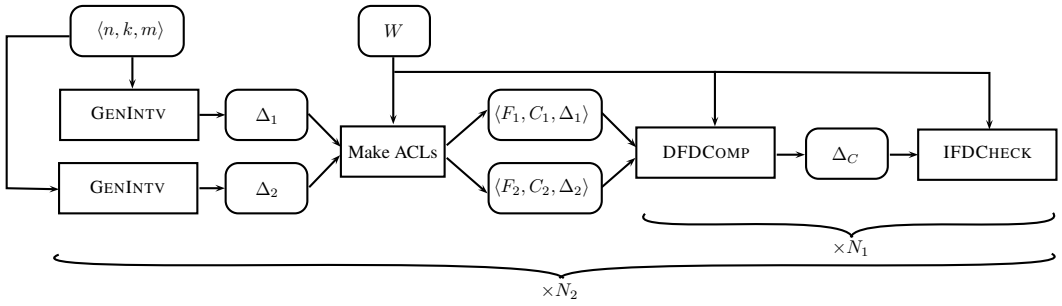


Figure 5.16: Experiment workflow.

we place the two DFDs in two access control layers $L_i = \langle F_i, C_i, \Delta_i \rangle$ (with $i \in \{1, 2\}$) in order to compute their composition, which requires generating corresponding request and coupling types. When doing so, we are left with one additional degree of freedom, that is to decide which fields, out of the m available, will be in common between the request (resp. coupling) types of the two layers. We denote such a set of common fields as $W = (F_1 \cap F_2) \cup C_2$. In order for the composition to be well defined (Definition 22), it is easy to check that the cardinality $|W|$ must be comprised between 1 and $(m - 1)$. Finally, we compute DFDComp$(L_1, L_2) = \langle C_1, F_1 \cup F_2, \Delta_C \rangle$ followed by IFDCheck(Δ_C, W, V), where $V = (F_2 \cup C_2) \setminus (F_1 \cup C_1) = F_2 \setminus F_1$[3] is the set of fields belonging exclusively to the layer L_2. In this process we measure the following time intervals:

- time to compute DFDComp(L_1, L_2);
- time to compute IFDCheck(Δ_C, W, V);
- time to compute PartIntv(Δ_C, W) as invoked by the IFDCheck algorithm.

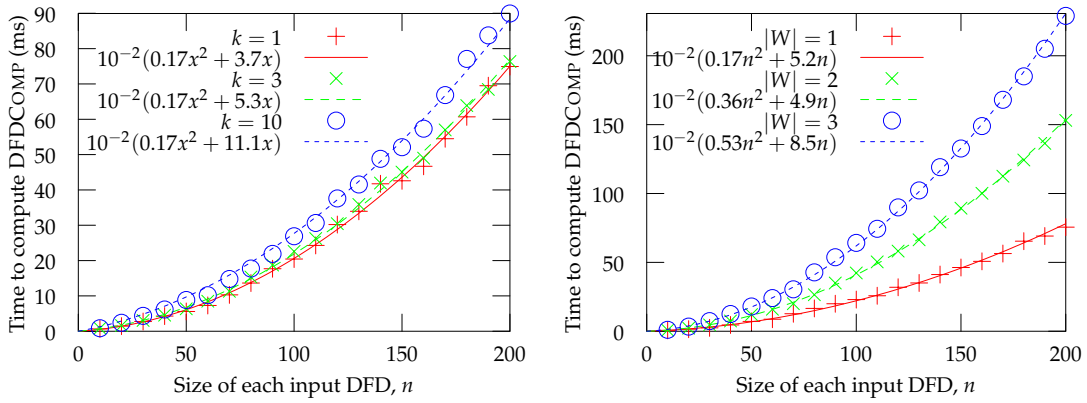
Executing IFDCheck directly on the output of DFDComp with respect to the set of common fields W, is a precise design choice. In fact, the result of the composition is surely decomposable with respect to W, hence (by Theorem 1) the IFD $W \twoheadrightarrow V$ necessarily holds on Δ_C. As a consequence, the IFDCheck algorithm will never terminate prematurely by returning **false** (line 14) and, as such, it will always be evaluated in the worst case scenario.

In order to remove the noise due to CPU contention, the measures of the last step are averaged over $N_1 = 25$ executions of the algorithms with the same input DFDs. Furthermore, for each choice of the parameters, we repeat the whole experiment (including the DFD generation step) $N_2 = 25$ times. This accounts for the inherent stochasticity of the GenIntv procedure, that will produce results exhibiting the expected degree of overlap k only in average over repeated executions. We executed the above experiment

[3]This equation holds because $C_2 \subset F_1$ and $C_1 \cap F_2 = \emptyset$ (cf. Definition 22).

n	$10, 20, 30, \ldots$
k	$1, 2, 3, 5, 10$
$m/\lvert W\rvert$	$2/1, 3/2, 4/3$

Table 5.17: Experiment parameters.



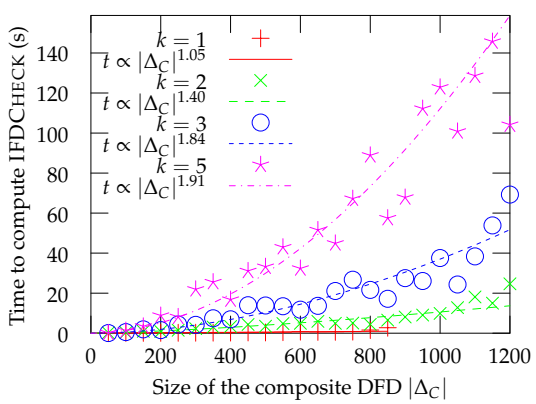
(a) Varying k with fixed $|W| = 1$.

(b) Varying $|W|$ with fixed $k = 2$.

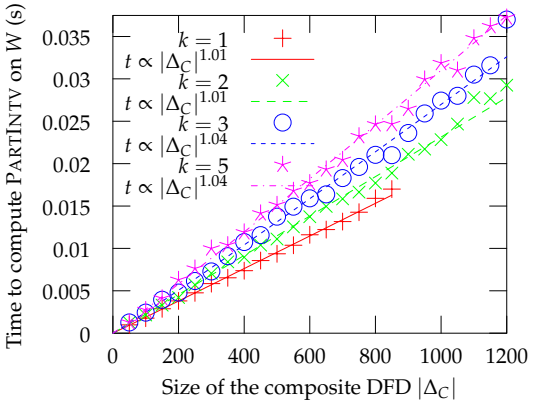
Figure 5.18: Evaluation of the DFDComp algorithm.

by varying the parameters $n, k, |W|$ as reported in Table 5.17. In the remainder of this section we discuss the results we obtained.

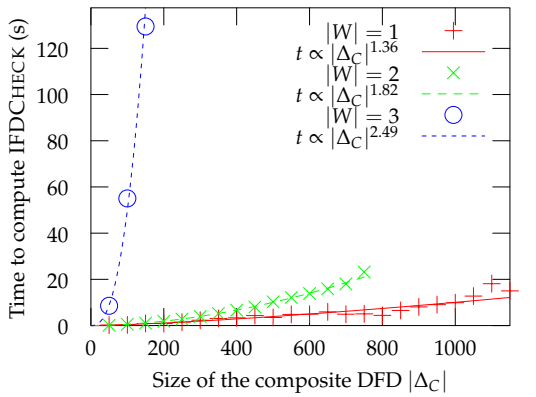
Figure 5.18 presents the trend of the execution time of the DFDComp algorithm measured as a function of the size n of the input DFDs, with parameters k and $|W|$ varying independently. Each point in the graphs is the average execution time over N_2 experiments with constant parameters. Figure 5.18a confirms the hypothesis (HP4) stated at the beginning of this section: increasing k yields, although not dramatically, increased computational cost. Note that the measured impact is limited because only the linear term of the quadratic fit is actually affected. To explain this fact, we recall that part of the cost of each iteration of the algorithm is weighted by the probability of RDs to overlap with each other, which is the same as the overlap density. As all generated DFDs contain a default RD, the overlap density is given by (5.6). Hence, the overlap probability expressed in terms of k equals $p(k) = \rho_{\text{def}} = k_{\text{def}}/(n-1) \simeq (k+2)/n$, with the last equality holding approximately for $n \gg 1$. Substituting for $k \in \{1, 3, 10\}$ yields the ratios $p(1)/p(3) = 0.6$ and $p(1)/p(10) = 0.25$, which approximately match the respective ratios among the linear coefficients of the experimental interpolations: $3.7/5.3 = 0.7$ and $3.7/11.1 = 0.33$. In Figure 5.18b, instead, we clearly see that the quadratic coefficient is affected when the number of common fields increases, yielding a more significant impact on the overall cost. At the same time, according to Equa-



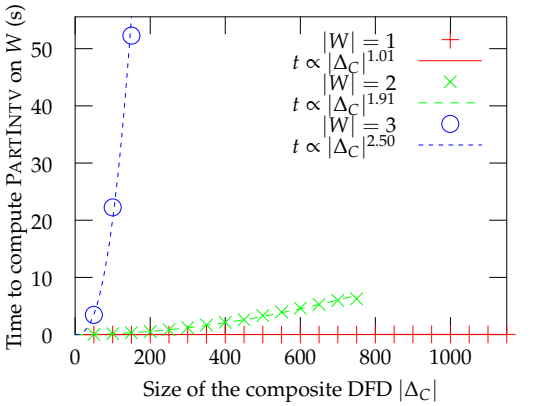
(a) IFDCHECK evaluation (var k, $|W| = 1$).



(b) PARTINTV evaluation (var k, $|W| = 1$).



(c) IFDCHECK evaluation (var $|W|$, $k = 2$).



(d) PARTINTV evaluation (var $|W|$, $k = 2$).

Figure 5.19: Evaluation of the IFDCHECK and PARTINTV algorithms.

tion (5.5), the actual average degree of overlap decreases, but the corresponding (linear) gain in performance cannot compensate the (quadratic) loss. Overall, this effect is consistent with the hypothesis (HP2).

Figure 5.19 depicts the evaluation results of the IFDCHECK and PARTINTV algorithms. Again parameters k and $|W|$ were changed independently to measure their influence on performances. In this case the algorithms' execution time is plotted as a function of the size of the composite DFD Δ_C, which constitutes the input of the algorithms. The graphs were obtained by filtering the experimental results through a moving average window of size N_2, to remove outliers due to the random nature of the input DFDs synthesis procedure.

From the comparison of Figures 5.19a and 5.19b we conclude that a change in the average degree of overlap definitely affects IFDCHECK, but has negligible impact on

PARTINTV, thereby confirming (HP3). Note also the considerable variance of the results with respect to the interpolating curve: this is due to the way samples were averaged. In fact, as the algorithms' input (Δ_C) is the result of composing two random DFDs, its size is also randomly distributed. As a consequence, being the moving average window of fixed size, there is a disparity in the number of samples that fall within each window. Hence, some points in the graph are less accurate as they result from averaging a smaller number of samples.

More significant is the sensitivity of both algorithms to increasing the number of common fields, as shown in Figures 5.19c and 5.19d. These graphs show that, for $k = 2$ and when W contains two or more fields, the asymptotic complexity of IFDCHECK is essentially determined by the partitioning subroutine PARTINTV. This is also very well reflected in the fact that the results for $|W| \geq 2$ do not almost exhibit any variance, as PARTINTV, not being significantly influenced by the actual average degree of overlap, dominates the overall complexity.

The worst-case runtime of our most complex algorithm, namely IFDCHECK, is of the order of some minutes for input DFDs containing more than a thousand RDs. We argue that this is reasonable for an offline policy analysis task. In fact, related work on conflict detection in firewall policies [Basile2012] exhibits performances of comparable order of magnitude.

5.6 Related Work

The pioneering work of Moffett and Sloman [Moffett1994] has opened a large avenue for research on policy conflict analysis in distributed systems and has been refined, classified and formalized in the network-level security field. For instance, Al-Shaer *et al.* [AlShaer2005] or Basile *et al.* [Basile2012] have proposed techniques and algorithms to automatically discover and manage inconsistencies between firewall rule sets. One key point of these techniques is to turn rule sets into an intermediate policy representation on which analysis is performed (e.g., ordered binary decision diagram [AlShaer2005], intersection closed semi-lattice of subsets [Basile2012]). Other approaches, e.g., [Alfaro2008], propose instead a more direct algorithmic solution that operates directly on firewall configurations. Although we do not focus on the detection of inconsistencies, we share with these approaches the abstract definition of policy decision points and the idea of policy composition, which we use to capture and analyze interactions between different layers. These techniques may complete our approach by furnishing a pre-processing procedure to remove local inconsistencies and help translate firewall configurations into instances of our model. For instance, the output of Algorithm 4 in [Alfaro2008] is a rule set where the mutual order of rules is irrelevant, i.e., where any overlap is removed. Hence, a corresponding DFD can be simply derived by computing one RD per each rule, plus an additional one for the implicit *deny-by-default* rule.

Network-level policy analysis is not limited to the detection of anomalies: another interesting perspective is the inference of high level policies from the reverse engineering of firewall configurations [Mayer2000; Tongaonkar2007; Nelson2010; Martnez2013]. These works aim at extracting the access control semantics contained in one or more firewalls' configuration by abstracting away low-level details and peculiarities of the different configuration languages. In contrast, we model the composition of policy decision points on different architectural layers, which is an orthogonal problem. As such, we could leverage these techniques to deal with intra-layer interactions between distributed firewalls, prior to performing our analysis. For instance, in [Martnez2013] a model-driven approach is used to turn firewall configurations into Platform Independent Models (PIMs). Next, such multiple PIMs are aggregated in a single one representing the global network access control policy of the system, which is what we require to instantiate our network Access Control Layer.

There exists a substantial body of work on combination of policies not limited to firewalls. Notably, several logical and algebraic approaches for composing and unifying access control policies have been proposed quite recently [Ni2009; Ramli2011; Bruns2011]. Different algebraic varieties have been used to combine rich decision spaces, for instance, $\mathcal{D}$-algebras [Ni2009], Belnap bilattices [Bruns2011] or XACML tailored logic [Ramli2011]. One of the goals is to provide mathematical foundations to the expressive XACML access control language and its many resolution strategies. The algebraic structure chosen here for decision space is motivated by previous work which have shown the need for expressive ones. However, the goal is not to capture resolution strategies but to find a structure both expressive enough and sufficient to define a decomposition with good properties. The use of an expressive common pivot model like XACML, logical frameworks or subject-target-condition rules [Zhao2011] is very interesting. However, we chose a different perspective by sticking as closely as possible to the original policies. Algorithms 5.7 and 5.9 carry computation directly on the original policies, with minimal prior normalization.

Our approach has a strong connection to the policy continuum model and the policy refinement problem. Davy et al. [Davy2008b] model policies at different inter-related abstraction layers in what they call policy continuum. Based on this model, they devise a generic algorithm for policy authoring. Their notion of continuum level essentially corresponds to a view on the policies at a particular abstraction level. In contrast, our ACLs represent types of decision points that operate at different architectural layers, but being at the same degree of abstraction (which roughly matches to the lowest possible continuum level). A similar argument applies to many existing works on policy refinement [Craven2010; Craven2011; Zhao2011]. Another key difference is that we start from interdependent concrete policies and we provide a device oriented decomposition instead of starting from high level requirements which are ultimately refined into operational policies. The refactoring problem studied in this chapter is *bottom-up*:

the global policy is nothing but the composition of all devices interacting along the network stack. By contrast, the policy refinement and deployment problem is clearly *top-down*. Another problem which is closely related to refinement is that of policy deployment, where the focus is put specifically on determining how refined policies shall be distributed among multiple policy enforcement points [Preda2010; Preda2011]. While also concerned with the generic notion of policy decomposition, these techniques focus on how to distribute network-layer policies either statically, according to the security properties required for the different possible paths in the network [Preda2010], or dynamically, according to the current context [Preda2011]. In contrast, we tackle the problem finding a decomposition of the global policy which removes the overlap among the request types of devices belonging to different architectural layers.

Our definition of access control layer is quite close to that of abstract access control systems [Tripunitara2007; Habib2009; Crampton2012b]. The first two references formally compare the expressiveness of access control models with respect to the set of decision functions they can produce. Interest is not brought on the configuration but on the model itself, whereas we focus on the first. Regarding the model, a set of desirable properties of abstract access control systems is identified. The question whether our decomposition technique still applies in their case is left for future work.

Finally, we shall mention that the idea of decomposing policies according to their structural properties for the purpose of facilitating their management is not new. Perhaps the most notable example is given by role mining techniques [Molloy2009; Frank2010], whose purpose is to decompose a direct *users-to-permissions* assignment relation into a pair of *users-to-roles* and *roles-to-permissions* ones. The main difference with our decomposition operation is that in role mining the total number of dimensions grows (the *role* dimension is introduced on top of *users* and *permissions*), whereas we aim at reducing the dimensionality (i.e., the request type's cardinality) of refactored policies in order to remove inter-layer overlap. However, we believe that the connection between the two problems deserves a deeper analysis. For instance, a substantial body of work in the role mining domain has been dedicated to finding approximate solutions that allow to reduce the number of mined roles at the expense of allowing a discrepancy between the original and the recomposed policies (cf. the definitions of δ-approximation and minimal-noise role mining in [Vaidya2007]). When discussing policy refactoring, we showed that it is not always possible to find a decomposition that preserves the permissiveness of the global policy, in which case we may want to look for approximate solutions. A key issue, for which we could take inspiration from the above mentioned role mining techniques, is to determine a meaningful metric to characterize the error that we aim at minimizing.

5.7 Discussion

In this section we discuss some additional technical issues and conclude with an outlook on more theoretical perspectives.

In order to execute our algorithms, input policies need to be represented as DFDs. We argued that, for rule-based access control languages, where rules are collections of independent filters on fields, this can be done by leveraging existing techniques. Other than most firewall configuration languages, the policy languages of many common network services fit into this category, arguably because of its simplicity. For those that do not, our approach can still be applied as long as the translation from the source language to DFDs remains feasible.

Both the DFDCOMP and DFDPROJ algorithms can exploit the fact that the request descriptors in a DFD are allowed overlap with each other, which allows to avoid computing the disjunction or the complement of RDs (respectively the union and the difference of their extensions). Avoiding such operations is beneficial because their computation is expensive. This is a consequence of RD extensions being defined as Cartesian products of sets: an operation that distributes with respect to intersection, but not to union and difference. Hence, a number of RDs that grows linearly with the number of sets involved in the product (namely the number of fields) is necessary to represent each union or difference. The IFDCHECK algorithm, computing the inter-field dependency check, is the most computationally-expensive. This is mainly due to the complexity of the partitioning subroutine, where, unlike for previous algorithms, we cannot avoid to compute the difference of RDs. We showed, however, that it is possible to trade off the generality of the field descriptors' constraint language with the complexity of the partitioning algorithm: by restricting to the class of convex intervals of integers, we obtained comparable performances to related configuration analysis algorithms, such as conflict analysis on firewall rulesets [Basile2012].

