Architectures et mécanismes de sécurité pour l'auto-protection des systèmes pervasifs

Ruan He

To cite this version:

Ruan He. Architectures et mécanismes de sécurité pour l'auto-protection des systèmes pervasifs. Informatique ubiquitaire. Télécom ParisTech, 2010. Français. NNT : . pastel-00579773

HAL Id: pastel-00579773
https://pastel.hal.science/pastel-00579773

Submitted on 24 Mar 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Doctorale
d'Informatique,
Télécommunications
et Électronique de Paris

Thèse

présentée pour obtenir le grade de doctoral de

TELECOM ParisTech

l'Ecole Nationale Supérieure des Télécommunications

Ruan HE

Architectures et mécanismes de sécurité pour l'auto-protection de systèmes pervasifs

Soutenance le 30 Novembre 2010 devant le jury composé de

Pr. Frédéric Cuppens	**Rapporteurs**
Pr. Dominique Gaïti	**Rapporteurs**
Dr. Thierry Coupaye	**Examinateurs**
Pr. Isabelle Demeure	**Examinateurs**
Pr. Julie A.McCann	**Examinateurs**
Dr. Marc Lacoste	**Encadrant Industriel**
Dr. Jean Leneutre	**Directeur de thèse**

École Doctorale
d'Informatique,
Télécommunications
et Électronique de Paris

Dissertation

Submitted in fulfillment of the requirements for the

Ph.D. degree in Computer Science of

TELECOM ParisTech

By

Ruan HE

Security Architecture and Mechanisms for Self-protection of Pervasive Systems

Defended on November 30, 2010

Dissertation Committee:

Pr. Frédéric Cuppens	Reviewer
Pr. Dominique Gaïti	Reviewer
Dr. Thierry Coupaye	Examiner
Pr. Isabelle Demeure	Examiner
Pr. Julie A.McCann	Examiner
Dr. Marc Lacoste	Thesis Advisor
Dr. Jean Leneutre	Thesis Advisor

Acknowledgement

It is hard to express in words how thankful I am to my advisors Marc Lacoste and Jean Leneutre, whose guidance and supports have made this thesis successfully realized. They have not only been my advisors but also friends who have taught me so many things beyond research. I would also like to especially thank Jacques Pulou whose suggestions have helped in improving my thesis. He too has been a genuine source of inspiration and jokes relating to research and beyond. I would also like to thank all members of *MAPS/SHINE/MADE* and *MAPS/STT/NDS* departments of *Orange Labs* for their support and friendship.

Secondly, I would like to thank the two reviewers Pr.Frédéric Cuppens, Pr.Dominique Gaïti, and the three examiners Dr.Thierry Coupaye, Pr.Isabelle Demeure, Pr.Julie A.McCann of my dissertation committee for all their hard work and their important and useful comments.

Finally I would like to thank my family, extraordinary parents and my lively wife for their relentless support and encouragement in all aspects of life, and making me a person who I am today.

Résumé

0.1 Introduction

0.1.1 Emergence de l'informatique pervasive

La notion de système pervasif a été introduite au début des années 90 par Mark Weiser dans [147] qui propose une intégration de l'informatique embarquée et de la communication sans fil à nos activités quotidiennes. Un système pervasif est un environnement dans lequel l'homme peut interagir avec des équipements qui ont des capacités de calcul et de communication. Le traitement d'informations par ces équipements peut guider et améliorer nos activités quotidiennes.

Les progrès dans les technologies liées aux systèmes embarqués et à la communication sans fil ont permis à ces systèmes de devenir une réalité. La miniaturisation des équipements informatiques et les progrès dans l'intégration de la micro-électronique pour les systèmes embarqués, ont transformé les interactions homme-machine, les rendent omniprésentes et diffuses. Certaines innovations récentes dans la conception des puces et des nanotechnologies permetent de réduire la consommation énergétique des systèmes embarqués. D'autre part, l'amélioration des communications sans fil permet de connecter ces appareils embarqués au réseau de manière plus efficace. Ces deux avancées majeures ont permis l'émergence de systèmes pervasifs concrétisant la vision de Mark Weiser.

Dans cette thèse, un système pervasif correspond à un réseau (composé de terminaux) qui est :

- distribué et décentralisé : les terminaux sont physiquement distribués dans une région et sont reliésentre eux par des connexions sans fil. Quelque soit leur localisation physique, la communication sans fil permet de les connecter entre eux. Cette architecture distribuée nécessite une coordination décentralisée des terminaux.

- dynamique et ouvert : les terminaux, en particulier les terminaux mobiles, peuvent rejoindre et quitter un réseau pervasif à tout moment. Cette ouverture rend l'architecture du système dynamique : c'est-à-dire que la topologie du système évolue au cours du temps pendant l'exécution. Ce caractère dynamique de l'architecture nécessite une modélisation du système suffisamment flexible pour prendre en compte l'évolution de topologie du système.

- complexe et de grande taille : l'échelle de ces systèmes peut atteindre des centaines voire des milliers de terminaux. La mise en place d'une fonction de coordination, gérant non seulement ces terminaux, mais également leurs connexions, est une tâche d'un niveau de complexité élevé. Au lieu de solutions traditionnellement centralisées qui gèrent un petit nombre de terminaux, les systèmes pervasifs à grande échelle doivent appliquer une solution plus efficace.

Plusieurs types de menaces sont identifiés dans cette thèse. Il existe des menaces locales qui insèrent des logiciels malicieux en compromettant la confidentialité et l'intégrité du

système local de chaque terminal. Les menaces au niveau de réseau peuvent provoquer des attaques du type “Denial of Service ” qui compromettent la disponibilité d’une partie ou de la totalité des terminaux du réseau. De plus, des menaces hybrides peuvent viser en même temps les terminaux et le réseau.

0.1.1.1 Système auto-protégeable

Les systèmes pervasifs sont sujets à un ensemble de menaces évoluant très rapidement et doivent faire face à des exigences de sécurité hétérogènes, ce qui nécessite des mécanismes de protection plus efficaces et flexibles. La gestion manuelle de la protection devient beaucoup trop complexe, les administrateurs de sécurité devant configurer chaque terminal. Ceci apparaît impossible pour les systèmes pervasifs à grande échelle. L’approche autonome de gestion de sécurité [47] automatise l’administration des manipulations répétitives de bas niveau et représente une solution prometteuse pour résoudre les problèmes précédents. Un système auto-protégeable réalise des opérations de protection sans ou avec peu d’intervention d’administrateurs ou d’utilisateurs.

Dans cette thèse, nous nous intéressons à l’auto-protection des systèmes pervasifs, se focalisant sur la faisabilité de la réalisation d’un canevas logiciel pour l’auto-protection. L’objectif de ce canevas logiciel est de permettre aux terminaux et équipements de réseau de réagir contre les menaces, les attaques de tous niveaux (i.e. du niveau du système d’exploitation jusqu’au niveau applicatif). Malheureusement, la diversité des menaces et le caractère dynamique des systèmes pervasifs rendent l’auto-protection très difficile. En se basant sur l’approche informatique autonome, nous appliquons les théories du contrôle aux mécanismes de protection classiques pour étendre le canevas de protection vers un canevas logiciel dans lequel les menaces peuvent être éliminées de manière autonome.

Contexte : l’approche informatique autonome (autonomic computing) IBM a tout d’abord présenté le principe de l’informatique autonome comme un système auto-géré, libérant les administrateurs de la gestion des tâches de bas niveau, en proposant d’utiliser des boucles de contrôle autonomes pour améliorer le comportement de systèmes [90]. Quatre propriétés sont définies : l’auto-configuration, l’auto-optimisation, l’auto-protection, et l’auto-guérison. L’informatique autonome consiste en un ensemble de patrons logiciels, d’architecture d’implantation, et de processus pour réduire la complexité d’administration d’un système. Elle répond à certaines problématiques des systèmes informatiques tel que :

- La complexité croissante : les systèmes informatiques, les applications, et les environnements d’exécution sont en forte croissance. Les mécanismes d’administration ne sont pas toujours effectifs face à cette croissance. Un système informatique à grande échelle composé de centaines ou de milliers d’applications nécessite un grand nombre de configurations pour chaque tâche d’administration, ce qui est hors du contrôle humain. La maintenance de ce type de système informatique devient un verrou technique pour le passage à échelle. L’informatique autonome fournit des outils pour administrer les systèmes d’une façon autonome en permettant aux ad-

ministrateurs de se concentrer sur les stratégies de haut niveau plutôt que sur des mécanismes de bas niveau.

- L'évolution continue des systèmes logiciels : les systèmes logiciels sont en évolution constante, ne peuvent jamais être complètement spécifiés, et doivent être soumis à une adaptation continue [141]. Cette évolution peut porter sur deux parties : sur les exigences de conception, ce qui conduit à une mise à jour du système, ou sur l'environnement d'exécution, ce qui se traduit par une adaptation du système. Le premier type d'évolution demande une infrastructure extensible et flexible, et illustre le fait que l'environnement d'exécution n'est pas connu a priori au moment de la conception. La deuxième évolution montre que l'environnement applicatif ne peut pas rester constant [87]. L'informatique autonome permet de prendre en compte ces deux évolutions en reconfigurant dynamiquement les fonctionnalités, l'architecture et les politiques d'administration d'un système informatique.

- La croissance du coût d'administration : le coût d'administration est devenu un facteur critique pour les systèmes informatiques complexes. De plus, les configurations manuelles peuvent également induire des dysfonctionnements et des pertes de données. L'informatique autonome est considérée comme un moyen de réduire ce coût de maintenance en permettant des reconfigurations dynamiques et des optimisations guidées par des rétroactions sur le comportement des systèmes.

Défis de l'auto-protection L'auto-protection est l'une des quatre propriétés principales de l'informatique autonome [90]. Le principe de l'auto-protection est d'intégrer des boucles de rétroaction dans des canevas logiciel de mécanismes de protection existants. Dans cette thèse, nous définissons un canevas logiciel d'auto-protection dans lequel les applications peuvent être mises en oeuvre et bien protégées. Pour les systèmes informatiques basés sur ce canevas, tous les modules logiciels, du niveau de matériel au niveau applicatif, sont protégés. La construction d'un tel canevas logiciel suppose de relever les défis suivants :

- La sécurité de bout en bout : du fait de leur ouverture de nouvelles vulnérabilités sont apparues dans les systèmes pervasifs par rapport aux systèmes informatiques classiques. Les comportements malveillants peuvent compromettre les systèmes aussi bien du côté du terminal que du côté réseau. Par conséquent, un canevas logiciel de protection efficace devrait couvrir tous ces aspects et se défendre contre toutes les menaces potentielles.

- La flexibilité du contrôle des applications : un canevas logiciel d'auto-protection peut se concevoir comme une couche séparée coopérant avec un système à superviser et à protéger. Il protège les applications en appliquant des mécanismes de réaction complémentaires. Par conséquent, ce type de canevas logiciel nécessite de pouvoir exercer un contrôle pendant l'exécution des applications. D'où le besoin d'une plateforme flexible permettant la reconfiguration dynamique du système à protéger.

- Le compromis entre l'efficacité des mécanismes de protection et leur coût : le coût des mécanismes de protection reste important, un système strictement protégé nécessite souvent des contrôles pour chaque opération. Ce coût comprend deux parties : un coût de calcul et un coût de communication. Le premier correspond à l'allocation de ressources de calcul. Le deuxième est le trafic supplémentaire induit pour la coordination des mécanismes de protection. Un canevas logiciel de protection efficace devrait donc minimiser ces deux types de coûts.

- La faisabilité d'implantation de la sécurité autonome : dans plusieurs domaines de recherche tels que les systèmes sensibles au contexte ou le cloud computing, l'ensemble des systèmes sont gérés d'une manière autonome, sans aucune intervention de l'utilisateur. Les mécanismes d'un système pervasif ne pourront être configurés manuellement et devront donc de protection devront donc être auto-gérés.

0.1.2 Approche et contributions

0.1.2.1 Approche

Malgré les progrès récents sur les systèmes informatiques autonomes, l'auto-protection reste un sujet relativement peu traitée. Nous considérons que l'on est encore loin d'atteindre un système auto-protégé, tant du point de vue de la conception, que de la mise en oeuvre, du fait des manques suivants :

1. une architecture de référence générique pour l'organisation des composants constitutifs du canevas;

2. une approche fondamentale, mais extensible pour la mise en oeuvre de la sécurité;

3. un patron de conception pour la sécurité des systèmes autonomes;

4. un paradigme efficace pour supporter l'exécution et le contrôle des applications.

Nous proposons les approches suivantes pour pallier ces manques :

- une architecture de protection de bout en bout : une telle architecture offre une structure générique qui peut être utilisé pour différents systèmes. On qualifie un canevas logiciel de protection de bout en bout s'il peut protéger à la fois ses terminaux et ses équipements de réseau. Par exemple, un système d'isolation local ne pourra pas lutter contre les attaques au niveau du réseau parce qu'il ne possède pas d'une vue globale sur l'ensemble du réseau.