We believe that it is important to further investigate up to which scale the results of our analysis can still be consumable and insightful for users. This point is particularly critical when non-decomposability occurs. In this case, the IFDCHECK algorithm provides the first counterexample violating the inter-field dependency that was found in the partitioned DFD. However, it may be difficult to manually trace back the origin of the issue in the input policies, especially if they are large in cardinality and/or have a high degree of overlap. We believe that this problem can be mitigated by equipping the algorithm with a root-cause analysis feature, which, e.g., isolates the fragment of the policies that prevents decomposition.

Relational database experts may have recognized the syntactical resemblance between $W \twoheadrightarrow V$ of Definition 27 and so-called *MultiValued Dependencies* (MVD) introduced in [Fagin1977]. The two concepts are related in the fact that Theorem 1 of [Fa-

gin1977], linking lossless decomposition to MVDs, is a specific instance of Theorem 1 of this chapter, when the decision space is reduced to the two-elements boolean algebra. Since 2007, the generalization of the classical relational framework to "non boolean" relations, coined as "provenance", has received a lot of attention from the database community [Karvounarakis2012]. Here we equipped the decision space with the structure of a distributive lattice, a more constrained algebraic variety than the semiring described in [Karvounarakis2012], and we showed that MVDs and the lossless decomposition property generalize to this structure. However, the generalization of the classical dependency theory (e.g., MVDs or functional dependencies as well as more expressive classes) to the provenance setting is still open.

We argue that our contribution provides novel evidence that database dependency theory can be fruitfully applied to help solve security problems: an avenue that is yet considerably unexplored if compared, for instance, to how the Datalog model, and some of its many variants, have been used in the past to model and reason on access control policies [Becker2006; Abadi2003; Bertino2003; Halpern2008]. We outline two more related problems for which we believe that there exist a strong link with dependency theory.

The first problem is about repairs. Assume an access control policy over several fields cannot be decomposed into independent sub-policies: is there any canonical or best way to update the policy such that it will become decomposable? For instance, in Example 5.7 presented at the end of Section 5.4, one such possible modification is proposed for the policy δ_c. Applying the many results for the repair problems (e.g., see [Bertossi2011]) to policies is not without difficulties. Basically, one has to generalize the relational framework to be able to capture generic definition of access control systems and thus has to provide new results inspired from classical ones, as we have done for MVD in some sense. Moreover, one has to find repair semantics meaningful from the security point of view. Standard repair semantics are repair via insertions, deletions or updates [Wijsen2005]. It is interesting to study how these semantics apply to policy decomposition, the last one in particular. As mentioned in Section 5.6, related work on role and policy mining may provide further insights to find candidate metrics to characterize the quality of a repair.

The second problem is about the mining of dependencies. In the classical setting of database normalization, the set of dependencies is known in advance and one tries to provide the best structure fitting with the given constraints, as it is done in this paper. The data-mining perspective reverses the approach: The goal is to discover new dependencies by revealing the internal structure of relations. Savnik and Flach have provided an efficient algorithm to discover MVD [Savnik2000], we envision to apply their techniques to reverse engineer large policies. The idea is to find some "best set of simplest access control systems" with the same authorized queries.

5.8 Synthesis

This chapter proposed a formalization of the problem of inter-layer policy refactoring and a solution based on policy (de)composition. The key concept that captures decomposability is the inter-field dependency condition given in Section 5.4 from which an algorithm is derived. This algorithm and the other ones computing (de)composition work on a constraint-based relational representation of access control policies. We formulated qualitative hypotheses about the dependency of algorithms' performances on different characteristics of input policies and we confirmed our hypotheses through a quantitative experimental evaluation. We obtained worst-case performances that are comparable in the order of magnitude with those of related existing configuration analysis tasks.

Our main theoretical result (Theorem 1) proves that the decomposability of decision functions holds if and only if the *inter-field dependency* condition is satisfied. This is extends the classical link between lossless join decomposition and so-called multivalued dependencies [Fagin1977, Theorem 1] to larger-than-boolean relations: more precisely, to a constrained variety of *k-relations à la* [Karvounarakis2012], where the algebra of labels annotating tuples (i.e., our decisions) is a distributive lattice instead of a commutative semiring.

This contribution is conceptually part of the PoSecCo's enforceability analysis activities [Basile2013b][4], which aim at determining whether a collection of policy decision points are suitable to enforce a given global policy. Furthermore, it has been published in the proceedings of an international conference on networking [Casalino2013b] and in those of a French national conference on databases [Casalino2013a].

[4]However, it has not been included in [Basile2013b], because the approach was not ready for publication at the due date of the deliverable.

I may not have gone where I intended to go, but I think I have ended up where I needed to be.

—Douglas Adams, The Long Dark Tea-Time of the Soul

6 Conclusion

▷ *In this chapter we synthesize the three contributions of this thesis: namely (i) the proposal of a standard-based configuration validation language for distributed systems, (ii) the formalization and change analysis of authorization configurations for JEE Web applications, and (iii) the refactoring of multi-layered access control policies. We then discuss how such contributions would be positioned in an common integrated configuration analysis framework as well as how they relate to top-down policy refinement techniques. Finally, we outline future perspectives about possible extensions of our approach towards either solving different problems, such as policy conflict detection, or encompassing larger classes of access control systems, such as stateful firewalls or history-based policy decision points.* ◁

Chapter Outline

THIS thesis presented three contributions in the area of security configuration management that focus specifically on different aspects of the management of configuration changes in distributed information systems: from detecting and assessing undesired misconfigurations, to supporting the implementation of changes that preserve security properties.

This chapter presents our conclusions, which consist of three parts. In Section 6.1, we provide a synthesis going through our initial motivations and objectives, as stated in the introduction of the thesis, and summarizing the techniques we proposed to meet such objectives, as well as the results we obtained.

In Section 6.2, we present our contributions from an integrated perspective: instead of viewing them as individual analysis tasks, we position them in an hypothetical integrated configuration analysis framework. We discuss the feasibility of developing such a framework by identifying possible criticalities and missing components. Furthermore, we highlight possible dependencies and integration perspectives with *top-down* policy refinement and configuration synthesis techniques.

In Section 6.3 we conclude with an outlook on two future research avenues that we believe are worthwhile exploring. The first concerns investigating the applicability of our techniques to the problem of anomaly and conflict detection in security configurations. The second is about generalizing the semantic analysis tasks proposed here to a broader scope of systems including, for instance, stateful firewalls or history-based access control.

6.1 Synthesis

6.1.1 Configuration Validation

We started from the observation that system misconfiguration constitutes a major source of security incidents. Configuration validation is a technique which tackles this problem through the execution of syntactic configuration checks that automatically detect non-compliant or insecure configuration settings. Standardization efforts like the SCAP initiative promoted the exchange and reuse of configuration checks leading to the increasing adoption of this practice by the industry. However, having been conceived primarily for the analysis of individual machines, SCAP specifications as well as other configuration validation tools are not suitable for the validation of distributed systems. Our first objective consisted in analyzing the reasons behind such a limitation and investigating whether and how it can be overcome by building on top of the SCAP standards. To meet this objective, in Chapter 3 we identified requirements for a configuration validation language to detect misconfigurations in distributed systems and we argued that not all such requirements are fulfilled by the relevant SCAP specifications.

Specifically, we found the most important missing features being (i) the possibility of specifying generic (i.e., *intensional*) as well as specific (i.e., *extensional*) check targets spanning over multiple inter-related system components, and (ii) the lack of clear distinction between the specification of to-be-checked configuration objects (which is a matter for security experts) and the mechanisms to collect such objects from the system (which should be provided by system administrators). We designed a formal language based on OVAL that bridges the gap with the missing requirements and we defined the semantics of intensional checks' target definitions with respect to an external data source containing the details about distributed system components (e.g., software name, vendor, release, installation directories, IP addresses, etc.). We implemented a proof-of-concept interpreter that uses the standard OVAL evaluation algorithm for computing check results and that interprets their targets according to our formal semantics, by relying on a CIM-based configuration managent database as a data source. The tool collects configuration settings from the distributed targets by leveraging existing system and configuration management protocols and standards (e.g., JMX, SMB, SSH, etc.). This tool became part of the suite of prototypes of the PoSecCo project [Bettan2012]. The results of the project evaluation [Demetz2013] showed that it helped to improve the coverage of the system under analysis and to reduce the time required by configuration validation activities.

6.1.2 Formalization and Change Analysis of JEE Authorizations

A key feature of syntactic configuration validation is that the checks' language itself is agnostic with respect to the semantics of the configuration settings being checked. We argued that, while on the one hand this broadens the domain of applicability of the approach, on the other hand it hinders the verification of interesting semantic (e.g., security-relevant) properties of more expressive configuration languages. We considered the case of access control configurations, for which it is not generally trivial to check for equivalence or inclusion, with respect to the permissiveness of the corresponding policies, by the means of mere syntactic comparison. While this problem has been already tackled in literature for some categories of access control languages (e.g., firewall rulesets [Liu2007] or constrained subsets of XACML [Fisler2005]), an approach tailored to the specificities of authorization configurations for Web applications was still missing. Our second objective focused on developing formally correct procedures to interpret and compare the access control configurations of JEE Web applications with respect to their permissiveness. To fulfill this objective, in Chapter 4 we equipped the JEE access control language with a formal denotational semantics, interpreting configurations into a structure capturing authorizations over hierarchical resources (URLs). On top of this, we developed an algorithm that computes the partial order of permissiveness between any pair of configurations. This result is useful both to complement syntactic configuration validation with semantic comparison, for assessing the impact

of potential misconfigurations, and to help implementing new changes by ensuring that the modified configuration does not produce undesired side effects. Furthermore, our formal semantics constitutes an unambiguous reference interpretation for the access control language of JEE Web applications, so far only informally specified in the Java Servlet Specification [Coward2003]. To evaluate its correctness, we instrumented an automated testing procedure that compares our interpretation of automatically-generated configurations with the corresponding operational semantics of two popular JEE containers: Apache Tomcat and Oracle Glassfish. Our interpretation was never in disagreement with both containers at the same time for any tested configuration, which (although not constituting conclusive proof) argues in favour of its correctness. We found nevertheless some discrepancies with respect to the interpretations of the two containers individually. This led to the discovery of a bug [ASF2012] in Tomcat and of an inconsistent behaviour in Glassfish mandated by the JACC (Java Authorization Contract for Containers) specification [Monzillo2013], which is implemented by the latter but not by the former.

6.1.3 Multi-Layered Access Control Policy Refactoring

The above technique is in fact a form of static configuration analysis of a specific access control system, considered in isolation with respect to the surrounding environment. The applicability of this kind of approach to a distributed system is therefore limited to the analysis of the local configurations of different components, whereas global aspects, such as the impact of changes in one component's configuration on other components' behaviour, are ignored. Inter-component interactions in access control systems have been already studied in related work on policy composition and conflict detection. Less attention has been instead dedicated to the treatment of the interactions across different architectural layers, for instance, network filtering versus application-layer (e.g., HTTP) authorization. Our third objective focused on investigating how such interactions can be modeled to guide the change of local policies by preserving the permissiveness of the global, composite policy. Specifically, we aimed at finding an equivalent rewriting that simplifies local policies by removing, where possible, inter-layer overlap and that is consistent with the least privilege principle. To tackle this problem, which we named inter-layer policy refactoring, in Chapter 5 we proposed to model policies as decision functions, i.e., mappings from access control requests to decisions, embedded in a structure that keeps track of how requests of different layers couple with each other to yield composite decisions. We showed that solving the refactoring problem amounts to computing the composition of such structures followed by a decomposition. Furthermore, we developed a criterion to test for decomposability and that allows to check whether the problem admits a solution. We provided algorithms to compute (de)composition, as well as to test for decomposability, that work on an intensional, constraint-based representation of decision functions. Finally, we evaluated the algo-

rithms on synthetic datasets as a function of the size as well as other characteristics of the input. The most complex algorithm, namely the implementation of the decomposability test, performed comparably to other policy analysis tasks that exist in literature. The main theoretical result underlying this contribution is the proof of the equivalence between the decomposability condition of decision functions and the alternative criterion based on the concept of *inter-field dependency*, which we used in our algorithms. This is in fact an extension of a previous result in database dependency theory, linking lossless join decomposition with so-called multivalued dependencies [Fagin1977, Theorem 1], to larger-than-boolean relations.

6.2 Discussion

6.2.1 An Integrated Perspective

The contributions of this thesis complement each other on different dimensions, thereby covering, as depicted in Figure 6.1, various aspects of the spectrum of configuration analysis activities. At the interface with the operational system, which is the target of the analysis, lies syntactic configuration validation (Chapter 3). Here, the focus is put especially on identifying single as well as distributed target systems, which to-be-checked configurations have to be collected from. This is done by querying a Configuration Management DataBase (CMDB), providing the structural details of the system, and by fetching configurations directly through the management interfaces of system components. Next, syntactic checks are executed to detect possible misconfigurations. In order to deal with more expressive configuration languages, it is convenient to interpret configurations according to some formal model that enables semantic reasoning. In Chapter 4, we provided such an interpretation structure for the JEE access control configuration language. On top of it (top-left corner of Figure 6.1), we developed a comparison algorithm to determine whether a change in the configuration leads to a more or less permissive policy. This analysis is local to individual system components, namely JEE Web applications. In Chapter 5 we instead discussed inter-layer policy refactoring, which is a global reasoning task (top-right corner of Figure 6.1) in that it keeps track of how the configurations of components on different architectural layers influence each other's behaviour. Both local and global semantic reasoning work directly at the higher level of abstraction provided by the underlying language-specific interpreters. This separation allows, e.g., to apply the same local reasoning task to different configuration languages sharing the same semantics, or, as we did in Chapter 5, to integrate them in a global model by ignoring the syntactic differences. However, implementing such an integrated analysis framework is not without difficulties. First, more configuration interpreters are necessary: for instance, policy refactoring requires firewall rulesets and Apache Web server authorizations (grey-filled boxes in Figure 6.1)

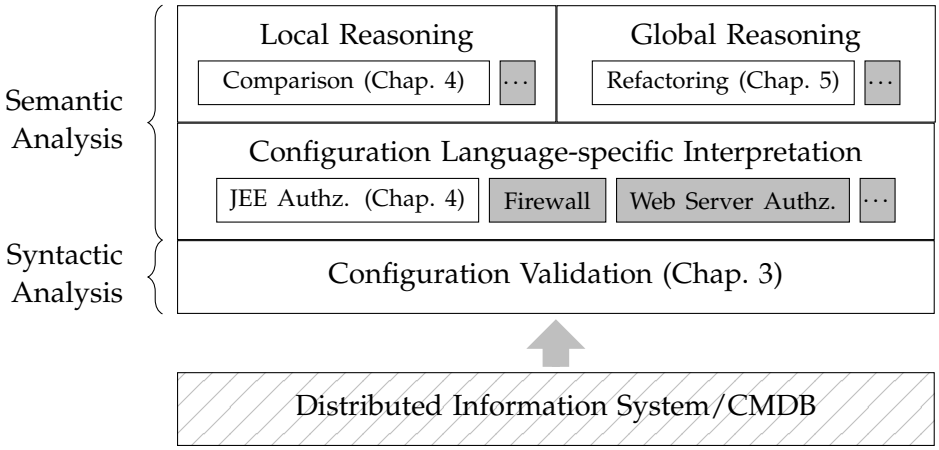


Figure 6.1: Integration of thesis' contributions (white-filled boxes) in configuration analysis framework.

to be interpreted as decision functions. As argued in Chapter 5, we believe that, for simple rule-based configuration languages, this can be done with limited effort by leveraging related work on policy and configuration analysis. However, existing approaches rarely share a common interpretation structure, but rather employ models that are tailored for the specific analysis task they support. Hence, it is not obvious to determine whether a single or multiple pivot models (and which ones) should be used for each class of configuration languages. Even more challenging would be to cope with configuration languages that have a radically different access control semantics, e.g., stateful firewalls or history-based access control, which would necessarily require modifying the global model. Furthermore, it would be interesting to explore whether our formal framework could support more reasoning tasks. We will expand on these last two points in Section 6.3.

6.2.2 Relations With Policy Refinement

As we discussed in the introduction of the thesis, all our contributions are meant to support the management of configuration changes (corresponding to phases (CM3) and (CM4) of the security configuration management cycle). We argued that the design and implementation phases (i.e., (CM1) and (CM2)) generally concern orthogonal problems, which have been widely discussed in related work. However, it is possible to identify cases where techniques developed for these two different contexts could benefit from a tighter integration. One example is the exploitation of policy refinement to support the automated generation of configuration validation checks. In Chapter 3 we proposed a language for expressing such checks, which we assumed being authored by security experts, product vendors, security auditors or system administrators. How-

ever, it may not be always worthwhile to author checks manually. For instance, this effort is more justified for checks that are generic in nature, e.g., security advisories or best practices — which, written once, are adopted by potentially many users — than for those that are specific to a particular system and policy, e.g., audit or policy compliance checklists. For the latter kind, it is envisageable to adapt existing policy refinement techniques, transforming high-level policies into concrete configurations, to automatically generate corresponding configuration checks as well. This approach has been explored, for example, in the context of the PoSecCo project (cf. Section 1.5), where the result of the top-down refinement process is exploited to produce checks testing for syntactic equivalence with respect to the *golden* (policy-mandated) configuration settings. We argue that this approach could be improved by taking into account the semantics of configurations while generating checks. For instance, if a *minimum password length* policy refines to the configuration setting "`min_length = x`", we would like to generate a check such as "`min_length >= x?`" (i.e., being aware of the *minimum length* semantics), rather than testing for equality with a check like "`min_length == x?`". Semantic threat graphs, which have been shown to be suitable tools to formalize and reason about vulnerability and best-practice descriptions [Foley2011], could be used as a base framework to support such a top-down check generation process.

A similar argument applies as well to access control configurations, where comparing semantic permissiveness is preferable to testing syntactic equivalence. This suggests the possibility of applying some of the techniques proposed in the thesis to such a problem: for instance, based on the model of Chapter 4, one could define a canonical representative in each equivalence class of all JEE configurations that have the same semantics (i.e., that interpret to the same decision function) and rewrite configurations to this canonical form prior to testing their syntactical equivalence. Furthermore, it may be worth to define such a canonical form in a way that it respects some criterion of optimality (e.g., minimizing the number of authorization rules). As such, policy refinement itself could benefit from incorporating our model to synthesize "optimal" configurations. Note that this is similar to what we did, on a different abstraction level, in Chapter 5, where we defined a rewriting (refactoring) for decision functions that minimizes the permissiveness of local policies according to the least privilege principle. Although, technically, such a rewriting is not canonical in general, it is unique for every choice of *coupling* and *request types* that decompose the global policy.

6.3 Future Work

As future research directions, we discuss two extension perspectives concerning, respectively, the opportunity of applying our techniques to other related problems and that of generalizing the problems discussed in this thesis to a larger context.