- Une approche orientée contrôle d'accès : le contrôle d'accès est considéré comme une fonction de sécurité fondamentale pour assurer la sécurité dans notre canevas logiciel d'auto-protection. Le mécanisme associé à cette fonction est généralement intégré comme un module de sécurité dans le noyau des systèmes d'exploitation afin de limiter l'accès aux ressources par des validations. En outre, d'autres fonctionnalités de sécurité telles que la gestion de l'anonymat ("privacy ") peuvent également dépendre

du contrôle d'accès. Le contrôle d'accès implique généralement une politique et des mécanismes. La politique spécifie les différentes règles d'accès, les mécanismes assurent leur validation.

- Une implantation du patron de conception : le patron autonome peut être réalisé au niveau du matériel (par exemple, l'auto-diagnostic du matériel [12]), du système d'exploitation (par exemple, l'auto-configuration ou gestion de la sécurité [83]), du middleware (par exemple, l'infrastructure d'auto-guérison), ou de l'application (par exemple, l'auto-optimisation [102], ou l'optimisation de la qualité de service). Notre canevas logiciel traite la sécurité à deux niveaux, en utilisant des boucles de rétroaction autonome distinctes pour l'auto-protection du réseau et du terminal: (1) une boucle de contrôle du terminal met à jour les règles de contrôle d'accès locaux; (2) une boucle de contrôle du réseau gère la coordination d'un ensemble de terminaux dans le réseau.

- Un mécanisme de contrôle en temps réel : une approche pour faire face aux évolutions continuelles est de migrer certaines activités de la phase de conception vers l'exécution. Avec cette philosophie, un système logiciel est constitué d'une infrastructure statique définie au moment de la conception et d'un ensemble de composants dynamiques qui peuvent être mis à jour ou remplacés lors de l'exécution. L'approche à base de politiques a démontré sa flexibilité et sa généricité pour pouvoir supporter cette philosophie [138]: les fonctionnalités du système sont régies par un ensemble de politiques. Lors du changement de contexte, une autre politique peut être sélectionnée et activée afin que le système puisse mieux s'adapter à son nouvel environnement.

Contributions principales Les éléments précédents identifient les principes de conception de base pour construire un canevas logiciel d'auto-protection. Différentes technologies peuvent être choisies pour réaliser ces principes. Les contributions principales de cette thèse décrivent des mécanismes développés et mis en oeuvre pour réaliser notre canevas logiciel d'auto-protection. Il s'agit des éléments suivants :

- Une architecture à trois couches pour l'auto-protection : un espace d'exécution fournit un environnement d'exécution pour des applications; un plan de contrôle supervise l'espace d'exécution ; et un plan autonome guide le plan de contrôle en prenant en compte l'état du système, l'évolution des risques, la stratégie de sécurité définie par l'administrateur, et les préférences de l'utilisateur.

- Une approche du contrôle d'accès à base d'attributs: l'approche proposée (appelée G-ABAC) exprime les politiques d'autorisation en se basant sur des attributs. Cette approche apporte à la fois une neutralité vis-à-vis du modèle de contrôle d'accès, et une flexibilité permettant des manipulations élémentaires sur ces politiques.

- Un canevas logiciel à base de politiques pour réaliser la gestion autonome de la sécurité : l'approche à base de politiques a montré ses avantages pour l'administration

des systèmes complexes et dynamiques [105]. Un canevas logiciel autonome de politiques de sécurité (ASPF) fournit une solution cohérente et décentralisée pour administrer les politiques d'autorisation pour les systèmes pervasifs à grande échelle. L'intégration des patrons autonomes améliore également la souplesse et la facilité d'adaptation au contexte.

- Un noyau de sécurité embarqué pour l'application des politiques de contrôle d'accès : les politiques d'autorisation définies précédemment sont appliquées par une architecture d'autorisation au niveau du système d'exploitation. Ce noyau appelé VSK contrôle l'accès aux ressources d'une manière dynamique afin de réduire le surcoût des mécanismes d'autorisation. Ce mécanisme permet également de supporter différents types de politiques d'autorisation.

- Un langage dédié (Domain-Specific Language ou DSL) pour la spécification de politiques d'adaptation : toutes les adaptations de notre canevas logiciel d'auto-protection de bout en bout sont contrôlées par des stratégies de haut niveau appelées politiques d'adaptation. Un DSL tenant compte de nombreux facteurs pour les décisions d'adaptation. est défini pour spécifier ces politiques.

0.1.3 Organisation de la thèse

Ce document de thèse commence par une présentation détaillée du contexte applicatif dans la Partie I. Les principaux choix de conception pour construire un canevas logiciel autonome qui ont été effectués, sont décrits dans le chapitre 2. Ce chapitre explore également quelques solutions existantes pour en déterminer les limites et les possibilités. Une modélisation du système d'adaptation prenant en compte le dynamisme et la flexibilité des systèmes pervasifs est présentée au chapitre 3. Le modèle obtenu est extensible et permet de décrire des aspects multiples tels que le niveau de risque, la qualité de service, la performance, etc. Trois scénarios de travail ainsi que les menaces et les contre-mesures correspondantes sont également présentés dans ce chapitre.

Les contributions principales de la thèse sont décrites dans la partie II. Dans le chapitre 4, nous introduisons G-ABAC, une approche pour le contrôle d'accès à base d'attributs. Bien que G-ABAC ne soit pas vraiment un modèle de contrôle d'accès formalisé, il explore un paradigme à base d'attributs pour spécifier un grand nombre de politiques d'autorisation. Il offre aussi l'avantage de pouvoir supporter des modèles d'administration différents. En se basant sur G-ABAC, nous proposons un canevas logiciel d'administration appelé ASPF qui réalise la gestion distribuée des politiques de sécurité dans les systèmes pervasifs. D'une part, ASPF peut être vu comme un canevas logiciel qui réalise la gestion de politiques G-ABAC de contrôle d'accès. D'autre part, ASPF étend XACML [77], l'architecture traditionnelle d'autorisation, en intégrant des fonctionnalités autonomes pour en simplifier la gestion. L'application des politiques G-ABAC du côté de terminal est réalisée par VSK, qui est une architecture d'autorisation souple et légère au niveau du système d'exploitation. VSK permet d'appliquer les politiques de contrôle d'accès spécifiées par G-ABAC. Il est possiblle avec VSK de configurer dynamiquement

les politiques de contrôle d'accès. Par ailleurs, il est également possible lors de la validation de demandes d'accès de combiner les décisions de plusieurs politiques de contrôle d'accès différentes. Toutes ces caractéristiques font VSK une plate-forme côté terminal relativement efficace et pouvant servir de base pour le canevas logiciel d'auto-protection. ASPF et VSK sont les contributions principales de cette thèse, et seront respectivement détaillées aux chapitres 5 et 6. Un langage dédié (DSL) pour la spécification des politiques d'adaptation est décrit dans le chapitre 7. Ce DSL permet d'exprimer des compromis entre la sécurité et d'autres aspects comme l'efficacité énergétique au cours de la phase de prise de décision. Un mécanisme de traduction raffine ce DSL en un mécanisme temps réel. L'intégration de ces politiques d'adaptation dans des canevas logiciel existants est également illustrée.

La validation du canevas logiciel d'auto-protection est présentée dans la partie III. Basé sur les résultats du projet E2R [95], la mise en oeuvre du côté du terminal est réalisée et évaluées au Chapitre 8 (dans le cadre des projet FUI Mind et Pronto). Les évaluations [81, 83] montrent l'efficacité et la flexibilité de l'architecture d'autorisation au niveau du système d'exploitation. Le chapitre 9 décrit ASPF et sa mise en oeuvre au niveau du réseau. Les évaluations du canevas logiciel d'auto-protection de bout en bout [82, 14] ont été réalisées au sein du projet ANR SelfXL. Ces évaluations illustrent l'efficacité des deux boucles autonomes en termes de temps de réponse et de résilience. Enfin, une autre validation du canevas dans le domaine du cloud computing est présentée dans le chapitre 10. Ces travaux font partie d'un projet en cours pour traiter la problématique de la sécurité des infrastructures de type cloud.

0.1.4 Contributions

0.1.4.1 Architecture de sécurité autonome

Avec la forte croissance de la complexité des systèmes de protection, l'administration des mécanismes de sécurité et des politiques d'autorisation devient un problème critique. L'informatique autonome [90] définit une approche prometteuse pour simplifier l'administration en intégrant des boucles de contrôle rétroactives. Un canevas d'auto-protection gère tous ses composants de sécurité sans ou avec un minimum d'intervention de l'humaine, et il peut réagir de façon autonome aux évolutions liées au changement des préférences de l'utilisateur ou de l'environnement de façon autonome.

Approche à base de politiques Un besoin de conception pour un tel système autonome est de développer un canevas logiciel d'administration efficace et simple qui fait collaborer tous les composants en suivant une stratégie de protection globale. L'approche à base de politiques peut répondre à ce besoin [105]. Dans cette approche, des politiques abstraites de haut niveau guident le comportement du canevas de protection et génèrent dynamiquement des opérations de bas niveau [137]. D'autre part, le canevas logiciel de contrôle d'accès applique les politiques : tous les accès aux ressources sont contrôlés par une politique d'autorisation [35, 132]. Par rapport à l'objectif de cette thèse, afin qui est de proposer un canevas logiciel de sécurité autonome, l'approche à base de politiques est

considérée comme une base de conception de conception qui faciliteant le contrôle d'accès.

Une politique de contrôle d'accès d'un tel canevas logiciel est un ensemble de règles qui définissent des conditions potentielles pour accéder aux ressources. La séparation entre la politique et ses mécanismes permet une évolution dynamique de la stratégie d'administration sans modifier l'infrastructure sous-jacente. Par conséquent, le canevas logiciel à base de politiques est capable de contrôler dynamiquement la protection des systèmes pervasifs sans modifier ses mécanismes de protection sous-jacents. Deux types des politiques sont utilisés dans notre canevas logiciel: les politiques d'autorisation qui sont des règles prédictives pour contrôler l'accès et les politiques d'adaptation qui modifient l'exécution du système lors de changement de contexte.

Fonctionnalité vs. maturité autonome Du point de vue des fonctionnalités autonomes, IBM échelonne le degré de maturité de l'informatique autonome en cinq niveaux : le niveau basique, le niveau gérable, le niveau prédictif, le niveau adaptatif et le niveau autonome. Les différents canevas logiciels autonomes peuvent être classés selon ce degré de maturité. Chaque niveau de maturité perfectionne le niveau le précédant en offrant des fonctionnalités supplémentaires qui rendent le canevas plus autonome. Par exemple, un canevas logiciel est classifié en niveau prédictif si les fonctionnalités des deux niveaux sous-jacents (le niveau basique et le niveau gérable) sont bien établis. Afin d'atteindre le niveau autonome qui est l'un des objectifs de cette thèse, notre canevas logiciel devrait contenir toutes les fonctionnalités qui correspondent à ces cinq niveaux.

Le niveau le plus bas (le niveau basique) peut être considéré comme un environnement d'exécution des applications sans mécanisme de contrôle. Le niveau gérable prévoit des mécanismes de contrôle fondamentaux et le niveau prédictif applique quelques règles de prédiction au-delà du niveau gérable. Le niveau adaptif tient compte du contexte pour optimiser l'exécution. Enfin, le niveau autonome propose des stratégies sémantiques pour guider le comportement de l'adaptation.

Architecture à trois couches Dans le contexte des systèmes pervasifs, le canevas logiciel de sécurité autonome de bout en bout que nous proposons réalise ces cinq niveaux de maturité selon trois couches (voir la figure). L'espace d'exécution est un environnement d'exécution pour les applications qui joue le rôle du niveau basique. Le plan de contrôle regroupe à la fois le niveau gérable et le niveau prédictif qui contrôle l'espace d'exécution en appliquant des règles de prédiction. Dans notre cas, les politiques d'autorisation sont des règles de prédiction pour contrôler l'accès aux ressources. Au-dessus, le plan adaptatif autonome est coordonné avec les politiques d'adaptation sémantique. Cette sous-section décrit ces trois plans, et plus de détails sur la réalisation de chaque couche seront présentés dans les sous-sections suivantes.

Espace d'exécution: l'espace d'exécution est un environnement dans lequel les applications peuvent être lancées et exécutées. Tous les systèmes sans mécanismes de contrôle peuvent être considérés comme espace d'exécution. Parce que les niveaux supérieurs appliquent le contrôle, l'espace d'exécution doit fournir une représentation uniforme de toutes les applications. L'approche à base de composants (Component-Based Software Engineering ou CBSE) [40] apparaît comme une bonne solution car elle permet d'abstraire toutes les

applications et ressources sous forme de composants. Les interfaces standards sont définies et utilisées pour contrôler ces composants. D'autres avantages comme la reconfiguration dynamique sont également fournis par CBSE.