A related problem that attracted the attention of several researchers is the detection of conflicts or anomalies in security policies [Moffett1994; Lupu1999; Uszok2003; Davy2008a] and configurations [Fu2001; AlShaer2005; Hamed2006; Alfaro2008; Basile2012]. Many of these approaches classify binary or n-ary relations among the authorization rules within a policy according to a taxonomy of conflicts: e.g., rules matching to an overlapping set of resources yield a *permit-deny* conflict if they mention opposite decisions, or a *redundancy* conflict if they mention the same one. However, especially for concrete configuration languages, it is not always obvious to determine whether a particular combination of rules constitutes an authentic anomaly or whether it should be considered a reasonable use of the language. In Chapters 4 and 5, we bypassed this issue by providing methods that abstract away syntactic details and let users gain assurance about the actual semantics of configurations. It is interesting to investigate whether such techniques could be used to detect conflicting or anomalous configurations. For instance, our interpretation structure for JEE authorizations (Chapter 4) is based on the hierarchy of URLs. We could leverage such a structure to define criteria for anomaly detection: e.g., it is reasonable to assume that authorization policies should be monotonically more restrictive according to the URL hierarchy, i.e., children nodes should not be accessible by more roles than their parents. We could apply a comparable reasoning to define a conflict semantics for multi-layer policies (Chapter 5), by checking for monotonicity of permissions along the stack of access control layers. It would be interesting to explore, more generally, which other structural properties of the interpretation model are able to capture semantic anomalies.

The semantics-aware configuration analysis techniques that we proposed in the thesis are restricted to the scope of access control configurations, as they constitute both an interesting and non trivial class of configuration languages. The model we chose for access control systems, namely authorization decision functions, is conceptually simple, yet substantially powerful. Indeed, as argued in [Crampton2010; Crampton2012b], it can instantiate several well-known access control models, e.g., DAC (Direct Access Control), MAC (Mandatory Access Control), RBAC (Role-Based Access Control), ABAC (Attribute-Based Acces Control), as well as more complex and concrete frameworks such as XACML. However, there are access control systems that do not fit this model, two notable examples being stateful firewalls [Gouda2005] and history-based access control frameworks which are needed, for example, to support delegation or to enforce the well-known Chinese Wall policy [Brewer1989]. These systems have in common the fact that the authorization decision cannot be computed only from the current state of the system (i.e., its configuration) and some access request, but it is necessary to keep track of past authorization decisions too. Formally, this corresponds to a different type for decision functions, which encodes this information as a state transition of the system: $\delta : Q \times \Sigma \to D \times \Sigma$ (with Q, Σ and D being respectively the spaces of requests, states and decisions).

Previous work has already been conducted in this area. For instance, in [Alfaro2013] Alfaro et al. propose a comprehensive algorithmic approach to detect inconsistencies between stateful firewall configurations and the specifications of session-oriented network protocols, described as finite-state automata. Although abstracting away vendor-specific details of firewall configuration languages, their model is still quite specific to the network domain: it would be interesting to study whether their approach can be extended to reason on generic decision functions. More generic stateful access control systems are instead discussed in [Guelev2004], where model-checking techniques are employed to detect whether malicious goals can be reached through the execution of a sequence of actions (attacks) carried out by an intruder potentially acting in parallel with legitimate users.

These approaches provide interesting models and techniques to reason on state-changing access control systems. However, we believe that studying whether and how some of the problems that we discussed in this thesis, e.g., change impact analysis or policy refactoring, generalize to this setting is yet an open and challenging issue. Solving such problems would require complementing our approach with different formal frameworks, e.g., temporal logics, that are more suitable to reason on state-transition systems.

A

XSD Schemas

▷ *This appendix reports the XSD schemas defining the grammar of the XML languages that are either defined as part of the thesis contribution or that are discussed in the scope of the thesis.* ◁

Chapter Outline

A.1 XML Configuration Object, State and Test

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3   xmlns:oval="http://oval.mitre.org/XMLSchema/oval-common-5"
4   xmlns:oval-def="http://oval.mitre.org/XMLSchema/oval-definitions-5"
5   xmlns:ind-def="http://oval.mitre.org/XMLSchema/oval-definitions-5#
      independent"
6   xmlns:sch="http://purl.oclc.org/dsdl/schematron" xmlns:coas-def="http://
      oval.mitre.org/XMLSchema/oval-definitions-5#coas"
7   targetNamespace="http://oval.mitre.org/XMLSchema/oval-definitions-5#coas
      "
8   elementFormDefault="qualified" version="5.8">
9
10  <xsd:annotation>
11    <xsd:documentation>The OVAL Schema is maintained by The MITRE
12    Corporation and developed by the public OVAL Community. For more
13    information, including how to get involved in the project and how to
14    submit change requests, please visit the OVAL website at
15    http://oval.mitre.org.
16    </xsd:documentation>
17    <xsd:appinfo>
18      <schema>COAS Definition</schema>
19      <version>5.10</version>
20      <date>9/15/2010 1:55:32 PM</date>
21      <terms_of_use>Copyright (c) 2002-2010, The MITRE Corporation. All
22      rights reserved. The contents of this file are subject to the terms
23      of the OVAL License located at
24      http://oval.mitre.org/oval/about/termsofuse.html. See the OVAL
25      License for the specific language governing permissions and
26      limitations for use of this schema. When distributing copies of the
27      OVAL Schema, this license header must be included.</terms_of_use>
28      <sch:ns prefix="oval-def"
29        uri="http://oval.mitre.org/XMLSchema/oval-definitions-5" />
30      <sch:ns prefix="coas-def"
31        uri="http://oval.mitre.org/XMLSchema/oval-definitions-5#coas" />
32      <sch:ns prefix="xsi" uri="http://www.w3.org/2001/XMLSchema-instance"
       />
33    </xsd:appinfo>
34  </xsd:annotation>
35  <xsd:element name="xmlconfiguration_test"
36    substitutionGroup="oval-def:test">
37    <xsd:annotation>
38      <xsd:documentation>
39      This test is for checking xmlconfigurations. Checked are Xpath
      expressions. The xmlconfiguration_object does not contain information
      about where the xml file is.
40      It extends the standard TestType as defined in the oval-definitions-
      schema and one should refer to the TestType description for more
      information. The required object element references an
```

```
      xmlconfiguration_test and the optional state element specifies the
      data to check.
41      The evaluation of the test is guided by the check attribute that is
      inherited from the TestType.
42    </xsd:documentation>
43    <xsd:appinfo>
44      <oval:element_mapping>
45        <oval:test>xmlconfiguration_test</oval:test>
46        <oval:object>xmlconfiguration_object</oval:object>
47        <oval:state>xmlconfiguration_state</oval:state>
48        <oval:item
49          target_namespace="http://oval.mitre.org/XMLSchema/oval-system-
      characteristics-5#coas">xmlconfiguration_item</oval:item>
50      </oval:element_mapping>
51    </xsd:appinfo>
52    <xsd:appinfo>
53      <sch:pattern id="coas-def_xmlconfigtst">
54        <sch:rule context="coas-def:xmlconfiguration_test/coas-
      def:object">
55          <sch:assert
56            test="@object_ref=ancestor::oval-def:oval_definitions/oval-
      def:objects/coas-def:xmlconfiguration_object/@id">
57            <sch:value-of select="../@id" />
58            - the object child element of a xmlconfiguration_test must
59            reference a xmlconfiguration_object
60          </sch:assert>
61        </sch:rule>
62        <sch:rule context="ind-def:xmlconfiguration_test/coas-def:state"
      >
63          <sch:assert
64            test="@state_ref=ancestor::oval-def:oval_definitions/oval-
      def:states/coas-def:xmlconfiguration_state/@id">
65            <sch:value-of select="../@id" />
66            - the state child element of a xmlconfiguration_test must
67            reference a xmlconfiguration_state
68          </sch:assert>
69        </sch:rule>
70      </sch:pattern>
71    </xsd:appinfo>
72  </xsd:annotation>
73  <xsd:complexType>
74    <xsd:complexContent>
75      <xsd:extension base="oval-def:TestType">
76        <xsd:sequence>
77          <xsd:element name="object" type="oval-def:ObjectRefType" />
78          <xsd:element name="state" type="oval-def:StateRefType"
79            minOccurs="0" maxOccurs="unbounded" />
80        </xsd:sequence>
81      </xsd:extension>
82    </xsd:complexContent>
83  </xsd:complexType>
```

```
84  </xsd:element>
85  <xsd:element name="xmlconfiguration_object"
86    substitutionGroup="oval-def:object">
87    <xsd:annotation>
88      <xsd:documentation>
89      The xmlconfiguration_object contains the information that is needed
      to evaluate the system-characteristics file.
90      </xsd:documentation>
91    </xsd:annotation>
92    <xsd:complexType>
93      <xsd:complexContent>
94        <xsd:extension base="oval-def:ObjectType">
95          <xsd:sequence>
96            <xsd:choice>
97              <xsd:element ref="oval-def:set" />
98              <xsd:sequence>
99                <xsd:element name="cpe" type="oval-
      def:EntityObjectStringType"
100                 minOccurs="1">
101                 <xsd:annotation>
102                   <xsd:documentation>
103                   Logical identifier of the XML configuration container.
       Case sensitive. Only operation="equals" is supported.
104                   </xsd:documentation>
105                 </xsd:annotation>
106               </xsd:element>
107               <xsd:element name="representation" type="oval-
      def:EntityObjectStringType"
108                 minOccurs="1">
109                 <xsd:annotation>
110                   <xsd:documentation>
111                   Contains a key to be understood by the collector, e.g
      ., "DeploymentDescriptor". Case sensitive.
112                   </xsd:documentation>
113                 </xsd:annotation>
114               </xsd:element>
115               <xsd:choice>
116                 <xsd:element name="xpath" type="oval-
      def:EntityObjectStringType"
117                   minOccurs="1">
118                   <xsd:annotation>
119                     <xsd:documentation>
120                     XPath expression for evaluating XML. Case sensitive.
       Only operation="equals" is supported.
121                     </xsd:documentation>
122                   </xsd:annotation>
123                 </xsd:element>
124                 <xsd:element name="xquery" type="oval-
      def:EntityObjectStringType"
125                   minOccurs="1">
126                   <xsd:annotation>
```

```
127                       <xsd:documentation>
128                       XQuery expression for evaluating XML. Case sensitive
    . Only operation="equals" is supported.
129                       </xsd:documentation>
130                     </xsd:annotation>
131                   </xsd:element>
132
133                 </xsd:choice>
134                 <xsd:element ref="oval-def:filter" minOccurs="0"
135                   maxOccurs="unbounded" />
136               </xsd:sequence>
137             </xsd:choice>
138           </xsd:sequence>
139         </xsd:extension>
140       </xsd:complexContent>
141     </xsd:complexType>
142   </xsd:element>
143   <xsd:element name="xmlconfiguration_state"
144     substitutionGroup="oval-def:state">
145     <xsd:annotation>
146       <xsd:documentation>
147       The xmlconfiguration_state element describes the desired (in case of
     compliance checks) or undesired (in case of vulnerability checks)
    value of the object.
148       </xsd:documentation>
149     </xsd:annotation>
150     <xsd:complexType>
151       <xsd:complexContent>
152         <xsd:extension base="oval-def:StateType">
153           <xsd:sequence>
154             <xsd:element name="cpe" type="oval-def:EntityStateStringType"
155               minOccurs="0" maxOccurs="1">
156               <xsd:annotation>
157                 <xsd:documentation>
158                 Logical identifier of the XML configuration container.
    Case sensitive. Only operation="equals" is supported.
159                 </xsd:documentation>
160               </xsd:annotation>
161             </xsd:element>
162             <xsd:element name="value_of" type="oval-
    def:EntityStateAnySimpleType"
163               minOccurs="0" maxOccurs="unbounded">
164               <xsd:annotation>
165                 <xsd:documentation>
166                 May contain any simple value. The result of the Xpath
    expression is checked against this value.
167                 </xsd:documentation>
168               </xsd:annotation>
169             </xsd:element>
170           </xsd:sequence>
171         </xsd:extension>
```

```
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

A.2 Check and Target Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault=
    "qualified" attributeFormDefault="unqualified">
  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <!--  Permit any of these tags in any order in any number     -->
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="software_component" type="scType" />
          <xs:element name="target_definition" type="tdType" />
          <xs:element name="check_definition" type="cdType" />
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:group name="tdGrp">
    <xs:choice>
      <xs:element name="association">
        <xs:complexType>
          <xs:sequence>
            <xs:group ref="tdGrp"/>
            <xs:group ref="tdGrp"/>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:choice>
        <xs:element name="software_component_ref" type="scRefType" />
        <xs:element name="software_component" type="scType" />
      </xs:choice>
    </xs:choice>
  </xs:group>

  <xs:complexType name="tdType">
    <xs:group ref="tdGrp"/>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="scRefType">
   <xs:attribute name="sc_ref" type="xs:string" use="required"/>
```

```
41  </xs:complexType>
42
43  <xs:complexType name="scType">
44   <xs:sequence maxOccurs="unbounded">
45     <xs:element name="condition">
46      <xs:complexType>
47       <xs:attribute name="property" type="xs:string" use="required"/>
48       <xs:attribute name="operator" use="required">
49         <xs:simpleType>
50           <xs:restriction base="xs:string">
51             <xs:enumeration value="equals" />
52             <xs:enumeration value="less" />
53             <xs:enumeration value="less_eq" />
54             <xs:enumeration value="greater" />
55             <xs:enumeration value="greater_eq" />
56           </xs:restriction>
57         </xs:simpleType>
58       </xs:attribute>
59       <xs:attribute name="value" type="xs:string" use="required"/>
60     </xs:complexType>
61   </xs:element>
62 </xs:sequence>
63 <xs:attribute name="id" type="xs:string" use="required"/>
64 </xs:complexType>
65
66 <xs:complexType name="cdType">
67   <xs:sequence maxOccurs="unbounded">
68     <xs:element name="target_mapping">
69       <xs:complexType>
70         <xs:sequence maxOccurs="unbounded">
71           <xs:element name="test">
72             <xs:complexType>
73               <xs:attribute name="test_ref" use="required"/>
74             </xs:complexType>
75           </xs:element>
76         </xs:sequence>
77         <xs:attribute name="sc_ref" type="xs:string" use="required"/>
78       </xs:complexType>
79     </xs:element>
80   </xs:sequence>
81   <xs:attribute name="id" type="xs:string" use="required"/>
82   <xs:attribute name="od_ref" type="xs:string" use="required"/>
83   <xs:attribute name="td_ref" type="xs:string" use="required"/>
84 </xs:complexType>
85
86 </xs:schema>
```

A.3 Collectors

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http:
      //www.w3.org/2001/XMLSchema-Instance" xmlns:oval="http://oval.mitre.
      org/XMLSchema/oval-common-5" xmlns:cm="http://www.sap.com/coas/xccdf/
      collector-mapping" targetNamespace="http://www.sap.com/coas/xccdf/
      collector-mapping" elementFormDefault="qualified" attributeFormDefault
      ="unqualified">
3
4   <xsd:element name="collector-mapping" type="cm:CollectorMappingType" />
5
6   <xsd:complexType name="CollectorMappingType">
7     <xsd:sequence>
8       <xsd:element name="collectors" type="cm:CollectorsType" maxOccurs="1
      " minOccurs="1" />
9     </xsd:sequence>
10  </xsd:complexType>
11
12  <xsd:complexType name="CollectorsType">
13    <xsd:sequence>
14      <xsd:element name="collector" type="cm:CollectorType" maxOccurs="
      unbounded" minOccurs="1">
15        <xsd:key name="collectorKeyId">
16          <xsd:selector xpath="cm:collector" />
17          <xsd:field xpath="@id" />
18        </xsd:key>
19      </xsd:element>
20    </xsd:sequence>
21  </xsd:complexType>
22
23  <xsd:complexType name="CollectorType">
24    <xsd:sequence>
25      <xsd:element name="description" type="xsd:string" maxOccurs="1"
      minOccurs="0" />
26      <xsd:element name="platform" type="cm:PlatformType" maxOccurs="1"
      minOccurs="0" />
27      <xsd:element name="oval_objects" type="cm:OvalObjectsType" maxOccurs
      ="1" minOccurs="0" />
28      <xsd:element name="parameters" type="cm:ParametersType" maxOccurs="1
      " minOccurs="0">
29        <xsd:unique name="uniqueParamName">
30          <xsd:selector xpath="./cm:parameter" />
31          <xsd:field xpath="@name" />
32        </xsd:unique>
33      </xsd:element>
34    </xsd:sequence>
35    <xsd:attribute name="id" type="cm:CollectorIdType" use="required" />
36    <xsd:attribute name="type" type="xsd:string" use="required" />
37  </xsd:complexType>
```

```
38
39   <xsd:complexType name="ParametersType">
40     <xsd:sequence>
41       <xsd:element name="parameter" type="cm:ParameterType" maxOccurs="
       unbounded" minOccurs="0" />
42     </xsd:sequence>
43   </xsd:complexType>
44
45   <xs:complexType name="PlatformType">
46     <xs:sequence maxOccurs="unbounded">
47       <xs:element name="condition">
48         <xs:complexType>
49           <xs:attribute name="property" type="xs:string" use="required"/>
50           <xs:attribute name="operator" use="required">
51             <xs:simpleType><xs:restriction base="xs:string">
52               <xs:enumeration value="equals" />
53               <xs:enumeration value="less" />
54               <xs:enumeration value="less_eq" />
55               <xs:enumeration value="greater" />
56               <xs:enumeration value="greater_eq" />
57             </xs:restriction></xs:simpleType>
58           </xs:attribute>
59           <xs:attribute name="value" type="xs:string" use="required"/>
60         </xs:complexType>
61       </xs:element>
62     </xs:sequence>
63   </xs:complexType>
64
65   <xsd:complexType name="ParameterType">
66     <xsd:attribute name="name" type="xsd:string" use="required" />
67     <xsd:attribute name="value" type="xsd:string" use="optional" />
68   </xsd:complexType>
69
70   <xsd:complexType name="OvalObjectsType">
71     <xsd:restriction base="xsd:string" />
72   </xsd:complexType>
73
74   <xsd:simpleType name="CollectorIdType">
75     <xsd:restriction base="xsd:string">
76       <xsd:pattern value="oval:[A-Za-z0-9_\-\.]+:col:[1-9][0-9]*" />
77     </xsd:restriction>
78   </xsd:simpleType>
79 </xsd:schema>
```

A.4 Target Mapping

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xccdf="
      http://checklists.nist.gov/xccdf/1.2" targetNamespace="http://www.sap.
      com/coas/xccdf/target-mapping" xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-Instance" xmlns:tm="http://www.sap.com/coas/xccdf/target-
      mapping"
3 xmlns:oval="http://oval.mitre.org/XMLSchema/oval-common-5" xmlns:cm="http:
      //www.sap.com/coas/xccdf/collector-mapping" elementFormDefault="
      qualified" attributeFormDefault="unqualified">
4
5   <xsd:annotation>
6     <xsd:documentation xml:lang="en">
7       This schema defines the Target-Mapping file used by the COAS
      application. This is mapping file used to select on which target
      systems a given collection of OVAL tests shall be executed. This file
      can also be used to provide additional parameters to the collectors.
8       <version date="25 October 2011">0.1</version>
9     </xsd:documentation>
10    <xsd:appinfo>
11      <schema>Target mapping</schema>
12      <version>0.1</version>
13      <date>2011-10-25</date>
14    </xsd:appinfo>
15  </xsd:annotation>
16
17  <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
      schemaLocation="../common/xml.xsd">
18    <xsd:annotation>
19      <xsd:documentation xml:lang="en"> Import the XML namespace because
      this schema uses the xml:lang and xml:base attributes.
20      </xsd:documentation>
21    </xsd:annotation>
22  </xsd:import>
23
24  <xsd:import namespace="http://oval.mitre.org/XMLSchema/oval-common-5"
      schemaLocation="../oval/oval-common-schema.xsd">
25    <xsd:annotation>
26      <xsd:documentation xml:lang="en"> Import the OVAL common schema
      because we will reuse the defined id types.
27      </xsd:documentation>
28    </xsd:annotation>
29  </xsd:import>
30
31  <xsd:import namespace="http://www.sap.com/coas/xccdf/collector-mapping"
      schemaLocation="../mappings/collector-mapping-schema.xsd">
32    <xsd:annotation>
33      <xsd:documentation xml:lang="en"> Import the collector mapping
      schema because we will reuse the collector id type.
```

```
34        </xsd:documentation>
35      </xsd:annotation>
36    </xsd:import>
37
38    <xsd:element name="target-mapping" type="tm:TargetMappingType">
39      <xsd:keyref name="targetInRuleRef" refer="tm:targetKeyId">
40        <xsd:selector xpath=".//tm:criterion" />
41        <xsd:field xpath="@target" />
42      </xsd:keyref>
43
44      <xsd:keyref name="targetInGroupRef" refer="tm:targetKeyId">
45        <xsd:selector xpath=".//tm:criterion" />
46        <xsd:field xpath="@target" />
47      </xsd:keyref>
48    </xsd:element>
49
50    <xsd:complexType name="TargetMappingType">
51      <xsd:sequence>
52        <xsd:element name="targets" type="tm:TargetsType" maxOccurs="1"
      minOccurs="1">
53          <xsd:key name="targetKeyId">
54            <xsd:selector xpath="tm:target" />
55            <xsd:field xpath="@id" />
56          </xsd:key>
57        </xsd:element>
58        <xsd:element name="mappings" type="tm:MappingsType" maxOccurs="1"
      minOccurs="1" />
59      </xsd:sequence>
60    </xsd:complexType>
61
62    <xsd:complexType name="TargetsType">
63      <xsd:sequence>
64        <xsd:element name="target" type="tm:TargetType" maxOccurs="unbounded
      " minOccurs="1" />
65      </xsd:sequence>
66    </xsd:complexType>
67
68      <xsd:complexType name="MappingsType">
69      <xsd:sequence minOccurs="1" maxOccurs="unbounded">
70        <xsd:element name="target-mapping-rule" type="tm:TargetMappingType"/
      >
71      </xsd:sequence>
72    </xsd:complexType>
73
74    <xsd:complexType name="TargetType">
75      <xsd:sequence>
76        <xsd:element name="description" type="xsd:string" maxOccurs="1"
      minOccurs="0" />
77        <xsd:element name="parameters" type="tm:ParametersType" maxOccurs="1
      " minOccurs="0">
78          <xsd:unique name="uniqueParameterName">
```

```
79            <xsd:selector xpath="tm:parameter" />
80            <xsd:field xpath="@name" />
81          </xsd:unique>
82        </xsd:element>
83      </xsd:sequence>
84      <xsd:attribute name="id" type="tm:TargetId" use="required" />
85    </xsd:complexType>
86
87    <xsd:complexType name="ParametersType">
88      <xsd:sequence>
89        <xsd:element name="parameter" type="tm:ParameterType" maxOccurs="
        unbounded" minOccurs="1" />
90      </xsd:sequence>
91    </xsd:complexType>
92
93    <xsd:complexType name="ParameterType">
94      <xsd:attribute name="name" type="xsd:string" use="required" />
95      <xsd:attribute name="value" type="xsd:string" use="required" />
96    </xsd:complexType>
97
98    <xsd:complexType name="TargetMappingType">
99      <xsd:element ref="tm:criterionTest" minOccurs="1" maxOccurs="unbounded
        " />
100     <xsd:attribute name="ovaldef" type="oval:DefinitionIDPattern"
101       use="optional" />
102   </xsd:complexType>
103
104   <xsd:element name="criterionTest"  type="tm:CriterionTestType" />
105
106   <xsd:complexType name="CriterionTestType">
107     <xsd:sequence>
108       <xsd:element name="test-mapping" type="tm:TestMappingType" maxOccurs
        ="unbounded" minOccurs="1"/>
109     </xsd:sequence>
110   </xsd:complexType>
111
112   <xsd:complexType name="TestMappingType">
113     <xsd:sequence>
114       <xsd:element name="parameters" type="tm:ParametersType" maxOccurs="1
        " minOccurs="0"/>
115       <xsd:element name="tests" type="tm:TestsType" maxOccurs="1"
        minOccurs="1"/>
116     </xsd:sequence>
117     <xsd:attribute name="target" type="xsd:string"/>
118   </xsd:complexType>
119
120   <xsd:complexType name="TestsType">
121     <xsd:sequence>
122       <xsd:element name="test" type="tm:TestType"
123             minOccurs="1" maxOccurs="unbounded"/>
124     </xsd:sequence>
```