Plan de contrôle: le plan de contrôle combine le niveau gérable et le niveau prédictif du modèle de maturité autonome d'IBM. Afin de réaliser le niveau gérable, des mécanismes de supervision et de reconfiguration sur l'espace d'exécution doivent être proposés. Pour réaliser le niveau prédictif, certains types de règles de prédiction doivent être utilisés pour indiquer les réactions potentielles. Comme nous nous intéressons à un canevas logiciel d'auto-protection, les politiques d'autorisation sont appliquées comme des règles de prédiction. Une politique d'autorisation est composée de plusieurs règles qui représentent les permissions d'autorisation des sujets sur les objets avec des opérations. Lorsqu'un composant dans l'espace d'exécution tente d'accéder à un autre composant, une requête d'accès est tout d'abord capturée par le plan de contrôle. Ensuite, un module prédictif de décision dans ce plan prend une décision cette requête. Un module de reconfiguration réalise ensuite une série de réactions en fonction de la décision.

Plan Autonome: Dans un canevas logiciel de contrôle d'accès, les règles d'autorisation sont généralement préfixées et ne peuvent pas être modifiées lors de l'exécution. Cependant, le canevas nécessite une adaptation à l'évolution du contexte, où les règles de prédiction peuvent varier en fonction du contexte [100]. Par exemple, une demande d'accès pourrait être accordée si le canevas est dans un contexte relativement sûr et devrait être refusée dans le cas contraire. Le niveau adaptatif propose des fonctions pour effectuer cette mise à jour en se basant sur des informations de contexte. Enfin, pour distinguer un système de niveau autonome d'un système de niveau adaptatif, des politiques d'adaptation sémantiques guidant le comportement d'adaptation du canevas logiciel d'auto-protection doivent être fournies. En bref, le niveau adaptatif réalise l'adaptation, alors que le niveau autonome guide le comportement d'adaptation en fonction des préférences de l'utilisateur, de condition d'exécution, etc.

0.1.5 Contrôle d'accès générique à base d'attributs (G-ABAC)

Comme décrit dans la section précédente, la politique de contrôle d'accès est le centre du canevas logiciel de bout en bout. Le plan autonome sélectionne la politique d'accès la plus adéquate parmi un ensemble de politiques de contrôle d'accès potentielles. Par la suite, il personnalise et déploie la politique choisie sur l'ensemble de terminaux à travers du réseau. Le plan de contrôle applique des politiques sur mesure dans tous les terminaux. Au niveau de la spécification des politiques de contrôle d'accès, certains modèles existants définissent des modèles d'administration [126, 113] afin de contrôler ou limiter les manipulations d'administration. Bien qu'il existe des modèles de contrôle d'accès qui soient très expressifs [98, 116], peu de modèles apportent leur appui sur l'administration qui est un point critique dans le cas où l'on souhaite que des terminaux mobiles puissent être intégrés dans des réseaux différents. Une approche adoptant une neutralité vis-àvis du modèle de de contrôle d'accès [24, 79, 136] permet améliorer non seulement l'expressivité en terme de politiques spécifiées, mais aussi la flexibilité pour les modèles d'administration.

Bien que les contributions principales de cette thèse soient la mise en place et l'application

de politiques de contrôle d'accès, et non la proposition d'un nouveau modèle de contrôle d'accès, nous présentons dans cette sous-section une approche appelée "contrôle d'accès générique à base d'attributs ou G-ABAC ". G-ABAC permet de spécifier un grand nombre de politiques de contrôle d'accès existants. Cette approche facilite l'administration, la validation et la configuration des politiques de contrôle d'accès au niveau du système d'exploitation embarqué. Son support des modèles d'administration est plus générique: un grand nombre de modèles d'administration peuvent être intégrés dans le canevas logiciel ASPF qui s'applique G-ABAC.

Le chapitre 4 de la thèse donne une description détaillée du G-ABAC. Dans le contexte du système pervasif, la neutralité vis-à-vis des modèles de contrôle d'accès, la décentralisation de la validation des requêtes d'accès, l'efficacité du contrôle d'accès, et la flexibilité pour changer les modèles sont cités comme les exigences clés de conception. En appliquant l'approche à base d'attributs et en effectuant une séparation entre la spécification des politiques et l'administration des attributs, G-ABAC devient plus flexible. G-ABAC permet de reconfigurer dynamiquement et de personnaliser des politiques afin d'améliorer l'efficacité. Par conséquent, toutes les exigences sont satisfaites avec G-ABAC. Ce chapitre présente également une définition de G-ABAC, illustrant comment mettre en oeuvre plusieurs politiques d'autorisation existantes. Toutefois, l'approche d'autorisation G-ABAC n'envisage pas le déploiement pour les systèmes distribués. Un canevas logiciel autonome d'administration de politiques de sécurité (ASPF) qui déploie les politiques G-ABAC à travers du système pervasif est introduit dans le chapitre 5 de la thèse.

0.1.5.1 Canevas logiciel autonome d'administration de politiques de sécurité

Le chapitre 4 a proposé une approche de contrôle d'accès générique, G-ABAC, qui permet d'exprimer les politiques de contrôle d'accès. Cependant, il faut également préciser comment s'effectue le déploiement des politiques G-ABAC. Le chapitre 5 de la thèse propose un canevas logiciel appelé "canevas logiciel autonome d'administration de politiques de sécurité ou ASPF " réalisant non seulement le déploiement de différents politiques de contrôle d'accès dans un environnement distribué, mais incluant également l'administration autonome pour améliorer les capacités d'auto-gestion. En se basant sur ce canevas logiciel, différents modèles d'administration correspondants aux différents modèles de contrôle d'accès peuvent être mis en oeuvre.

L'approche à base de politiques a été initialement développée pour réduire la complexité d'administration des systèmes informatiques. L'utilisation des politiques d'autorisation guide la protection des systèmes gérés. Donc, l'intégration de ces deux approches améliore le déploiement et l'administration des politiques d'autorisation et garantit leur cohérence. Avec l'aide de l'informatique autonome, un canevas logiciel à base de politiques devient à la fois convivial, sans intervention humaine et sensible aux évolutions du contexte.

ASPF met en oeuvre deux boucles d'auto-protection, une au niveau de terminal et l'autre au niveau de réseau. En plus, les politiques d'autorisation s'adaptent en fonction de l'évolution du contexte de sécurité. Pour un tel canevas logiciel qui est à grande échelle et distribué, les exigences de conception tels que le support de G-ABAC, l'évolutivité de l'architecture, la cohérence des politiques distribuées, la convivialité, et la sensibilité

au contexte sont bien satisfaites grâce à l'utilisation de l'approche à base de politiques, l'architecture d'autorisation décentralisée, le patron de conception d'auto-protection et le patron de conception d'auto-configuration. En outre, la séparation des modèles de base d'ASPF par rapport aux modèles étendus et les modèles d'implémentation améliore la flexibilité et permet une réutilisation dans différents contextes. Cependant, les politiques d'autorisation administrées et déployées par ASPF devraient être appliquées par un système d'exploitation embarqué sous-jacent. L'intégration et la mise en oeuvre des politiques G-ABAC au niveau de système d'exploitation sont détaillées dans le chapitre 6. La réutilisation du canevas ASPF dans le cadre du cloud computing est décrite dans le chapitre 10.