```
125   </xsd:complexType>
126
127   <xsd:complexType name="TestType">
128     <xsd:attribute name="id" type="tm:TestId" use="required" />
129     <xsd:attribute name="collector" type="cm:CollectorIdType"
130       use="optional" />
131   </xsd:complexType>
132
133   <xsd:simpleType name="TestId">
134     <xsd:restriction base="xsd:string">
135     </xsd:restriction>
136   </xsd:simpleType>
137
138   <xsd:simpleType name="TargetId">
139     <xsd:restriction base="xsd:string">
140       <xsd:pattern value="xccdf_[^_]+_target_.+" />
141     </xsd:restriction>
142   </xsd:simpleType>
143
144 </xsd:schema>
```

B

Proofs of the Propositions

▷ *This appendix provides the fully-detailed proofs of all the results stated in this thesis.* ◁

Chapter Outline

B.1 Proofs of Chapter 4 Results

Proposition 1. We first show that $\prec$ is indeed a partial order for $\mathcal{U}$, as it is a reflexive, antisymmetric and transitive relation.

Reflexivity requires proving that $|u| \leq |u|$, which is trivial, and $u = u^{\leq |u|}$ which follows directly from the definition of URL prefix.

Antisymmetry holds since, assuming $u \prec v$ and $v \prec u$, it follows that $|u| = |v|$ and hence $u = v^{\leq |u|} = v^{\leq |v|} = v$.

For proving *transitivity* we first state a general property which follows from the definition of URL prefix:

$$\forall i, j : i < j \Rightarrow (u^{\leq j})^{\leq i} = u^{\leq i}. \tag{B.1}$$

In particular, given URLs u and v, s.t. $u = v^{\leq |u|}$, we can take the l-prefix of both sides of the equality $u^{\leq l} = (v^{\leq |u|})^{\leq l}$ and conclude, by (B.1):

$$l \leq |u| \Rightarrow u^{\leq l} = (v^{\leq |u|})^{\leq l} = v^{\leq l}. \tag{B.2}$$

Therefore, assuming $u \prec w$ and $w \prec v$, we directly have $|u| \leq |w| \leq |v|$ and $w = v^{\leq |w|} \Rightarrow w^{\leq |u|} = v^{\leq |u|} = u$, hence $u \prec v$.

To prove that $u\!\downarrow$ is well-ordered according to $\prec$, we shall prove that (i) $\prec$ is a total order for $u\!\downarrow$ and (ii) all its subsets have a least element.

For (i) we need to show that $\forall v, w \in u\!\downarrow$: $v \prec w \vee w \prec v$. Let $|w| \leq |v|$. Since $v \in u\!\downarrow$, then $v \prec u$, hence $v = u^{\leq |v|}$. As $|w| \leq |v|$, we can write $v^{\leq |w|} = u^{\leq |w|} = w$ (B.2), hence $w \prec v$. Assuming $|v| \leq |w|$ we would analogously obtain $v \prec w$.

For (ii) we want to prove that $\forall S \in \wp(u\!\downarrow)$ and $\forall v \in S$, $\exists w \in S : w \prec v$. It is easy to see that such element is the URL w having minimum length in S, because $\forall v \in S$ we have $w \prec u$, $v \prec u$ and $|w| \leq |v|$, hence $w \prec v$. □

Proposition 2. We rewrite $u\!\downarrow_*$ in terms of the set of u predecessors $u\!\downarrow$, as $u\!\downarrow_* = \{w' \oplus \langle * \rangle \mid w' \in u\!\downarrow \wedge w' \oplus \langle * \rangle \in U\}$. We know from Proposition 1 that $u\!\downarrow$ is totally ordered w.r.t. $\prec$, therefore $\forall v, v' \in u\!\downarrow \Rightarrow v \prec v' \vee v' \prec v$. If we assume $v \neq v'$, it follows from Definition 12 that either $|v| < |v'|$ or $|v'| < |v|$ hold, hence:

$$v, v' \in u\!\downarrow \wedge\, v \neq v' \Rightarrow |v| \neq |v'|. \tag{B.3}$$

We now observe that all the URLs $v \in u\!\downarrow$ with $v \neq u$ can't end with the $*$ symbol by definition. In fact, if such a URL w could exist, then we would have $w = u^{\leq |w|} = \langle \ldots, * \rangle$, and therefore $u = \langle \ldots, *, \ldots \rangle$ which is not a URL according to Definition 11. We can then write $v \in u\!\downarrow \Rightarrow v_{|v|} \neq *$ leading, according to (4.4), to the following conclusion:

$$v \in u\!\downarrow \Rightarrow |v \oplus \langle * \rangle| = |v| + 1. \tag{B.4}$$

From both (B.3) and (B.4) it follows that the URL length function, restricted to the domain of $*$-predecessors $|\cdot| : u\downarrow_* \to \mathbb{N}$, is injective. Indeed $\forall w = (v \oplus \langle * \rangle), \forall w' = (v' \oplus \langle * \rangle)$ with $v, v' \in u\downarrow$ (resp. $w, w' \in u\downarrow_*$), if $v \neq v'$ (equivalently $w \neq w'$) then $|w| \neq |w'|$; formally:

$$w, w' \in u\downarrow_* \wedge w \neq w' \Rightarrow |w| \neq |w'|. \tag{B.5}$$

Finally, recalling that $\max(u\downarrow_*) = \{v \in u\downarrow_* \mid \forall w \in u\downarrow_*, |w| \leq |v|\}$, we conclude:

1. if $u\downarrow_* = \emptyset$ then $\max(u\downarrow_*) = \emptyset$, since $\nexists v \in u\downarrow_*$;
2. otherwise $\exists!\ w \in \max(u\downarrow_*)$ and w is the longest URL in $u\downarrow_*$. This follows since every distinct element of $u\downarrow_*$ is mapped through the injective function $|\cdot|$ (B.5) to a distinct element in a finite non-empty subset of $\mathbb{N}$, which is a totally-ordered set according to the natural ordering of integers, and hence it has exactly one maximum element.

□

Proposition 3. For the *only if* direction, we assume $t_1 \leq_T t_2$ and $\delta_1 = 1$. According to definition of δ we have two cases. First case, $r = \top$ and $\hat{\rho}_1(u, m) = \top$, but as $t_1 \leq_T t_2$, $\hat{\rho}_2(u, m)$ is $\top$ too and $\delta_2(u, m, \top) = 1$. Second case, $r \neq \top$, so $\hat{\rho}_1(u, m) \bigsqcap r \neq \emptyset$, however, as $\mathcal{L}$ is a lattice, $\bigsqcap$ is monotonic with respect to $\leq_R$ and $\leq_R \hat{\rho}_2(u, m) \bigsqcap r$ is not empty either. In both cases we conclude that $\delta_2(u, m, r) = 1$.

For the *if* direction, we use proof by contrapositive. Assume we have some u and m such that $r_1 = \hat{\rho}_1(u, m)$ and $r_2 = \hat{\rho}_2(u, m)$ with $r_2 \leq_R r_1$ and $r_2 \neq r_1$. We consider two cases. If $r_1 = \top$, then it suffices to look at the value for $r = \top$: $\delta_1(u, m, \top) = 1$ and r_2 is different from $\top$ so $\delta_2(u, m, r) = 0$ by definition of δ. Otherwise $r_1 \neq \top$ thus r_1 is some subset of R and $r_2 \subseteq r_1$, so we consider an $x \in r_1 \backslash r_2$ which exists as the difference is not empty. $\delta_1(u, m, \{x\}) = 1$ and $\delta_2(u, m, \{x\}) = 0$ as x is in r_1 but not in r_2.

□

B.2 Proofs of Chapter 5 Results

Lemma 1. *Let* $\langle F_1 \cup F_2, C_1, \delta \rangle = \langle F_1, C_1, \delta_1 \rangle \otimes \langle F_2, C_2, \delta_2 \rangle$, $W = (F_1 \cap F_2) \cup C_2$, $U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1$ *and* $V = F_2 \setminus F_1$.

Then, $\{U, V, W\}$ *is a partition of* $(F_1 \cup C_1 \cup F_2)$.

Proof. Recall that $F_i \cap C_i = \emptyset, i \in \{1, 2\}$ (Definition 20) and that $C_2 \subseteq F_1, F_2 \cap C_1 = \emptyset$ (Definition 22).

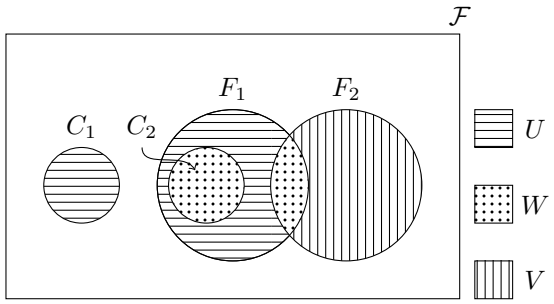


Figure B.1: Request Types Partition

We then have the following equalities, which can be easily verified by inspection of Figure B.1:

$$\begin{aligned}
U \cup W &= ((F_1 \setminus F_2) \setminus C_2) \cup C_1 \cup (F_1 \cap F_2) \cup C_2 = \\
&= (F_1 \setminus F_2) \cup (F_1 \cap F_2) \cup C_1 = F_1 \cup C_1 \\
U \cap W &= (((F_1 \setminus F_2) \setminus C_2) \cup C_1) \cap ((F_1 \cap F_2) \cup C_2) = \emptyset \\
V \cup W &= (F_2 \setminus F_1) \cup (F_1 \cap F_2) \cup C_2 = F_2 \cup C_2 \\
V \cap W &= (F_2 \setminus F_1) \cap ((F_1 \cap F_2) \cup C_2) = \emptyset \\
U \cap V &= (((F_1 \setminus F_2) \setminus C_2) \cup C_1) \cap (F_2 \setminus F_1) = \emptyset \\
U \cup V \cup W &= (F_1 \cup C_1) \cup (F_2 \cup C_2) = F_1 \cup C_1 \cup F_2.
\end{aligned}$$

□

Lemma 2. *Let $\mathcal{L}$ be a finite bounded distributive lattice and $f : X \to \mathcal{L}$, $g : Y \to \mathcal{L}$ two generic functions having $\mathcal{L}$ as codomain. Then the following holds true:*

$$\bigsqcup_{x \in X} f(x) \sqcap \bigsqcup_{y \in Y} g(y) = \bigsqcup_{x \in X \wedge y \in Y} f(x) \sqcap g(y). \tag{B.6}$$

Proof. As $\mathcal{L}$ is finite, the image of f, $\mathcal{L}_f$, is finite too and, because of lattices' idempotence,

$$\bigsqcup_{x \in X} f(x) = \bigsqcup \mathcal{L}_f = \bigsqcup \{l \in \mathcal{L} \mid \exists x \in X,\, l = f(x)\} = l_1 \sqcup \ldots \sqcup l_{n_1},$$

with finite $n_1 = |\mathcal{L}_f|$. Likewise $\bigsqcup_{y \in Y} g(y) = l'_1 \sqcup \ldots \sqcup l'_{n_2}$, with $n_2 = |\mathcal{L}_g|$. By applying $n = n_1 \times n_2$ times the distributive law of $\sqcup$ with respect to $\sqcap$, we obtain:

$$\bigsqcup_{x \in X} f(x) \sqcap \bigsqcup_{y \in Y} g(y) = \bigsqcup_{l \in \mathcal{L}_f} l \sqcap \bigsqcup_{l \in \mathcal{L}_g} l' = \bigsqcup_{\substack{l \in \mathcal{L}_f \wedge \\ l' \in \mathcal{L}_g}} l \sqcap l'.$$

As there are up to n different pairs $\langle l, l' \rangle$, by idempotence we conclude the thesis. □

Proof of Proposition 4. Let $L = \langle F_1 \cup F_2, C_1, \delta \rangle$ such that $L = \langle F_1, C_1, \delta_1 \rangle \otimes \langle F_2, C_2, \delta_2 \rangle$.

By Lemma 1, sets $W = (F_1 \cap F_2) \cup C_2$, $U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1$ and $V = F_2 \setminus F_1$ partition F. Moreover $F_1 \cup C_1 = W \cup U$ (resp. $F_2 \cup C_2 = W \cup V$).

By combining the definitions of projection (Definition 23) and composition (Definition 22) and using the distributive law we obtain, $\forall q = wu \in \mathfrak{Q}(F_1 \cup C_1) = \mathfrak{Q}(W \cup U)$,

$$\pi_{W \cup U}(\delta)(wu) = \bigsqcup_{v \in \mathfrak{Q}(V)} \delta(wuv) = \bigsqcup_{v \in \mathfrak{Q}(V)} \delta_1(wu) \sqcap \delta_2(wv) = \delta_1(wu) \sqcap \bigsqcup_{v \in \mathfrak{Q}(V)} \delta_2(wv) \leq \delta_1(wu).$$

Likewise we have, $\forall q' = wv \in \mathfrak{Q}(F_2 \cup C_2) = \mathfrak{Q}(W \cup V)$, $\pi_{W \cup V}(\delta)(wv) \leq \delta_2(wv)$. □

Proof of Proposition 6. By hypothesis, we have $F_1 \cup F_2 \in 2^{\mathfrak{F}}$ request type, with $F_2 \not\subseteq F_1$, and $C_1 \in 2^{\mathfrak{F}} \setminus \{\emptyset\}$ coupling type. We need to show that, in general, it is not possible to find $L_1 = \langle F_1, C_1, \delta_1 \rangle$ and $L_2 = \langle F_2, C_2, \delta_2 \rangle$ such that $L_1 \otimes L_2 = \langle F_1 \cup F_2, C_1, \delta \rangle$ (for any $C_2 \subseteq F_1$). To do that, we show that, if we assume that such L_1, L_2 exist, it is always possible to find a δ that leads to a contradiction.

Let $F = F_1 \cup C_1 \cup F_2$. By Lemma 1, sets $W = (F_1 \cap F_2) \cup C_2$, $U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1$ and $V = F_2 \setminus F_1$ partition F. As $C_1 \neq \emptyset$ (Definition 20), the set U cannot be empty and, since $F_2 \not\subseteq F_1$, V is not empty either. Hence, the corresponding request spaces are not singletons: namely $\mathfrak{Q}(U) \neq \{\square\}$ and $\mathfrak{Q}(V) \neq \{\square\}$. Thus $\exists\, u_1, u_2 \in \mathfrak{Q}(U)$ and $\exists\, v_1, v_2 \in \mathfrak{Q}(V)$ such that $u_1 \neq u_2 \wedge v_1 \neq v_2$. Moreover, as W, U, V partition F, there exist $q_1, q_2, q_3 \in \mathfrak{Q}(W \cup U \cup V) = \mathfrak{Q}(F)$ that can be expressed as follows (the concatenations are well defined because W, U, V are disjoint):

$$\begin{aligned} q_1 &= w + u_1 + v_1 \\ q_2 &= w + u_2 + v_1 \\ q_3 &= w + u_2 + v_2. \end{aligned}$$

Assuming now $\delta(q_1) \neq \mathbf{0}_D \wedge \delta(q_2) = \mathbf{0}_D \wedge \delta(q_3) \neq \mathbf{0}_D$ entails, according to Definition 22, the following contradiction:

$$\begin{cases} \delta_1(q_3|_{C_1 \cup F_1}) = \delta_1(wu_2) \neq \mathbf{0}_D \\ \delta_2(q_1|_{C_2 \cup F_2}) = \delta_2(wv_1) \neq \mathbf{0}_D \\ \delta_1(q_2|_{C_1 \cup F_1}) = \delta_1(wu_2) = \mathbf{0}_D \vee \delta_2(q_2|_{C_2 \cup F_2}) = \delta_2(wv_1) = \mathbf{0}_D. \end{cases}$$

□

Proof of Theorem 1. As stated in Lemma 1, it is possible to partition the decision space of δ in the following three sets of fields: $W = (F_1 \cap F_2) \cup C_2$, $U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1$ and

$V = F_2 \setminus F_1$. A generic request q in such a request space can then be expressed as the concatenation of sequences $w \in \mathfrak{Q}(W)$, $u \in \mathfrak{Q}(U)$ and $v \in \mathfrak{Q}(V)$, i.e., $q = wuv$.

$\Leftarrow$. For the *if* direction we assume $\delta \models W \twoheadrightarrow V$ and we show that $\forall q \in \mathfrak{Q}(F)$, $\pi_{F_1 \cup C_1}(\delta)(q|_{F_1 \cup C_1}) \sqcap \pi_{F_2 \cup C_2}(\delta)(q|_{F_2 \cup C_2}) = \delta(q)$.

First, notice that, as $\delta \models W \twoheadrightarrow V$, according to Definition 27 the following is true $\forall w \in \mathfrak{Q}(W), u, u' \in \mathfrak{Q}(U), v, v' \in \mathfrak{Q}(V)$:

$$\delta(wuv) \sqcap \delta(wu'v') = \delta(wuv') \sqcap \delta(wu'v). \tag{B.7}$$

We now have the following sequence of equalities.

$$\begin{aligned}
&\pi_{W \cup U}(\delta)(wu) \sqcap \pi_{W \cup V}(\delta)(wv) = \\
&= \left(\bigsqcup_{v' \in \mathfrak{Q}(V)} \delta(wuv') \right) \sqcap \left(\bigsqcup_{u' \in \mathfrak{Q}(U)} \delta(wu'v) \right) = && \text{by (5.2)} \\
&= \delta(wuv) \sqcup \bigsqcup_{v' \in \mathfrak{Q}(V) \setminus \{v\}} \delta(wuv') \sqcap \delta(wuv) \sqcup \bigsqcup_{u' \in \mathfrak{Q}(U) \setminus \{u\}} \delta(wu'v) = \\
&= \delta(wuv) \sqcup \bigsqcup_{v' \in \mathfrak{Q}(V) \setminus \{v\}} \delta(wuv') \sqcap \bigsqcup_{u' \in \mathfrak{Q}(U) \setminus \{u\}} \delta(wu'v) = && \text{by distributivity} \\
&= \delta(wuv) \sqcup \bigsqcup_{\substack{u' \in \mathfrak{Q}(U) \setminus \{u\} \\ v' \in \mathfrak{Q}(V) \setminus \{v\}}} \delta(wuv') \sqcap \delta(wu'v) = && \text{by (B.6)} \\
&= \delta(wuv) \sqcup \bigsqcup_{\substack{u' \in \mathfrak{Q}(U) \setminus \{u\} \\ v' \in \mathfrak{Q}(V) \setminus \{v\}}} \delta(wuv) \sqcap \delta(wu'v') = && \text{by (B.7)} \\
&= \delta(wuv) \sqcup \left(\delta(wuv) \sqcap \bigsqcup_{\substack{u' \in \mathfrak{Q}(U) \setminus \{u\} \\ v' \in \mathfrak{Q}(V) \setminus \{v\}}} \delta(wu'v') \right) = && \text{by distributivity} \\
&= \delta(wuv) && \text{by absorption.}
\end{aligned}$$

$\Rightarrow$. For the *only if* direction we reason by contradiction.

Assume $\forall q \in \mathfrak{Q}(F)$

$$\pi_{F_1 \cup C_1}(\delta)(q|_{F_1 \cup C_1}) \sqcap \pi_{F_2 \cup C_2}(\delta)(q|_{F_2 \cup C_2}) = \delta(q), \tag{B.8}$$

and $\delta \not\models W \twoheadrightarrow V$. For instance, let

$$\delta(wu_1v_1) \sqcap \delta(wu_2v_2) \neq \delta(wu_1v_2) \sqcap \delta(wu_2v_1), \tag{B.9}$$

for some $w \in \mathfrak{Q}(W), u_1, u_2 \in \mathfrak{Q}(U), v_1, v_2 \in \mathfrak{Q}(V)$.

We know, from the first half of the proof, that we can rewrite (B.8) as

$$\delta(wu_1v_1) = \delta(wu_1v_1) \sqcup \bigsqcup_{\substack{u' \in \mathfrak{Q}(U)\setminus\{u_1\} \\ v' \in \mathfrak{Q}(V)\setminus\{v_1\}}} \delta(wu_1v') \sqcap \delta(wu'v_1).$$

Now we take the greatest lower bound of the expression $\delta(wu_1v_2) \sqcap \delta(wu_2v_1)$ with both sides of the equality. Then we recognize that the same expression is always a factor of the series of least upper bound operations. This allows us to cancel out the series by absorption:

$$\begin{aligned}
&\delta(wu_1v_1) \sqcap \delta(wu_1v_2) \sqcap \delta(wu_2v_1) \\
&= \delta(wu_1v_2) \sqcap \delta(wu_2v_1) \sqcap \left(\delta(wu_1v_1) \sqcup \bigsqcup_{\substack{u' \in \mathfrak{Q}(U)\setminus\{u_1\} \\ v' \in \mathfrak{Q}(V)\setminus\{v_1\}}} \delta(wu_1v') \sqcap \delta(wu'v_1)\right) \\
&= \delta(wu_1v_2) \sqcap \delta(wu_2v_1) \sqcap \left(\delta(wu_1v_1) \sqcup \delta(wu_1v_2) \sqcap \delta(wu_2v_1) \sqcup \bigsqcup_{\substack{u'v' \in \mathfrak{Q}(U\setminus\{u_1\}\cup V\setminus\{v_1\})\setminus \\ \{u_2v_2\}}} \delta(wu_1v') \sqcap \delta(wu'v_1)\right) \\
&= \delta(wu_1v_2) \sqcap \delta(wu_2v_1);
\end{aligned}$$

thus $\delta(wu_1v_2) \sqcap \delta(wu_2v_1) \leq \delta(wu_1v_1)$.

By repeating the same reasoning on $\delta(wu_1v_2)$, $\delta(wu_2v_1)$, and $\delta(wu_2v_2)$ we conclude

$$\begin{cases}
\delta(wu_1v_2) \sqcap \delta(wu_2v_1) \leq \delta(wu_1v_1) \\
\delta(wu_1v_2) \sqcap \delta(wu_2v_1) \leq \delta(wu_2v_2) \\
\delta(wu_1v_1) \sqcap \delta(wu_2v_2) \leq \delta(wu_1v_2) \\
\delta(wu_1v_1) \sqcap \delta(wu_2v_2) \leq \delta(wu_2v_1),
\end{cases}$$

that, as $x \leq y \wedge z \leq w \Rightarrow x \sqcap z \leq y \sqcap w$, yields

$$\begin{cases}
\delta(wu_1v_2) \sqcap \delta(wu_2v_1) \leq \delta(wu_1v_1) \sqcap \delta(wu_2v_2) \\
\delta(wu_1v_1) \sqcap \delta(wu_2v_2) \leq \delta(wu_1v_2) \sqcap \delta(wu_2v_1).
\end{cases} \tag{B.10}$$

By antisymmetry, the latter entails $\delta(wu_1v_1) \sqcap \delta(wu_2v_2) = \delta(wu_1v_2) \sqcap \delta(wu_2v_1)$, in contradiction with (B.9). □

Proof of Proposition 5. Let $\langle F_1 \cup F_2, C_1, \Delta\rangle = \langle F_1, C_1, \Delta_1\rangle \otimes \langle F_2, C_2, \Delta_2\rangle$ and $\langle F_1 \cup F_2, C_1, \delta\rangle = \langle F_1, C_1, \delta_1\rangle \otimes \langle F_2, C_2, \delta_2\rangle$. We need to show that

$$\delta_1 = \mathrm{ext}\,(\Delta_1) \wedge \delta_2 = \mathrm{ext}\,(\Delta_2) \Rightarrow \delta = \mathrm{ext}\,(\Delta)\,.$$

Let us denote $W \cup U \cup V = F$. Let $wu \in \mathfrak{Q}(W \cup U)$ and $wv \in \mathfrak{Q}(W \cup V)$. Then consider all the pairs $\langle \psi_1, d_1\rangle \in \Delta_1$ and $\langle \psi_2, d_2\rangle \in \Delta_2$ such that $wu \in [\![\psi_1]\!]_{W\cup U} \wedge wv \in$

$[\![\psi_2]\!]_{W\cup V}$. Since all such ψ_1, ψ_2 guarantee $w \in [\![\psi_1|_W \wedge \psi_2|_W]\!]_W$, they surely fulfill the condition of line 6. Hence, by lines 7 and 8, there exists $\langle \psi, d_1 \sqcap d_2 \rangle \in \Delta$ such that $wuv \in [\![\psi]\!]_F$. We thus have the following result:

$$\begin{gathered} \langle \psi_1, d_1 \rangle \in \Delta_1 \wedge wu \in [\![\psi_1]\!]_{W\cup U} \wedge \langle \psi_2, d_2 \rangle \in \Delta_2 \wedge wv \in [\![\psi_2]\!]_{W\cup V} \\ \Leftrightarrow \\ \langle \psi, d_1 \sqcap d_2 \rangle \in \Delta \wedge wuv \in [\![\psi]\!]_F. \end{gathered} \tag{B.11}$$

We can now prove our thesis:

$$\begin{aligned} &\forall wuv \in \mathfrak{Q}(F),\ \delta(wuv) = \delta_1(wu) \sqcap \delta_2(wv) = \\ &= \operatorname{ext}(\Delta_1) \sqcap \operatorname{ext}(\Delta_2) = && \text{by Hypothesis} \\ &= \bigsqcup \{d_1 \mid \langle \psi_1, d_1 \rangle \in \Delta_1 \wedge wu \in [\![\psi_1]\!]_{W\cup U}\} \sqcap \bigsqcup \{d_2 \mid \langle \psi_2, d_2 \rangle \in \Delta_2 \wedge wv \in [\![\psi_2]\!]_{W\cup V}\} = && \text{by (5.3)} \\ &= \bigsqcup \{d_1 \sqcap d_2 \mid \langle \psi_1, d_1 \rangle \in \Delta_1 \wedge wu \in [\![\psi_1]\!]_{W\cup U} \wedge \langle \psi_2, d_2 \rangle \in \Delta_2 \wedge wv \in [\![\psi_2]\!]_{W\cup V}\} = && \text{by (B.6)} \\ &= \bigsqcup \{d \mid \langle \psi, d \rangle \in \Delta \wedge wuv \in [\![\psi]\!]_F\}. && \text{by (B.11)} \end{aligned}$$

Hence, by (5.3), we conclude $\delta = \operatorname{ext}(\Delta)$. □

Lemma 3. *Let $\Delta \subseteq \Psi(F) \times D$ and $X \subseteq F$. Then, $\forall q \in \mathfrak{Q}(F)$, $\exists! \langle \psi_X, \Delta_X \rangle \in$ PARTITION(Δ, X) s.t.*

$$q|_X \in [\![\psi_X]\!]_X \tag{B.12}$$

and

$$\operatorname{ext}(\Delta)(q) = \operatorname{ext}(\Delta_X)(q|_{F\setminus X}) \tag{B.13}$$

Proof. We recall that a DFD always covers the entire request space on which it is defined (Definition 26). Formally:

$$\bigcup \{[\![\psi]\!]_F \mid \langle \psi, \cdot \rangle \in \Delta\} = \mathfrak{Q}(F). \tag{B.14}$$

The first result (B.12) follows by observing that the main loop of Algorithm 5.10 removes all the possible overlaps among the X-projection of the RDs in Δ. Hence, at most one ψ_X will exist that matches every possible $q|_X$. More strongly, as consequence of (B.14), exactly one such $q|_X$ must exist. The second result (B.13) follows by substituting the definition of DFD extension (5.3). □

Proof of Proposition 7. $\Leftarrow$. We obtain the first half by contraposition.

$\textsc{Partition}(\Delta, W)$	$\textsc{Partition}(\Delta^i_W, U)$	$\textsc{Partition}(\Delta^j_U, V)$
$\langle \psi_W, \Delta_W \rangle$	$\langle \psi_U, \Delta_U \rangle$	$\langle \psi_V, \Delta_V = \{d_1, \ldots, d_n\} \rangle$
		$\langle \psi''_V, \Delta''_V = \{d''_1, \ldots, d''_m\} \rangle$
		$\ldots$
	$\langle \psi'_U, \Delta'_U \rangle$	$\langle \psi'_V, \Delta'_V = \{d'_1, \ldots, d'_l\} \rangle$
		$\langle \psi'''_V, \Delta'''_V = \{d'''_1 \ldots, d'''_k\} \rangle$
		$\ldots$
	$\ldots$	$\ldots$
$\ldots$	$\ldots$	$\ldots$

Table B.2: Partitioned Δ

Assume that Algorithm 5.9 returns **false** (Line 14). Hence, there exist $\langle \psi_W, \Delta_W \rangle$, $\langle \psi_U, \Delta_U \rangle$, $\langle \psi'_U, \Delta'_U \rangle$, $\langle \psi_V, \Delta_V \rangle$, $\langle \psi'_V, \Delta'_V \rangle$, $\langle \psi''_V, \Delta''_V \rangle$, $\langle \psi'''_V, \Delta'''_U \rangle$ such that:

$$\begin{cases} \langle \psi_W, \Delta_W \rangle \in \textsc{Partition}(\Delta, W) \\ \langle \psi_U, \Delta_U \rangle, \langle \psi'_U, \Delta'_U \rangle \in \textsc{Partition}(\Delta_W, U) \\ \langle \psi_V, \Delta_V \rangle, \langle \psi''_V, \Delta''_V \rangle \in \textsc{Partition}(\Delta_U, V) \\ \langle \psi'_V, \Delta'_V \rangle, \langle \psi'''_V, \Delta'''_V \rangle \in \textsc{Partition}(\Delta'_U, V) \end{cases}$$

and (Line 11):

$$[\![\psi'_V \wedge \psi''_V]\!]_V \neq \emptyset \wedge [\![\psi_V \wedge \psi'''_V]\!]_V \neq \emptyset \tag{B.15}$$

and (Lines 9, 12 and 13):

$$\bigsqcup_{d \in \Delta_V} d \sqcap \bigsqcup_{d' \in \Delta'_V} d' \neq \bigsqcup_{d'' \in \Delta''_V} d'' \sqcap \bigsqcup_{d''' \in \Delta'''_V} d'''. \tag{B.16}$$

Note that for all $\langle \psi, d \rangle \in \Delta_V$ (and likewise for $\Delta'_V, \Delta''_V, \Delta'''_V$), ψ must equal the empty sequence $\langle\rangle$. This is a consequence of partitioning $\Delta_U \subseteq \Psi(V) \times D$ (resp. Δ'_U) on V. For the sake of clarity, we abuse notation and simply write d for $\langle \langle\rangle, d \rangle$. According to (5.3), we can then rewrite (B.16) as:

$$\mathrm{ext}(\Delta_V)(\langle\rangle) \sqcap \mathrm{ext}(\Delta'_V)(\langle\rangle) \neq \mathrm{ext}(\Delta''_V)(\langle\rangle) \sqcap \mathrm{ext}(\Delta'''_V)(\langle\rangle). \tag{B.17}$$

The partitions of Δ are represented in Table B.2.

Because of (B.12) and (B.15) we can choose $q, q', q'', q''' \in \mathfrak{Q}(W \cup U \cup V)$ such that:

$$\begin{cases} q|_W, q'|_W, q''|_W, q'''|_W \in [\![\psi_W]\!]_W \\ q|_U, q''|_U \in [\![\psi_U]\!]_U \\ q'|_U, q'''|_U \in [\![\psi'_U]\!]_U \\ q|_V, q'''|_V \in [\![\psi_V \wedge \psi'''_V]\!]_V \\ q'|_V, q''|_V \in [\![\psi'_V \wedge \psi''_V]\!]_V \end{cases}$$

moreover, as a consequence of (B.13), we have:

$$\begin{cases} \mathrm{ext}(\Delta)(q) = \mathrm{ext}(\Delta_W)(q|_{U\cup V}) = \mathrm{ext}(\Delta_U)(q|_{U\cup V}|_V) = \mathrm{ext}(\Delta_V)(q|_{U\cup V}|_V|_\varnothing) = \mathrm{ext}(\Delta_V)(\langle\rangle) \\ \mathrm{ext}(\Delta)(q') = \mathrm{ext}(\Delta_W)(q'|_{U\cup V}) = \mathrm{ext}(\Delta'_U)(q'|_{U\cup V}|_V) = \mathrm{ext}(\Delta'_V)(q'|_{U\cup V}|_V|_\varnothing) = \mathrm{ext}(\Delta'_V)(\langle\rangle) \\ \mathrm{ext}(\Delta)(q'') = \mathrm{ext}(\Delta_W)(q''|_{U\cup V}) = \mathrm{ext}(\Delta_U)(q''|_{U\cup V}|_V) = \mathrm{ext}(\Delta''_V)(q''|_{U\cup V}|_V|_\varnothing) = \mathrm{ext}(\Delta''_V)(\langle\rangle) \\ \mathrm{ext}(\Delta)(q''') = \mathrm{ext}(\Delta_W)(q'''|_{U\cup V}) = \mathrm{ext}(\Delta'_U)(q'''|_{U\cup V}|_V) = \mathrm{ext}(\Delta'''_V)(q'''|_{U\cup V}|_V|_\varnothing) = \mathrm{ext}(\Delta'''_V)(\langle\rangle) \end{cases}$$

that allows to rewrite (B.17) as $\mathrm{ext}(\Delta)(q) \sqcap \mathrm{ext}(\Delta)(q') \neq \mathrm{ext}(\Delta)(q'') \sqcap \mathrm{ext}(\Delta)(q''')$, which, by Definition 27, is equivalent to $\mathrm{ext}(\Delta) \not\models W \twoheadrightarrow V$.

$\Rightarrow$. For the second half, we reason again by contraposition.

Assume $\mathrm{ext}(\Delta) \not\models W \twoheadrightarrow V$. Then, there exist $q, q', q'', q''' \in \mathfrak{Q}(W \cup U \cup V)$ such that:

$$\begin{cases} q|_W = q'|_W = q''|_W = q'''|_W \\ q|_U = q''|_U \wedge q'|_U = q'''|_U \\ q|_V = q'''|_V \wedge q'|_V = q''|_V \end{cases}$$

and

$$\mathrm{ext}(\Delta)(q) \sqcap \mathrm{ext}(\Delta)(q') \neq \mathrm{ext}(\Delta)(q'') \sqcap \mathrm{ext}(\Delta)(q'''). \tag{B.18}$$

By (B.12) then, there is a unique $\langle \psi_W, \Delta_W \rangle$ such that $q|_W, q'|_W, q''|_W, q'''|_W \in [\![\psi_W]\!]_W$.

Let $\langle \psi_U, \Delta_U \rangle \in \textsc{Partition}(\Delta_W, U)$. If $q|_U, q'|_U \in [\![\psi_U]\!]_U$, then $q''|_U, q'''|_U \in [\![\psi_U]\!]_U$ too, as $q|_U = q''|_U$ and $q'|_U = q'''|_U$. Similarly, as $q|_V = q'''|_V$ and $q'|_V = q''|_V$, there exist two unique $\langle \psi_V, \Delta_V \rangle, \langle \psi'_V, \Delta'_V \rangle \in \textsc{Partition}(\Delta_U, V)$ such that $q|_V, q'''|_V \in [\![\psi_V]\!]_V$ and $q'|_V, q''|_V \in [\![\psi'_V]\!]_V$. By (B.13), we then rewrite (B.18) as

$$\mathrm{ext}(\Delta_V)(\langle\rangle) \sqcap \mathrm{ext}(\Delta'_V)(\langle\rangle) \neq \mathrm{ext}(\Delta_V)(\langle\rangle) \sqcap \mathrm{ext}(\Delta'_V)(\langle\rangle),$$

a contradiction. Hence it must be $q|_U \in [\![\psi_U]\!]_U$ and $q'|_U \in [\![\psi'_U]\!]_U$ for some $\langle \psi_U, \Delta_U \rangle, \langle \psi'_U, \Delta'_U \rangle \in \textsc{Partition}(\Delta_W, U)$.

Let now $\langle \psi_V, \Delta_V \rangle, \langle \psi''_V, \Delta''_V \rangle \in \textsc{Partition}(\Delta_U, V)$ and $\langle \psi'_V, \Delta'_V \rangle, \langle \psi'''_V, \Delta'''_V \rangle \in \textsc{Partition}(\Delta'_U, V)$ such that:

$$\begin{cases} q|_V, q'''|_V \in [\![\psi_V]\!]_V \wedge q|_V, q'''|_V \in [\![\psi'''_V]\!]_V \\ q'|_V, q''|_V \in [\![\psi'_V]\!]_V \wedge q'|_V, q''|_V \in [\![\psi''_V]\!]_V. \end{cases}$$

It is clear that such $\psi_V, \psi'_V, \psi''_V, \psi'''_V$ satisfy the condition of Line 11 of Algorithm 5.9. Moreover, by applying (B.13), we can rewrite (B.18) as

$$\mathrm{ext}(\Delta_V)(\langle\rangle) \sqcap \mathrm{ext}(\Delta'_V)(\langle\rangle) \neq \mathrm{ext}(\Delta''_V)(\langle\rangle) \sqcap \mathrm{ext}(\Delta'''_V)(\langle\rangle),$$

which means that the condition of Line 13 is satisfied too. As a consequence, the algorithm returns **false** (Line 14). □

C

Synthesis of Experimental Input Datasets

▷ *This appendix provides details on the generation of the input datasets used for the experiments described in Chapters 4 and 5.* ◁

Chapter Outline

C.1 Generation of JEE Security Contraint Configurations

Algorithm GENSCSET($\mathcal{U}, \mathcal{M}, \mathcal{R}$)
  Input: Finite domains $\mathcal{U}, \mathcal{M}, \mathcal{R}$
  Output: Set of configurations C
1 $C \leftarrow \emptyset$;
2 $RM \leftarrow (\wp(\mathcal{R}) \cup \top) \times \wp(\mathcal{M})$; // $\wp(X)$ denotes the set of subsets (powerset) of X.
3 **foreach** $U \in \wp^+(\wp^+(\mathcal{U}))$ **do** // $\wp^+(X)$ is the set of non-empty subsets of X.
4 **foreach** $P \in \text{perm}(RM, |U|)$ **do** // $\text{perm}(X, k)$ is the set of all k-permutations of elements in X.
5 $C \leftarrow C \cup \{\text{GENSC}(P, U)\}$;
6 **return** C;

Algorithm GENSC(P, U)
  Input: $P = \{\langle R_i, M_i \rangle\}$ set of pairs with R_i, M_i sets of roles and methods, and U set of sets of URLs. Input sets have equal cardinality $|P| = |U|$.
  Output: SC configuration composed of $|U|$ security constraints.
7 $SC \leftarrow$ "";
8 **for** $i = 0$ **to** $|U| - 1$ **do**
9 $\langle R, M \rangle \leftarrow P[i]$;
10 $SC \leftarrow SC +$ "{";
11 **foreach** $u \in U[i]$ **do**
12 $SC \leftarrow SC + u +$ ", ";
13 $SC \leftarrow SC +$ "} [";
14 **foreach** $m \in M$ **do**
15 $SC \leftarrow SC + m +$ ", ";
16 $SC \leftarrow SC +$ "] ";
17 **if** $R \neq \bot$ **then**
18 $SC \leftarrow SC +$ "<";
19 **foreach** $r \in R$ **do**
20 $SC \leftarrow SC + r +$ ", ";
21 $SC \leftarrow SC +$ "> ";
22 **return** SC;

Figure C.1: Generation of security constraint configurations.

C.2 Synthesis of Decision Function Descriptors (DFDs)

C.2.1 Controlling the Average Degree of Overlap

Consider the procedure GENINTV reported in Figure 5.14. In this algorithm n intervals are generated, having deterministic equally-spaced centers $\{c_i\}$ and random diameters $\{d_i\}$. The probability distribution of the diameters is log-normal (as such, only positive values are possible): its mean equals the distance between two centers and its variance can be tuned via the parameter ν.

Intuitively, we expect increasing values of ν to yield, in average, an increasing degree of overlap. To confirm this intuition, we measured the minimum, average and maximum degrees of overlap as a function of the ν parameter. We repeated the experiment for increasing values of n by keeping the domain upper bound fixed to $M = 10000$.

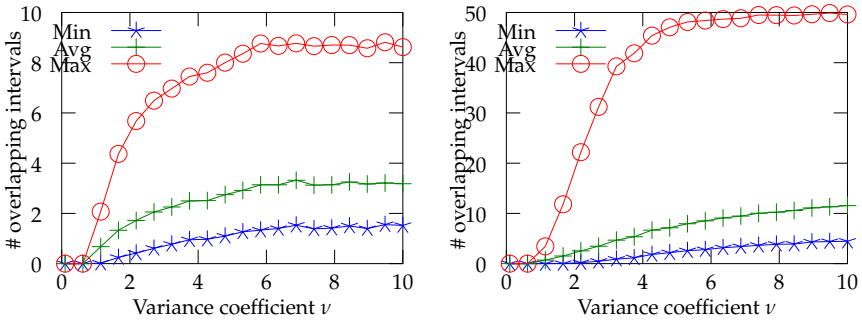
The results of this experiment, depicted in Figures C.2a to C.2d, confirm the expected correlation between the variance coefficient and the degree of overlap of intervals. Note that varying the value of M would only change the average interval size (M/n), but it would not modify how frequently intervals overlap with each other, as long as M is sufficiently greater than n to avoid integer division problems.

By inspecting Figure C.2, we observe that the more intervals are generated, the more they tend to overlap with each other for the same values of ν. E.g., for $\nu = 10$, each interval overlaps in average with slightly more than 3 others, when $n = 10$, and slightly less than 50, when $n = 500$. In order to control the average degree of overlap independently from the number of generated intervals, we repeated the aforementioned experiment for more fine-grained values of n. As such, we were able to empirically determine the family of functions $\nu_k(n)$ with the following property: every ν_k maps a number n to the variance coefficient needed to generate n intervals such that each one will overlap with k others in average. Table C.3 reports the estimated values of such functions for $n \in \{10, 20, \ldots, 200\}$ and $k \in \{1, 2, 3, 5, 10\}$.

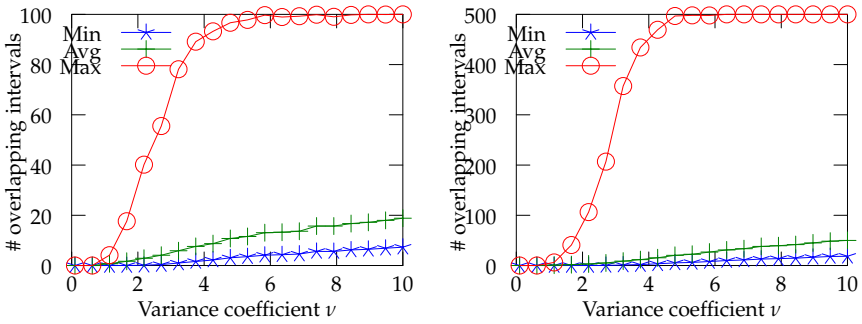
Given a field f, with an associated domain of integers $\mathrm{dom}(f) = [0, \ldots, M]$, we are now in a position to generate a random unidimensional DFD $\Delta_{synth} \subseteq \Psi(\{f\}) \times D$ containing n RDs with an average degree of overlap of k. To do so, it is sufficient to associate each interval generated by the procedure GENINTV$(n, M, \nu_k(n))$ with a decision selected randomly in the decision space D.

C.2.2 Multidimensional Overlap Density and Average Degree

It is convenient to think about the overlap between two RDs as a Bernoulli trial with success probability p_e, with $1 \leq e \leq \binom{n}{2}$. Note that the index e ranges over all the 2-



(a) Number of generated intervals $n = 10$ (b) Number of generated intervals $n = 50$

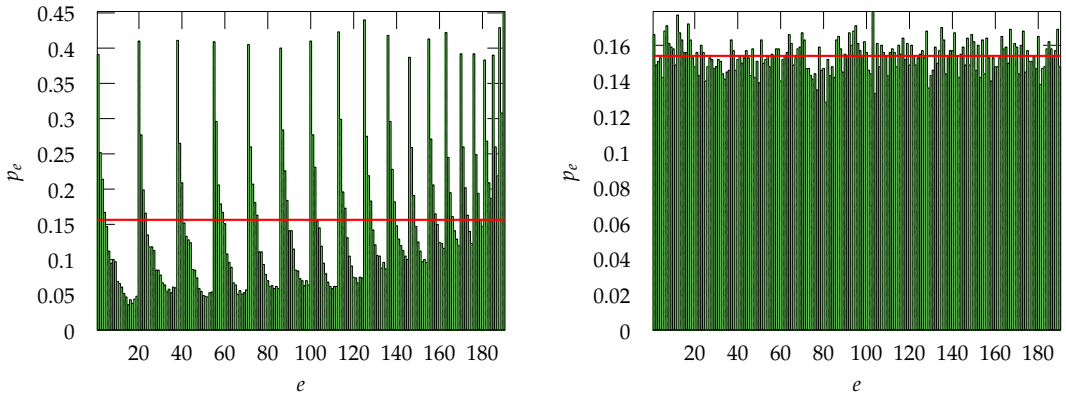


(c) Number of generated intervals $n = 100$(d) Number of generated intervals $n = 500$

Figure C.2: Minimum, average and maximum number of overlapping intervals as a function of the variance coefficient ν when generating n intervals in the domain $0, \ldots, 10000$

n	$\nu_1(n)$	$\nu_2(n)$	$\nu_3(n)$	$\nu_5(n)$	$\nu_{10}(n)$
10	1.50	2.50	5.00	10.00	100.00
20	1.25	2.20	3.00	6.00	100.00
30	1.23	2.00	2.55	4.20	25.00
40	1.22	2.00	2.50	3.50	10.00
50	1.22	2.00	2.50	3.30	7.00
60	1.22	1.90	2.40	3.20	6.50
70	1.22	1.88	2.40	3.15	6.00
80	1.22	1.875	2.40	3.10	5.50
90	1.22	1.875	2.40	3.05	5.00
100	1.22	1.875	2.40	3.00	4.75
110	1.22	1.875	2.39	2.90	4.50
120	1.22	1.875	2.38	2.80	4.50
130	1.22	1.875	2.37	2.80	4.30
140	1.22	1.875	2.36	2.70	4.20
150	1.22	1.875	2.35	2.70	4.20
160	1.22	1.87	2.30		
170	1.22	1.87	2.25		
180	1.22	1.87	2.25		
190	1.22	1.87	2.25		
200	1.22	1.87	2.25		

Table C.3: Estimated $\nu_k(n)$ functions for $k \in \{1, 2, 3, 5, 10\}$



(a) GENINTV without shuffling

(b) GENINTV with shuffling

Figure C.4: Probabilities of overlap p_e with e ranging over all the possible 2-choices of the intervals generated by GENINTV (with and without shuffling).

choices of n RDs or, equivalently, all the possible edges of the graph having n RDs as vertices.

The reason for distinguishing the success probabilities of the different trials, is that we naturally expect them to distribute differently in a generic outcome of the GENINTV algorithm. Indeed, as the generated intervals are centered on equally-spaced points, adjacent intervals are more likely to overlap than those being far apart. In other words not all the pairs of intervals have the same probability to overlap, hence each pair (equivalently each edge of the graph) is associated with a different success probability p_e.

If all the trials are independent, it is well known that the total amount of successes over all trials follows a Poisson binomial distribution. Its mean equals the sum of all individual probabilities $\mu = \sum_{e=1}^{\binom{n}{2}} p_e$ and, in our setting, represents the expected number of overlapping pairs of intervals (equivalently edges of the graph). It follows that the expected overlap density of a set of n unidimensional RDs, as generated by the GENINTV procedure, can be expressed in terms of the individual probabilities p_e as:

$$\rho = \frac{\mu}{\binom{n}{2}} = \frac{2\sum_{e=1}^{\binom{n}{2}} p_e}{n(n-1)}. \tag{C.1}$$

Note that, if all the overlap probabilities equal each other, i.e., $p_1 = p_2 = \ldots = p_{\binom{n}{2}} = p$, the latter reduces to $\rho = p$. This means that, in this case, the overlap probability of each pair of RDs coincides with the overall overlap density. This situation can be achieved by artificially shuffling the relative order of the intervals generated by GENINTV. By depicting the result of the analysis of a large number of executions of GENINTV, Figure C.4 shows how the p_e probabilities tend to converge to the same

value when shuffling is applied. The values have been estimated by counting the frequencies of overlaps over 1000 executions of the algorithm with the following fixed choice of parameters $n = 20, M = 10000, \nu = 3$.

Let $\psi = \langle \varphi_1, \ldots, \varphi_m \rangle$ and $\psi' = \langle \varphi'_1, \ldots, \varphi'_m \rangle$ be two m-dimensional RDs. According to Definition 28, they overlap with each other when the extension of their conjunction is not empty. According to Definition 25, this is true if and only if $[\![\varphi_i \wedge \varphi'_i]\!] \neq \emptyset$ for $1 \leq i \leq m$, i.e., if all the m dimensions do simultaneously overlap.

We know that the event of a pair of RDs overlapping on each individual dimension is determined by an independent Bernoulli trial. For the e^{th} pair and the i^{th} dimension we denote the corresponding overlap probability as $p_{e,i}$, with $1 \leq e \leq \binom{n}{2}$ and $1 \leq i \leq m$. The probability of the e^{th} pair overlapping on all dimensions simultaneously is then equal to the product $p_e = \prod_{i=1}^{m} p_{e,i}$. By substituting in (C.1) we obtain the overlap density of a set of n m-dimensional RDs:

$$\rho_{\text{mul}} = \frac{2 \sum_{e=1}^{\binom{n}{2}} \prod_{i=1}^{m} p_{e,i}}{n(n-1)}. \tag{C.2}$$

The last expression can be used in general to estimate the multidimensional overlap density (and therefore the average degree of overlap) from the statistical properties of each dimension. In order to use it, however, one has to estimate the overlap probability $p_{e,i}$ for every dimension of every possible pair of RDs. This is not however always necessary. Under specific circumstances it is in fact possible to reduce (C.2) to a simpler expression. As first simplifying assumption we require all the dimensions to be generated independently by executing GENINTV with the same fixed choice of parameters. In this case, all the dimensions clearly need to exhibit the same statistical behaviour. In particular, the overlap probabilities will be the same over all dimensions, i.e., $p_{e,1} = p_{e,2} = \ldots = p_{e,m} = p_e$. Moreover, as shown previously in this section, if we shuffle each set of generated intervals, all the overlap probabilities p_e tend to coincide with the unidimensional overlap density ρ.

Under these assumptions, we can then rewrite (C.2) as:

$$\rho_{\text{mul}} = \frac{2 \sum_{e=1}^{\binom{n}{2}} {p_e}^m}{n(n-1)} = \frac{2\binom{n}{2}\rho^m}{n(n-1)} = \rho^m. \tag{C.3}$$

As this latest expression only involves overlap densities, we can as well rewrite it in terms of the respective average degrees of overlap:

$$k_{\text{mul}} = (n-1)\rho_{\text{mul}} = (n-1)\left(\frac{k}{n-1}\right)^m = \frac{k^m}{(n-1)^{m-1}}. \tag{C.4}$$

Bibliography

[7Safe2010] 7Safe & University of Bedfordshire. *UK Security Breach Investigations Report 2010*. Technical report. 2010 (cited on page 10).

[Abadi2003] Martín Abadi. "Logic in Access Control." In: *LICS'03: 18th IEEE Symposium on Logic in Computer Science, Ottawa, Canada*. IEEE Computer Society, 2003, pages 228– (cited on page 136).

[Ahn2010] Gail-Joon Ahn, Hongxin Hu, Joohyung Lee & Yunsong Meng. "Representing and Reasoning about Web Access Control Policies". In: *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*. July 2010, pages 137–146 (cited on pages 14, 80, 96).

[Alfaro2007] J.G. Alfaro, F. Cuppens & N. Cuppens-Boulahia. "Aggregating and Deploying Network Access Control Policies". In: *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*. Apr. 2007, pages 532–542 (cited on page 14).

[Alfaro2008] J.G. Alfaro, N. Boulahia-Cuppens & F. Cuppens. "Complete analysis of configuration rules to guarantee reliable network security policies". English. In: *International Journal of Information Security* 7.2 (2008), pages 103–122 (cited on pages 12, 14, 132, 147).

[Alfaro2013] Joaquín Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, Salvador Martínez Perez & Jordi Cabot. "Management of stateful firewall misconfiguration." In: *Computers & Security* 39 (2013), pages 64–85 (cited on page 148).

[AlShaer2004] E. Al-Shaer & H. Hamed. "Modeling and Management of Firewall Policies". In: *Network and Service Management, IEEE Transactions on* 1.1 (Apr. 2004), pages 2–10 (cited on pages 32, 33).

[AlShaer2005] E. Al-Shaer, H. Hamed, R. Boutaba & M. Hasan. "Conflict classification and analysis of distributed firewall policies". In: *IEEE Journal on Selected Areas in Communications* 23(10) (2005), pages 2069–2084 (cited on pages 12, 14, 17, 103, 132, 147).

[AlShaer2009] E. Al-Shaer, W. Marrero, A. El-Atawy & K. Elbadawi. "Network configuration in a box: towards end-to-end verification of network reachability and security". In: *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*. Oct. 2009, pages 123–132 (cited on page 14).

[AlShaer2011] Ehab Al-Shaer. "Security Automation Research: Challenges and Future Directions". In: *IAnewsletter* 14.4 (2011), pages 14–18 (cited on pages 11, 103).

[ASF2012] Apache Software Foundation. *Tomcat Bug 53801: Nondeterministic behaviour of security constraints*. 2012. URL: `https://issues.apache.org/bugzilla/show_bug.cgi?id=53801` (visited on 02/06/2014) (cited on pages 95, 143).

[ASF2014a] Apache Software Foundation. *The Apache Tomcat Connector – Generic HowTo*. URL: `http://tomcat.apache.org/connectors-doc/generic_howto/proxy.html` (visited on 02/06/2014) (cited on page 41).

[ASF2014b] Apache Software Foundation. *mod_ssl – Apache HTTP Server Documentation*. 2014. URL: `http://httpd.apache.org/docs/current/mod/mod_ssl.html` (visited on 02/06/2014) (cited on page 7).

[Baker2012] Jonathan Baker, Matthew Hansbury & Daniel Haynes. *The OVAL Language Specification (Version 5.10.1)*. Specification. MITRE, Jan. 2012 (cited on pages 39, 46, 50, 60, 70).

[Barrere2012] M. Barrere, R. Badonnel & O. Festor. "Towards the assessment of distributed vulnerabilities in autonomic networks and systems". In: *Network Operations and Management Symposium (NOMS), 2012 IEEE*. Apr. 2012, pages 335–342 (cited on page 74).

[Basile2012] C. Basile, A. Cappadonia & Antonio Lioy. "Network-Level Access Control Policy Analysis and Transformation". In: *Networking, IEEE/ACM Transactions on* 20.4 (2012), pages 985–998 (cited on pages 12, 14, 132, 135, 147).

[Basile2013a] Cataldo Basile, Matteo Maria Casalino, Simone Mutti & Stefano Paraboschi. In: John Vacca. *Computer and Information Security Handbook*. Edited by John Vacca. 2nd. Morgan Kaufmann, June 2013. Chapter Detection of conflicts in security policies (cited on page 99).

[Basile2013b] Cataldo Basile, Marco Vallini & Matteo Casalino. *D3.7 – ENFORCEABILITY ANALYSIS*. Deliverable. FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu), Jan. 2013. URL: http://posecco.eu/fileadmin/POSECCO/user_upload/deliverables/D3.7_EnforceabilityAnalysis.pdf (visited on 02/06/2014) (cited on pages 99, 137).

[Becker2006] Moritz Y. Becker, Cédric Fournet & Andrew D. Gordon. *SecPAL: Design and Semantics of a Decentralized Authorization Language*. Technical report MSR–TR–2006–120. Microsoft Research, Roger Needham Building, 7 J.J. Thomson Avenue,Cambridge, CB3 0FB, United Kingdom: Microsoft Research UK, 2006 (cited on page 136).

[Behl2012] A. Behl & K. Behl. "An analysis of cloud computing security issues". In: *Information and Communication Technologies (WICT), 2012 World Congress on*. 2012, pages 109–114 (cited on page 13).

[Bellovin2009] S.M. Bellovin & R. Bush. "Configuration management and security". In: *Selected Areas in Communications, IEEE Journal on* 27.3 (2009), pages 268–274 (cited on page 11).

[Bertino2003] Elisa Bertino, Barbara Catania, Elena Ferrari & Paolo Perlasca. "A logical framework for reasoning about access control models". In: *ACM Transactions on Information & System Security* 6.1 (2003), pages 71–127 (cited on page 136).

[Bertino2006] Elisa Bertino, Anna C. Squicciarini, Ivan Paloscia & Lorenzo Martino. "Ws-AC: A Fine Grained Access Control System for Web Services". In: *World Wide Web* 9 (2 June 2006), pages 143–171 (cited on page 96).

[Bertossi2011] Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011 (cited on page 136).

[Bettan2012] Olivier Bettan, Serena Ponta, Kreshnik Musaraj & Matteo Casalino. *D4.8 – PROTOTYPE: STANDARDIZED AUDIT INTERFACE*. Deliverable. FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu), Oct. 2012. URL: http://posecco.eu/fileadmin/POSECCO/user_upload/deliverables/D4.8_Prototype_Standardized_Audit_Interface_v1.0.pdf (visited on 02/06/2014) (cited on pages 76, 99, 142).

[Bishop2006] Matt Bishop & Sean Peisert. "Your Security Policy is What??" 2006. URL: http://www.cs.ucdavis.edu/~peisert/research/security-policy-report.pdf (visited on 02/06/2014) (cited on page 11).

[Bonatti2002] Piero Bonatti, Sabrina De Capitani di Vimercati & Pierangela Samarati. "An algebra for composing access control policies". In: *ACM Trans. Inf. Syst. Secur.* 5 (1 Feb. 2002), pages 1–35 (cited on pages 13, 103).

[Bonatti2009] PieroAndrea Bonatti, JuriLuca Coi, Daniel Olmedilla & Luigi Sauro. "Rule-Based Policy Representations and Reasoning". In: *Semantic Techniques for the Web*. Edited by François Bry & Jan Maluszyński. Volume 5500. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pages 201–232 (cited on page 13).

[Brewer1989] D.F.C. Brewer & M.J. Nash. "The Chinese Wall security policy". In: *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*. May 1989, pages 206–214 (cited on page 147).

[Bruns2011] Glenn Bruns & Michael Huth. "Access control via belnap logic: Intuitive, expressive, and analyzable policy composition". In: *ACM Trans. Inf. Syst. Secur.* 14.1 (June 2011), 9:1–9:27 (cited on pages 110, 133).

[Bryans2005] Jery Bryans. "Reasoning about XACML policies using CSP". In: *SWS '05: Proceedings of the 2005 workshop on Secure web services*. Fairfax, VA, USA: ACM, 2005, pages 28–35 (cited on pages 14, 80, 96).

[BT2004] British Telecommunications. *Security and Business Continuity Solutions From BT: Thriving in the Age of the Digital Networked Economy*. Technical report. 2004 (cited on page 10).

[Buttner2009] A. Buttner & N. Ziring. *Common Platform Enumeration (CPE) - Specification.* Specification. MITRE, 2009 (cited on page 47).

[Casalino2012a] Matteo Maria Casalino, Michele Mangili, Henrik Plate & Serena Elisa Ponta. "Detection of Configuration Vulnerabilities in Distributed (Web) Environments". In: *SecureComm*. Edited by Angelos D. Keromytis & Roberto Di Pietro. Volume 106. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Acceptance rate of 28%. Springer, 2012, pages 131–148 (cited on page 76).

[Casalino2012b] Matteo Maria Casalino, Henrik Plate & Serena Elisa Ponta. "Configuration Assessment as a Service". In: *DPM/SETOP*. Edited by Roberto Di Pietro, Javier Herranz, Ernesto Damiani & Radu State. Volume 7731. Lecture Notes in Computer Science. Springer, 2012, pages 217–226 (cited on page 76).

[Casalino2012c] Matteo Maria Casalino, Romuald Thion & Mohand-Said Hacid. "Access Control Configuration for J2EE Web Applications: A Formal Perspective". In: *TrustBus*. Edited by Simone Fischer-Hübner, Sokratis K. Katsikas & Gerald Quirchmayr. Volume 7449. Lecture Notes in Computer Science. Springer, 2012, pages 30–35 (cited on page 99).

[Casalino2013a] Matteo Maria Casalino & Romuald Thion. "Extending Multi-valued Dependencies for Access Control Policy Refactoring". In: *29eme journées Bases de Données Avancées (BDA)*. Conference without formal proceedings. Oct. 2013. URL: `http://liris.cnrs.fr/publis/?id=5601` (visited on 07/01/2014) (cited on page 137).

[Casalino2013b] Matteo Maria Casalino & Romuald Thion. "Refactoring Multi-Layered Access Control Policies Through (De)Composition". In: *Proceedings of the 9th international conference on Network and Service Management, CNSM'13, Zurich, Switzerland*. Acceptance rate of 18%. IEEE Computer Society, Oct. 2013, pages 243–250. URL: `http://www.cnsm-conf.org/2013/documents/papers/CNSM/p243-casalino.pdf` (visited on 07/01/2014) (cited on page 137).

[CFEngine] *CFEngine*. URL: `http://www.cfengine.org` (visited on 02/06/2014) (cited on page 74).

[Chan2013] Shing Wai Chan & Rajiv Mordani. *Java Servlet Specification, Version 3.1*. Specification JSR-340. Oracle, Apr. 2013 (cited on page 97).

[Cheikes2011] B. A. Cheikes, D. Waltermire & K. Scarfone. *Common Platform Enumeration: Naming Specification Version 2.3*. Specification. NIST, 2011 (cited on page 47).

[Chen2008] Xiuzhen Chen, Qinghua Zheng & Xiaohong Guan. "An OVAL-based active vulnerability assessment system for enterprise computer networks". In: *ISF* (Nov. 2008), pages 573–588 (cited on page 73).

[Covington2002] M.J. Covington, P. Fogla, Zhiyuan Zhan & M. Ahamad. "A context-aware security architecture for emerging applications". In: *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*. 2002, pages 249–258 (cited on page 12).

[Coward2003] Danny Coward & Yutaka Yoshida. *Java Servlet Specification, Version 2.4*. Specification JSR-154. Sun Microsystems, Inc, Nov. 2003 (cited on pages 30, 31, 79, 82, 88, 89, 97, 143).

[Crampton2010] Jason Crampton & Michael Huth. "A Framework for the Modular Specification and Orchestration of Authorization Policies". In: *NordSec*. Edited by Tuomas Aura, Kimmo Järvinen & Kaisa Nyberg. Volume 7127. Lecture Notes in Computer Science. Springer, 2010, pages 155–170 (cited on page 147).

[Crampton2012a] Jason Crampton & Charles Morisset. "PTaCL: A Language for Attribute-Based Access Control in Open Systems". In: *POST*. Edited by Pierpaolo Degano & Joshua D. Guttman. Volume 7215. Lecture Notes in Computer Science. Springer, 2012, pages 390–409 (cited on page 96).

[Crampton2012b] Jason Crampton & Charles Morisset. "Towards A Generic Formal Framework for Access Control Systems". In: *CoRR* abs/1204.2342 (2012) (cited on pages 16, 98, 105, 134, 147).

[Craven2010] R. Craven, J. Lobo, E. Lupu, A. Russo & M. Sloman. "Decomposition techniques for policy refinement". In: *Network and Service Management (CNSM), 2010 International Conference on*. 2010, pages 72–79 (cited on pages 14, 133).

[Craven2011] Robert Craven, Jorge Lobo, Emil Lupu, Alessandra Russo & Morris Sloman. "Policy refinement: Decomposition and operationalization for dynamic domains". In: *CNSM*. IEEE, 2011, pages 1–9 (cited on page 133).

[CSIS2008] CSIS. *Securing Cyberspace for the 44th Presidency*. Technical report. Washington, DC: Center for Strategic and International Studies, Dec. 2008. URL: `http://csis.org/files/media/csis/pubs/081208_securingcyberspace_44.pdf` (visited on 02/06/2014) (cited on page 10).

[Cuppens2004] Frédéric Cuppens, Nora Cuppens-Boulahia, Thierry Sans & Alexandre Miège. "A Formal Approach to Specify and Deploy a Network Security Policy". In: *Formal Aspects in Security and Trust*. Edited by Theodosis Dimitrakos & Fabio Martinelli. Springer, 2004, pages 203–218 (cited on page 11).

[CVE-2003-0042] National Vulnerability Database. *CVE-2003-0042*. 2003. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2003-0042 (visited on 02/06/2014) (cited on page 48).

[CVE-2010-0738] National Vulnerability Database. *CVE-2010-0738*. 2010. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0738 (visited on 02/06/2014) (cited on pages 80, 82).

[CVE-2011-3190] National Vulnerability Database. *CVE-2011-3190*. 2011. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3190 (visited on 02/06/2014) (cited on page 40).

[CVE-2012-0287] National Vulnerability Database. *CVE-2012-0287*. 2014. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0287 (visited on 02/06/2014) (cited on page 48).

[CVE-2012-0392] National Vulnerability Database. *CVE-2012-0392*. 2012. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0392 (visited on 02/06/2014) (cited on page 41).

[CVE-2014-0160] National Vulnerability Database. *CVE-2014-0160*. 2014. URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160 (visited on 04/23/2014) (cited on page 48).

[Davis2012] Michael A. Davis. *2012 Strategic Security Survey*. Technical report. Information Week, May 2012 (cited on page 9).

[Davy2008a] S. Davy, B. Jennings & J. Strassner. "Application Domain Independent Policy Conflict Analysis Using Information Models". In: *IEEE/IFIP Network Operations and Management Symposium (NOMS 2008)*. 2008 (cited on pages 13, 14, 103, 147).

[Davy2008b] Steven Davy, Brendan Jennings & John Strassner. "The policy continuum-Policy authoring and conflict analysis". In: *Computer Communications* 31.13 (2008), pages 2981–2995 (cited on pages 14, 133).

[Demetz2013] Lukas Demetz, Ronald Maier, Markus Manhart & Henrik Plate. *D1.7 – FINAL PROJECT EVALUATION*. Deliverable. FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu), Dec. 2013. URL: http://posecco.eu/fileadmin/POSECCO/user_upload/deliverables/D1.7-Final_Project_

Evaluation_v1.0.pdf (visited on 02/06/2014) (cited on pages 76, 142).

[DMTF2000] DMTF. *Common Information Model (CIM) Core Model*. Specification DSP0111. Distributed Management Task Force, 2000 (cited on page 70).

[DMTF2007] DMTF. *CIM Query Language Specification*. Specification DSP0202. Distributed Management Task Force, 2007 (cited on page 70).

[DMTF2010] DMTF. *Configuration Management Database (CMDB) Federation Specification Version 1.0.1*. Specification DSP0252. Distributed Management Task Force, Apr. 2010 (cited on page 75).

[Fabian2010] Benjamin Fabian, Seda Gürses, Maritta Heisel, Thomas Santen & Holger Schmidt. "A comparison of security requirements engineering methods". English. In: *Requirements Engineering* 15.1 (2010), pages 7–40 (cited on page 13).

[Fagin1977] Ronald Fagin. "Multivalued dependencies and a new normal form for relational databases". In: *ACM Trans. Database Syst.* 2.3 (Sept. 1977), pages 262–278 (cited on pages 135, 137, 144).

[Fisler2005] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich & Michael Carl Tschantz. "Verification and change-impact analysis of access-control policies". In: *ICSE*. Edited by Gruia-Catalin Roman, William G. Griswold & Bashar Nuseibeh. ACM, 2005, pages 196–205 (cited on pages 16, 97, 142).

[Foley2011] Simon N. Foley & William M. Fitzgerald. "Management of security policy configuration using a Semantic Threat Graph approach". In: *J. Comput. Secur.* 19 (3 Aug. 2011), pages 567–605 (cited on page 146).

[Frank2010] Mario Frank, Joachim M. Buhmann & David Basin. "On the Definition of Role Mining". In: *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*. SACMAT '10. New York, NY, USA: ACM, 2010, pages 35–44 (cited on page 134).

[Fu2001] Zhi Fu, S. Wu, He Huang, Kung Loh, Fengmin Gong, Ilia Baldine & Chong Xu. "IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution". In: Lecture Notes in Computer Science 1995 (2001). Edited by Morris Sloman, Emil Lupu & Jorge Lobo, pages 39–56 (cited on pages 12, 14, 17, 147).

[Gouda2005] M.G. Gouda & A.X. Liu. "A model of stateful firewalls and its properties". In: *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. June 2005, pages 128–137 (cited on pages 12, 147).

[Guelev2004] Dimitar P. Guelev, Mark Ryan & Pierre Yves Schobbens. "Model-Checking Access Control Policies". In: *Information Security*. Edited by Kan Zhang & Yuliang Zheng. Volume 3225. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pages 219–230 (cited on pages 12, 148).

[Gutierrez2003] Claudio Gutiérrez, Carlos A. Hurtado & Alberto O. Mendelzon. "Formal aspects of querying RDF databases." In: *SWDB*. Edited by Isabel F. Cruz, Vipul Kashyap, Stefan Decker & Rainer Eckstein. 2003, pages 293–307 (cited on page 74).

[Habib2009] L. Habib, M. Jaume & Charles Morisset. "Formal definition and comparison of access control models". In: *Journal of Information Assurance and Security* 4 (2009), pages 372–378 (cited on pages 16, 134).

[Hall2002] Thomas W. Hall, James E. Hunton & Bethane Jo Pierce. "Sampling Practices of Auditors in Public Accounting, Industry, and Government". In: *Accounting Horizons* 16.2 (June 2002), pages 125–136 (cited on page 11).

[Halpern2008] Joseph Y. Halpern & Vicky Weissman. "Using First-Order Logic to Reason about Policies". In: *ACM Trans. Inf. Syst. Secur.* 11.4 (2008), pages 1–41 (cited on page 136).