0.1.5.2 Noyau de sécurité virtuel (VSK)

Basé sur l'approche G-ABAC (chapitre 4), le canevas logiciel ASPF (chapitre 5) administre des politiques d'autorisation à travers le système pervasif. Dans le chapitre 6 de la thèse, une nouvelle architecture d'autorisation au niveau du système d'exploitation appelée noyau de sécurité virtuel (VSK) est proposée. Elle applique les politiques d'autorisation G-ABAC au niveau du système d'exploitation. Suivant les évolutions du contexte, ASPF met à jour les politiques d'autorisation. Ces politiques mises à jour sont installées dans VSK, les futures demandes d'accès sont ensuite validées en appliquant ces nouvelles politiques d'autorisation.

VSK effectue une séparation complète entre un noyau de contrôle minimal (le plan de contrôle) et l'exécution de ressources (l'espace d'exécution). Le noyau effectue la reconfiguration efficace lors de l'exécution des ressources. Il gère également l'autorisation par un moniteur de référence qui supporte différents types de politiques. La protection devient donc cachée grâce à un mécanisme de contrôle d'accès optimisé. L'architecture de VSK est à base de composants ce qui offre une grande flexibilité à la fois au niveau des ressources pour la personnalisation et au niveau du noyau de système d'exploitation pour supporter la neutralité vis-à-vis des politiques d'autorisation.

Dans l'architecture d'administration des politiques à trois couches proposée au chapitre 3, l'espace d'exécution est séparé par rapport au plan de contrôle. Le chapitre 5 se concentre sur l'architecture du système d'exploitation permettant la réalisation du plan de contrôle du côté de terminal. VSK est donc un plan de contrôle dynamique et léger (en terme de surcoût lié à la validation des autorisations), permettant au noyau de contrôler totalement l'exécution des applications. VSK applique l'approche à base de composants qui fournit une abstraction uniforme des ressources ou applications hétérogènes. La définition d'un plan de contrôle dynamique mais léger, séparé par rapport aux ressources d'exécution, permet aux applications de contrôler et de personnaliser leur environnement d'exécution, ce qui offre une architecture du système d'exploitation très flexible. L'intégration des mécanismes de protection dans ce plan permet également de réduire le surcoût d'autorisation sans compromettre la sécurité, grâce au mécanisme "one-time check ", lors de la création de liaisons jusqu'au prochain changement de politiques d'autorisation. L'utilisation de l'approche G-ABAC pour spécifier les politiques d'autorisation rend l'architecture d'autorisation "policy-neutral ", i.e. l'architecture supporte des modèles de politiques d'autorisation de

différents types. La séparation claire entre les attributs d'autorisation et les règles du G-ABAC augmente également la granularité de la spécification des politiques. Le noyau à base de composants permet de reconfigurer dynamiquement les modules de contrôle d'accès lors de l'exécution. Du point de vue de la mise en oeuvre d'un canevas logiciel autonome, il est également nécessaire de fournir une politique d'adaptation pour guider la reconfiguration dynamique du noyau VSK. La spécification d'une telle politique est réalisée dans le chapitre 7.

0.1.5.3 Spécification de la politique d'adaptation

Le canevas logiciel d'auto-protection présenté dans les chapitres 4, 5 et 6 ne précise pas la spécification des politiques d'adaptation qui permettent guider la sélection ou la re-configuration des politiques d'autorisation en fonction de l'environnement. Les politiques d'adaptation sont généralement spécifiées dans un formalisme ad hoc telles que des règles "if-then-else " qui ne sont pas intuitives pour les utilisateurs. Elles manquent généralment de mécanismes de vérification pour corriger des erreurs. Un autre aspect manquant est l'intégration de ces politiques dans notre canevas logiciel d'auto-protection.

Des langages dédiés (Domaine-Specific Langues ou DSL) offrent de nombreux avantages pour atteindre ces objectifs en prenant en compte la simplicité, la vérification, et l'intégration les politiques en exécution. Le chapitre 7 de la thèse présente un DSL pour décrire les politiques d'adaptation. Le DSL est basé sur l'approche "Event-Condition-Action ou ECA" et sur les taxonomies d'attaques et de contre-mesures. Il permet de réaliser un compromis entre la sécurité et d'autres aspects tel que l'économie d'énergie pour la prise de décision. Un mécanisme de traduction qui raffine dynamiquement le DSL vers un exécutable et l'intègre dans le canevas logiciel d'auto-protection précédent est également illustré. Les différents acteurs intervenant dans le processus de la spécification de politique d'adaptation sont identifiés : l'administrateur de système utilise les politiques d'adaptation génériques (Generic Adaptation Policy ou GAP) pour spécifier les politiques d'adaptation d'une manière intuitive. GAP est ensuite contrôlée pour garantir la fiabilité lors de l'exécution, et traduite en politique d'adaptation autonome (Autonomic Adaptation Policy ou AAP) qui sera intégrée à notre canevas logiciel d'auto-protection en exécution. Pendant cette intégration, un compromis entre la sécurité et d'autre aspects peut notamment guider le processus de raffinement.

Actuellement, un canevas de spécification et d'application de DSL appelé yTune est en cours de développement. Il propose un éditeur et un analyseur syntaxique pour la spécification et la vérification des différents DSL. yTune est un métalangage pour la définition et la mise en oeuvre de DSL. Les travaux en cours se basent sur l'implémentation des mécanismes de raffinement pour un DSL d'auto-protection dans l'analyseur d'yTune, et le couplage de la chaîne d'outils DSL avec le canevas logiciel d'auto-protection ASPF. Dans l'avenir, nous prévoyons également d'améliorer le DSL d'auto-protection avec des taxonomies d'attaques et de contre-mesures plus complètes et réalistes. A travers des ontologies dédiées à la sécurité, une boucle de rétroaction peut détecter les intrusions, prendre une décision en prenant en compte les différents aspects et proposer des réactions.

0.1.6 Perspectives

L'objectif de cette thèse était de réaliser un canevas logiciel d'auto-protection de bout en bout pour les systèmes pervasifs. Les résultats d'évaluation montrent la faisabilité et l'efficacité. Les travaux à venir envisagent l'application du canevas logiciel d'auto-protection pour d'autres types des systèmes informatiques ou l'extension du canevas logiciel vers d'autres aspects que la sécurité.