[Hamed2006] H. Hamed & E. Al-Shaer. "Taxonomy of conflicts in network security policies". In: *Communications Magazine, IEEE* 44.3 (2006), pages 134–141 (cited on pages 13, 147).

[Han2012] Weili Han & Chang Lei. "A survey on policy languages in network and security management". In: *Computer Networks* 56.1 (2012), pages 477–489 (cited on pages 13, 14).

[ITIL2007] ITIL. *Glossary of Terms, Definitions and Acronyms*. Technical report. May 2007. URL: `http://www.best-management-practice.com/gempdf/itil_glossary_v3_1_24.pdf` (visited on 02/06/2014) (cited on page 5).

[Johnson2011] Arnold Johnson, Kelley Dempsey, Ron Ross, Sarbari Gupta & Dennis Bailey. *Guide for Security-Focused Configuration Management of Information Systems*. NIST Special Publication SP800-128. NIST, Aug. 2011 (cited on page 6).

[Juniper2008] Juniper Networks. *What's Behind Network Downtime?* Technical report. 2008 (cited on page 9).

[Kark2006] K. Kark, L. M. Orlov & S. Bright. *How To Manage Your Information Security Policy Framework*. Technical report. Forrester Research, 2006 (cited on page 9).

[Karvounarakis2012] Grigoris Karvounarakis & Todd J. Green. "Semiring-annotated data: queries and provenance?" In: *SIGMOD Rec.* 41.3 (Oct. 2012), pages 5–14 (cited on pages 19, 136, 137).

[Kassab1998] Lora L. Kassab & Steven J. Greenwald. "Towards formalizing the Java security architecture of JDK 1.2". In: *Computer Security — ESORICS 98*. Edited by Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows & Dieter Gollmann. Volume 1485. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pages 191–207 (cited on page 11).

[Kerravala2004] Zeus Kerravala. *As the Value of Enterprise Networks Escalates, So Does the Need for Configuration Management*. Technical report. The Yankee Group, Jan. 2004 (cited on pages 9, 10).

[Kolovski2007] Vladimir Kolovski, James Hendler & Bijan Parsia. "Analyzing web access control policies". In: *Proceedings of the 16th international conference on World Wide Web*. e. Banff, Alberta, Canada: ACM, 2007, pages 677–686 (cited on pages 14, 80, 96).

[Lenzerini2002] Maurizio Lenzerini. "Data integration: a theoretical perspective". In: PODS. Madison, Wisconsin, 2002, pages 233–246 (cited on page 75).

[Ligatti2009] Jay Ligatti, Lujo Bauer & David Walker. "Run-Time Enforcement of Nonsafety Policies". In: *ACM Trans. Inf. Syst. Secur.* 12.3 (Jan. 2009), 19:1–19:41 (cited on page 16).

[Liu2007] AlexX. Liu. "Change-Impact Analysis of Firewall Policies". In: *Computer Security – ESORICS 2007*. Edited by Joachim Biskup & Javier López. Volume 4734. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pages 155–170 (cited on pages 16, 142).

[Lodderstedt2002] Torsten Lodderstedt, David Basin & Jürgen Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security". English. In: *UML 2002 – The Unified Modeling Language*. Edited by Jean-Marc Jézéquel, Heinrich Hussmann & Stephen Cook. Volume 2460. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pages 426–441 (cited on page 14).

[Lupu1999] Emil C. Lupu & Morris Sloman. "Conflicts in Policy-Based Distributed Systems Management". In: *IEEE Trans. Softw. Eng*. 25.6 (Nov. 1999), pages 852–869 (cited on pages 13, 14, 103, 147).

[MacDonald2011] Neil MacDonald & Peter Firstbrook. *How to Devise a Server Protection Strategy*. Technical report. Gartner, Dec. 2011 (cited on page 10).

[Martnez2013] Salvador Martínez, Joaquin Garcia-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia & Jordi Cabot. "Model-Driven Extraction and Analysis of Network Security Policies". In: *Model-Driven Engineering Languages and Systems*. Edited by Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo & Peter Clarke. Volume 8107. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pages 52–68 (cited on page 133).

[Mayer2000] A. Mayer, A. Wool & E. Ziskind. "Fang: a firewall analysis engine". In: *Security and Privacy, 2000. Proceedings. 2000 IEEE Symposium on*. 2000, pages 177–187 (cited on page 133).

[McAfee] McAfee. *McAfee Policy Auditor*. URL: `http://www.mcafee.com/us/products/policy-auditor.aspx` (visited on 02/06/2014) (cited on page 48).

[Mellado2010] Daniel Mellado, Carlos Blanco, Luis E. Sánchez & Eduardo Fernández-Medina. "A Systematic Review of Security Requirements Engineering". In: *Comput. Stand. Interfaces* 32.4 (2010), pages 153–165 (cited on page 13).

[Moen2010] R. Moen & C. Norman. "Circling Back: Clearing up myths about the Deming cycle and seeing how it keeps evolving". In: *Quality Progress* (Nov. 2010), pages 22–28 (cited on page 4).

[Moffett1994] Jonathan D. Moffett & Morris S. Sloman. "Policy Conflict Analysis in Distributed System Management". In: *Journal of Organizational Computing* 4 (1994), pages 1–22 (cited on pages 13, 14, 132, 147).

[Molloy2009] Ian Molloy, Ninghui Li, Tiancheng Li, Ziqing Mao, Qihua Wang & Jorge Lobo. "Evaluating Role Mining Algorithms". In: *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*. SACMAT '09. New York, NY, USA: ACM, 2009, pages 95–104 (cited on page 134).

[Montanari2011] Mirko Montanari, Ellick Chan, Kevin Larson, Wucherl Yoo & RoyH. Campbell. "Distributed Security Policy Conformance". In: *Future Challenges in Security and Privacy for Academia and Industry*. Edited by Jan Camenisch, Simone Fischer-Hübner, Yuko Murayama, Armand Portmann & Carlos Rieder. Volume 354. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2011, pages 210–222 (cited on page 73).

[Monzillo2013] Ron Monzillo. *Java Authorization Contract for Containers, Version 1.5*. Specification JSR-115. Oracle America, Inc, May 2013 (cited on pages 95, 143).

[Naumovich2004] Gleb Naumovich & Paolina Centonze. "Static analysis of role-based access control in J2EE applications". In: *SIGSOFT Softw. Eng. Notes* 29 (5 Sept. 2004), pages 1–10 (cited on page 97).

[Nelson2010] T. Nelson, C. Barrat, D. J. Dougherty, K. Fisler & Krishnamurthi S. "The Margrave Tool for Firewall Analysys". In: *Proceedings of the 24th international conference on Large installation system administration, LISA '10*. Berkeley, CA, USA: USENIX Association, 2010, pages 1–8 (cited on page 133).

[Nessus] Tenable. *Nessus Vulnerability Scanner*. URL: `http://www.tenable.com/products/nessus` (visited on 02/06/2014) (cited on pages 15, 73).

[Ni2009] Qun Ni, Elisa Bertino & Jorge Lobo. "D-algebra for composing access control policy decisions". In: *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. Sydney, Australia: ACM, 2009, pages 298–309 (cited on pages 11, 14, 80, 96, 110, 133).

[NIST2009] NIST. *Security Content Automation Protocols*. 2009. URL: `http://scap.nist.gov` (visited on 02/06/2014) (cited on pages 14, 15, 39, 45).

[NVD] NIST. *National Vulnerability Databases*. URL: `http://nvd.nist.gov` (visited on 02/06/2014) (cited on page 15).

[OASIS2003] T.M. Simon Godik. *eXtensible Access Control Markup Language (XACML) Version1.0*. Specification. OASIS, Feb. 2003 (cited on pages 14, 96).

[OGC2007] Office of Government Commerce. *The Introduction to the ITIL Service Lifecycle Book*. The Stationery Office, Jan. 2007 (cited on page 5).

[OpenSCAP] *OpenSCAP*. URL: http://www.open-scap.org (visited on 04/23/2014) (cited on page 48).

[OpenVAS] *OpenVAS vulnerability scanner*. URL: http://www.openvas.org (visited on 02/06/2014) (cited on pages 48, 73).

[Oppenheimer2003a] David Oppenheimer. "The importance of understanding distributed system configuration". In: *System Administrators are Users, Too: Designing Workspaces for Managing Internet-Scale Systems CHI 2003 (Conference on Human Factors in Computing Systems) workshop*. Apr. 2003 (cited on page 9).

[Oppenheimer2003b] David Oppenheimer, Archana Ganapathi & David A. Patterson. "Why Do Internet Services Fail, and What Can Be Done About It?" In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS'03. Seattle, WA: USENIX Association, 2003 (cited on pages 8, 10).

[Oracle2014] Oracle. *OpenSSL Security Bug – Heartbleed / CVE-2014-0160*. 2014. URL: http://www.oracle.com/technetwork/topics/security/opensslheartbleedcve-2014-0160-2188454.html (visited on 04/23/2014) (cited on page 48).

[Ou2005] Xinming Ou, Sudhakar Govindavajhala & Andrew W. Appel. "MulVAL: a logic-based network security analyzer". In: *USENIX Security Symposium*. Baltimore, MD, 2005 (cited on page 73).

[Ovaldi] MITRE. *Ovaldi, the OVAL interpreter reference implementation*. URL: http://oval.mitre.org/language/interpreter.html (visited on 02/06/2014) (cited on page 48).

[OWASP2007] OWASP. *Securing Tomcat*. 2007. URL: https://www.owasp.org/index.php/Securing_tomcat (visited on 02/06/2014) (cited on page 41).

[OWASP2010] OWASP. *Top 10 Most Critical Web Application Security Risks*. Technical report. 2010. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2010 (visited on 02/06/2014) (cited on pages 10, 79).

[OWASP2013] OWASP. *Top 10 Most Critical Web Application Security Risks*. Technical report. 2013. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2013 (visited on 02/06/2014) (cited on pages 10, 79).

[Parmelee2011] M. C. Parmelee, H. Booth, D. Waltermire & K. Scarfone. *Common Platform Enumeration: Name Matching Specification Version 2.3*. Specification. NIST, 2011 (cited on page 47).

[Polyakov2011] Alexander Polyakov. *A crushing blow at the heart of SAP J2EE Engine*. White Paper. pages 27–29. ERPScan, Mar. 2011. URL: http://erpscan.com/wp-content/uploads/2011/08/A-crushing-blow-at-the-heart-SAP-J2EE-engine_whitepaper.pdf (visited on 04/23/2014) (cited on pages 80, 82).

[Ponemon2013a] Ponemon Institute. *2013 Cost of Data Breach Study*. Technical report. May 2013 (cited on page 10).

[Ponemon2013b] Ponemon Institute. *2013 Cost of Cyber Crime Study*. Technical report. Oct. 2013 (cited on page 10).

[Ponta2012] Serena Ponta, Madonna Mbatshi, Baptiste Cazagou, Lan Xu, Arnaud De la Bretesche, Henrik Plate & Matteo Casalino. *D4.5 – FINAL VERSION OF A CONFIGURATION VALIDATION LANGUAGE*. Deliverable. FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu), Apr. 2012. URL: http://posecco.eu/fileadmin/POSECCO/user_upload/deliverables/D4.5_v1.0.pdf (visited on 02/06/2014) (cited on page 76).

[Posecco2011] Various Authors. *Policy and Security Configuration Management*. Description of Work. Seventh Framework Programme, ICT-2009.1.4, 2011 (cited on pages 19, 20).

[Preda2010] Stere Preda, Nora Cuppens-Boulahia, Frédéric Cuppens, Joaquin Garcia-Alfaro & Laurent Toutain. "Model-Driven Security Policy Deployment: Property Oriented Approach". In: *Engineering Secure Software and Systems*. Edited by Fabio Massacci, Dan Wallach & Nicola Zannone. Volume 5965. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pages 123–139 (cited on pages 14, 134).

[Preda2011] Stere Preda, Frédéric Cuppens, Nora Cuppens-Boulahia, Joaquin Garcia-Alfaro & Laurent Toutain. "Dynamic deployment of context-aware access control policies for constrained security devices". In: *Journal of Systems and Software* 84.7 (2011), pages 1144–1159 (cited on page 134).

[Ramli2011] Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson *&* Flemming Nielson. "The Logic of XACML". In: *FACS*. Edited by Farhad Arbab *&* Peter Csaba Ölveczky. Volume 7253. Lecture Notes in Computer Science. Springer, 2011, pages 205–222 (cited on pages 11, 14, 80, 96, 98, 105, 110, 133).

[Revesz1995] Peter Z. Revesz. "Constraint Databases: A Survey". In: *Selected Papers from Semantics in Databases Workshop, Prague, Czech Republic*. Edited by Leonid Libkin *&* Bernhard Thalheim. Volume 1358. Lecture Notes in Computer Science. Springer-Verlag, 1995, pages 209–246 (cited on pages 19, 113).

[Samak2009] Taghrid Samak, Adel El-Atawy *&* Ehab Al-Shaer. "Towards Network Security Policy Generation for Configuration Analysis and Testing". In: *Proceedings of the 2Nd ACM Workshop on Assurable and Usable Security Configuration*. SafeConfig '09. Chicago, Illinois, USA: ACM, 2009, pages 45–52 (cited on page 124).

[SANS2010] SANS. *Seven Security (Mis)Configurations in Java web.xml Files*. 2010. URL: `http://software-security.sans.org/blog/2010/08/11/security-misconfigurations-java-webxml-files` (visited on 02/06/2014) (cited on pages 41, 42).

[SANS2013] SANS. *Critical Controls for Effective Cyber Defense*. Technical report Version 4.1. Mar. 2013 (cited on page 10).

[Satoh2008] Fumiko Satoh *&* Takehiro Tokuda. "Security Policy Composition for Composite Services". In: *Proceedings of the 2008 Eighth International Conference on Web Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pages 86–97 (cited on pages 13, 14).

[Savnik2000] Iztok Savnik *&* Peter A. Flach. "Discovery of multivalued dependencies from relations". In: *Intell. Data Anal.* 4.3,4 (Sept. 2000), pages 195–211 (cited on page 136).

[Schneider2000] Fred B. Schneider. "Enforceable security policies". In: *ACM Trans. Inf. Syst. Secur.* 3 (1 Feb. 2000), pages 30–50 (cited on page 16).

[Sloman2002] M. Sloman *&* E. Lupu. "Security and management policy specification". In: *Network, IEEE* 16.2 (2002), pages 10–19 (cited on pages 12, 13, 18, 103).

[Sun2008] Lianshan Sun, Gang Huang & Hong Mei. "Validating Access Control Configurations in J2EE Applications". In: *Proceedings of the 11th International Symposium on Component-Based Software Engineering*. CBSE'08. Karlsruhe, Germany: Springer-Verlag, 2008, pages 64–79 (cited on page 97).

[Tanenbaum2002] Andrew S Tanenbaum & Maarten Van Steen. *Distributed systems*. Volume 2. Prentice Hall, 2002 (cited on page 3).

[Thomas2004] Roshan K. Thomas & Ravi Sandhu. "Models, Protocols, and Architectures for Secure Pervasive Computing: Challenges and Research Directions". In: *Pervasive Computing and Communications Workshops, IEEE International Conference on* (2004), page 164 (cited on page 12).

[Tongaonkar2007] Alok Tongaonkar, Niranjan Inamdar & R. Sekar. "Inferring Higher Level Policies from Firewall Rules." In: *Proceedings of the 21st international conference on Large installation system administration, LISA '07*. Berkeley, CA, USA: USENIX Association, 2007, pages 17–26 (cited on page 133).

[Tripunitara2007] Mahesh V. Tripunitara & Ninghui Li. "A theory for comparing the expressive power of access control models". In: *J. Comput. Secur.* 15.2 (2007), pages 231–272 (cited on pages 16, 134).

[Tudor2013] Jan Tudor. *Web Application Vulnerability Statistics 2013*. Technical report. Context Information Security, June 2013 (cited on page 10).

[Ullman1997] Jeffrey D. Ullman. "Information integration using logical views". In: Springer-Verlag, 1997 (cited on page 75).

[Uszok2003] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni & J. Lott. "KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement". In: *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*. June 2003, pages 93–96 (cited on pages 13, 14, 147).

[Vaidya2007] Jaideep Vaidya, Vijayalakshmi Atluri & Qi Guo. "The Role Mining Problem: Finding a Minimal Descriptive Set of Roles". In: *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*. SACMAT '07. New York, NY, USA: ACM, 2007, pages 175–184 (cited on page 134).

[Verizon2009] Verizon. *2009 Data Breach Investigations Report*. Technical report. 2009. URL: http://www.verizonenterprise.com/resources/security/reports/2009_databreach_rp.pdf (visited on 02/06/2014) (cited on page 10).

[Verizon2010] Verizon. *2010 Data Breach Investigations Report*. Technical report. 2010. URL: http://www.verizonenterprise.com/resources/reports/rp_2010-data-breach-report_en_xg.pdf (visited on 02/06/2014) (cited on page 10).

[Vimercati2007] S. Capitani di Vimercati, S. Foresti, S. Jajodia *&* P. Samarati. "Access Control Policies and Languages in Open Environments". English. In: *Secure Data Management in Decentralized Systems*. Edited by Ting Yu *&* Sushil Jajodia. Volume 33. Advances in Information Security. Springer US, 2007, pages 21–58 (cited on page 13).

[Waltermire2011] D. Waltermire, S. Quinn *&* K. Scarfone. *The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP Version 1.1*. Technical report. NIST, 2011 (cited on page 45).

[Westerinen2001] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry *&* S. Waldbusser. *Terminology for Policy-Based Management*. Request for Comments RFC3198. Internet Engineering Task Force, Nov. 2001. URL: http://www.ietf.org/rfc/rfc3198.txt (visited on 02/06/2014) (cited on pages 6, 7).

[Wijesekera2003] Duminda Wijesekera *&* Sushil Jajodia. "A propositional policy algebra for access control". In: *ACM Trans. Inf. Syst. Secur.* 6 (2 May 2003), pages 286–325 (cited on pages 13, 103).

[Wijsen2005] Jef Wijsen. "Database repairing using updates". In: *ACM Trans. Database Syst.* 30.3 (Sept. 2005), pages 722–768 (cited on page 136).

[Wool2010] Avishai Wool. "Trends in Firewall Configuration Errors: Measuring the Holes in Swiss Cheese". In: *IEEE Internet Computing* 14.4 (2010), pages 58–65 (cited on page 11).

[Wullems2004] C. Wullems, M. Looi *&* A. Clark. "Towards context-aware security: an authorization architecture for intranet environments". In: *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*. 2004, pages 132–137 (cited on page 12).

[Yin2011] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram & Shankar Pasupathy. "An Empirical Study on Configuration Errors in Commercial and Open Source Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pages 159–172 (cited on page 9).

[Yuan2005] Eric Yuan & Jin Tong. "Attributed Based Access Control (ABAC) for Web Services". In: *Proceedings of the IEEE International Conference on Web Services*. ICWS '05. Washington, DC, USA: IEEE Computer Society, 2005, pages 561–569 (cited on page 96).

[Zhao2011] Hang Zhao, Jorge Lobo, Arnab Roy & Steven M Bellovin. "Policy Refinement of Network Services for MANETs". In: *The 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*. Dublin, Ireland, May 2011 (cited on pages 14, 133).

[Ziring2008] N. Ziring & S. Quinn. *Specification for the Extensible Configuration Checklist Description Format (XCCDF) Version 1.1.4*. Specification. NIST, 2008. URL: `http://csrc.nist.gov/publications/nistir/ir7275r3/NISTIR-7275r3.pdf` (visited on 04/23/2014) (cited on pages 45, 69).