0.1.6.1 Auto-protection pour d'autres types de systèmes informatiques

Une question en suspens est de savoir si ce canevas logiciel d'auto-protection peut être utilisé pour protéger d'autres types de systèmes informatiques que les systèmes pervasifs. Les systèmes informatiques actuels ont plusieurs problématiques telles que la complexité croissante, l'évolution continue et un surcoût d'administration élevée. Certaines solutions de sécurité existantes se concentrent généralement sur une de ces problématique sans porposer une approche globale pour l'ensemble des problématiques. De plus, les administrateurs doivent manuellement configurer les mécanismes de protection ce qui implique un surcoût élevé et peut induire des dysfonctionnements et des pertes de performances. Nous croyons que l'auto-protection constitue une approche de conception inévitable pour gérer la sécurité des systèmes informatiques émergents.

L'extension de notre canevas logiciel d'auto-protection est basée sur une architecture à trois couches. Le canevas de protection a besoin d'une séparation claire entre l'exécution des applications, les mécanismes de contrôle et les fonctions autonomes afin de réduire la complexité. De plus, l'approche à base de politiques permet une adaptation dynamique. Toutes les mises à jour des modules de sécurité peuvent être réalisées grâce à la modification de politiques en laissant l'infrastructure sous-jacente constante. Le contrôle en temps réel est un autre défi de conception important. La reconfiguration dynamique des mécanismes de protection doit être réalisée pendant l'exécution. Malheureusement, très peu de systèmes informatiques existants fournissent des solutions de reconfiguration dynamique. Ce verrou technique devrait être enlevé dans le futur.

La validation de notre canevas logiciel d'auto-protection dans le contexte de cloud computing est une extension d'application de notre canevas. Nos premières expériences montrent que le canevas logiciel côté serveur peut être réutilisé. Les politiques d'autorisation sont personnalisése en fonction de chaque hyperviseur et sont délivré à travers le cloud pour contrôler les accès locaux. Toutefois, le canevas logiciel côté terminal est dépendant de la plateforme sous-jacente. Le prototype du côté terminal du système pervasif a été implanté sur Fractal / Think dans lequel la reconfiguration dynamique peut être facilement réalisée. Malheureusement, dans une infrastructure cloud, chaque serveur est équipé de son propre hyperviseur. Pour l'hyperviseur Xen, la reconfiguration dynamique est impossible. Nous ne pouvons pas créer ou supprimer dynamiquement des liaisons pour gérer l'autorisation d'accès. Une solution existante consiste à utiliser des crochets statiques par lesquels chaque demande d'accès doit être contrôlée. Une autre solution prometteuse est de ré-implanter les systèmes informatiques existants sur une plate-forme à base de composants. Cette solution permet d'encapsuler les applications sous format de composants.

Des interfaces et des contrôleurs spécifiques peuvent être insérés pour chaque composant. Cependant, cette solution nécessite une charge de travail énorme.

0.1.6.2 Extensions à d'autres aspects

Une autre extension de nos travaux est d'appliquer la même approche pour d'autres aspects, par exemple la qualité de service ou l'économie d'énergie. Nous croyons que l'architecture à trois couches pourra être applicable et que l'approche à base de politiques pourra également être utilisée pour guider les comportements. Dans ce cas d'autres types de politiques doivent être appliquées au lieu de politiques d'autorisation.

Notre canevas d'auto-protection utilise des politiques d'autorisation pour le plan de contrôle et les politiques d'adaptation pour le plan autonome. Afin d'étendre le canevas logiciel vers d'autres aspects, nous avons besoin de spécifier d'autres catégories de politiques de contrôle. Par exemple, pour améliorer la qualité de service, certains types de politiques de qualité de service seront intégrées dans le plan de contrôle. Les politiques d'adaptation du plan autonome doivent donc tenir compte d'autres aspects pour guider l'exécution du système. Ceci nécessite un nouveau DSL plus générique permettant de combiner les aspects correspondants. D'autre part, le contrôle à l'exécution reste toujours un obstacle. Certaines adaptations ont besoin des mécanismes supplémentaires, par exemple pour réaliser le changement des protocoles de communication ou le remplacement de module de codage au niveau du matériel. Toutes ces exigences nécessitent des mécanismes de reconfiguration dynamique, ce qui représente actuellement un réel verrou technologique.

Abstract

Advances in pervasive system are rapidly taking us to a novel frontier in security, revealing a whole new landscape of threats. In open and dynamic environments, malicious terminals may enter a network without being detected, and various malwares may invisibly install themselves on a device. While roaming between heterogeneous networks which are adjusted for their own protection requirements, a device may also take advantage of security policy conflicts to gain unauthorized privileges. In an embedded setting including limited and often unstable computing and networking resources, denial of service attacks are somewhat easier, with little lightweight security countermeasures. Finally, these decentralized, large-scale systems make end-to-end security supervision difficult, with the risk of some sub-system security policies not being up-to-date. These threats can only be mitigated with security mechanisms which are highly adaptable to conditions and security requirements.

Moreover, the administration overhead of security infrastructures usually remains high. Operations to achieve the administration become increasingly complex which would be out of control. One promising direction initiated by IBM is to extend context-awareness to the security mechanisms themselves in order to make them autonomic. In this approach, protection schemes are automatically adapted at run-time according to the actual security requirements of the environment.

Our work applies autonomic computing to conventional authorization infrastructure. We illustrate that autonomic computing is not only useful for managing IT infrastructure complexity, but also to mitigate continuous software evolution problems. However, its application in pervasive systems calls for a collection of design building blocks, ranging form overall architecture to terminal OS design. In this thesis, we propose:

- A three-layer abstract architecture: a three-layer self-protection architecture is applied to the framework. A lower *execution space* provides running environment for applications, a *control plane* controls the *execution space*, and an *autonomic plane* guides the control behavior of the *control plane* in taking into account system status, context evolution, administrator strategy and user preferences.

- An attribute-based access control model: the proposed model (Generic Attribute-Based Access Control) is an attribute-based access control model which improves both the policy-neutrality to specify other access control policies and flexibility to enable fine-grain manipulations on one policy.

- A policy-based framework for authorization integrating autonomic computing: the policy-based approach has shown its advantages when handling complex and dynamic systems. In integrating autonomic functions into this approach, an Autonomic Security Policy Framework provides a consistent and decentralized solution to administer G-ABAC policies in large-scale distributed pervasive systems. Moreover, the integration of autonomic functions enhances user-friendliness and context-awareness.

- A terminal-side access control enforcement OS: the distributed authorization policies are then enforced by an OS level authorization architecture. It is an efficient

OS kernel which controls resource access through a dynamic manner to reduce authorization overhead. On the other hand, this dynamic mechanism improves the integrability of different authorization policies.

- An adaptation policy specification Domain Specific Language (DSL): all the adaptations of this end-to-end self-protection framework are controlled by some high-level strategies called adaptation policies. A specification DSL for such policies is given which takes into account various aspects for adaptation decision.

Implementations of the terminal-side OS and the network-side server show the feasibility to realize the design and fulfill the requirements such as the flexibility of run-time control, efficiency of protection mechanism, and integration of autonomic functions. The results of evaluation for both a local micro-benchmark and an end-to-end global benchmark show that such a framework provides strong and yet flexible security while still achieving good performance, making it applicable to build self-protected pervasive systems.

List of Figures

3.1 Cluster Life-cycle . . . 30
3.2 Node Life-cycle . . . 31
3.3 Smart Home Scenario . . . 32
3.4 Shopping Mall Scenario . . . 34
3.5 A military surveillance scenario . . . 35
3.6 3-Level Autonomic Security Architecture . . . 39

4.1 G-ABAC Approach . . . 53
4.2 Access Control List Policies in G-ABAC . . . 55
4.3 Multiple Security Level Policies in G-ABAC . . . 56
4.4 Organization-Based Access Control Policies in G-ABAC . . . 58
4.5 G-ABAC Specification Model . . . 59

5.1 The ASPF Overall Design. . . . 70
5.2 The Resource Model. . . . 71
5.3 The Security Model. . . . 72
5.4 The Self-Protection Model. . . . 72
5.5 The Self-configuration Model. . . . 73
5.6 A Cluster Extended Model. . . . 74
5.7 The Node Extended Model. . . . 76
5.8 The Cluster Implementation Model. . . . 77
5.9 The Node Implementation Model. . . . 77
5.10 The Cluster-level Self-protection Control Loop. . . . 78
5.11 The Node-level Self-protection Control Loop. . . . 79
5.12 The Authorization Architecture. . . . 80

6.1 VSK Architecture . . . 95
6.2 VSK One-time Check Sequence Diagram . . . 95
6.3 The ACM Request Process with MLS. . . . 97

7.1 Definition and Use of Security Adaptation Policies. . . . 105
7.2 DSL Core Model. . . . 106
7.3 Attack Taxonomy. . . . 107
7.4 Countermeasure Taxonomy. . . . 108

7.5 A GAP Sample Specification. 109
7.6 Adaptation Policy Transformation. 110
7.7 A Typical Generic Adaptation Policy. 112
7.8 GAP fulfilling Property 1. 112
7.9 GAP fulfilling Properties 1 and 2. 113

8.1 Global Extension. 118
8.2 VSK-based System Overview 119
8.3 Execution Space 120
8.4 VSK Kernel 122
8.5 Authorizations Overhead of ACM 124
8.6 Authorization Validation and Enforcement Overheads of VSK 125
8.7 VSK vs Microkernel 126
8.8 Access Control Attribute Reconfiguration 127
8.9 Access Control Rule Reconfiguration 128
8.10 Access Control Policy Reconfiguration 128
8.11 Kernel Occupation Rate between Microkernel and VSK 129

9.1 The Cluster Implementation Model. 133
9.2 The Node Implementation Model. 134
9.3 A Basic ASPF Implementation. 135
9.4 an iPOJO-based ASPF Implementation 138
9.5 ASPF Service-oriented Implementation. 139
9.6 Cluster-Level Self-protection Latencies 139
9.7 Node-Level Self-protection Latencies 140
9.8 Benchmarking Self-Protection Capabilities: Principles 140
9.9 Benchmarking Self-Protection Capabilities: Results 141

10.1 An Adaptable Quarantine Zone. 145
10.2 Cloud Resource Model. 146
10.3 Cloud Security Model. 146
10.4 Machine Extended Model. 148
10.5 VSB Extended Model. 149
10.6 System Extended Model. 150
10.7 The SECloud Authorization Architecture. 151

List of Tables

2.1 Autonomic Frameworks Overview 20

4.1 Access Control Models Comparison 49

5.1 Policy-Based Frameworks Comparison 68

6.1 The Operating Systems Comparison. 93

7.1 GAP Event Specification (Attack Taxonomy). 109
7.2 GAP Reaction Specification (Countermeasure Taxonomy). 109

Contents

0.1 Introduction i
0.1.1 Emergence de l'informatique pervasive i
0.1.1.1 Système auto-protégeable ii
0.1.2 Approche et contributions iv
0.1.2.1 Approche iv
0.1.3 Organisation de la thèse vi
0.1.4 Contributions vii
0.1.4.1 Architecture de sécurité autonome vii
0.1.5 Contrôle d'accès générique à base d'attributs (G-ABAC) ix
0.1.5.1 Canevas logiciel autonome d'administration de politiques de sécurité x
0.1.5.2 Noyau de sécurité virtuel (VSK) xi
0.1.5.3 Spécification de la politique d'adaptation xii
0.1.6 Perspectives xiii
0.1.6.1 Auto-protection pour d'autres types de systèmes informatiques xiii
0.1.6.2 Extensions à d'autres aspects xiv

1 Introduction **1**
1.1 Pervasive Systems 1
1.2 Motivation For Self-protecting Systems 2
1.2.1 Some Background on Autonomic Computing 2
1.2.2 Self-protection Challenges 3
1.3 Approach and Contributions 4
1.3.1 Approach 4
1.3.2 Main Contributions 5
1.4 Outline of the Document 6

I Research Context **7**

2 An Overview of Autonomic Computing **9**
2.1 Introduction 9
2.1.1 Research Areas 9

2.1.2 Common Features of Autonomic Systems 10
2.2 A Taxonomy of Autonomic Systems 12
2.2.1 Self-* Properties 12
2.2.2 Autonomic Maturity 13
2.2.3 Autonomic Architectures 13
2.2.4 Autonomic Enforcement Paradigm 14
2.2.5 Adaptation Policy Approaches 15
2.2.6 Dynamic Reconfiguration 17
2.3 Some Existing Frameworks 17
2.4 Summary 20

3 System Modeling 23
3.1 Our Definition of Pervasive Systems 23
3.1.1 Infrastructureless Architecture 24
3.1.2 Dynamic Topology 25
3.1.3 Multiple Concerns 26
3.2 Abstract Modeling 27
3.2.1 Architecture Model 27
3.2.2 State Model 27
3.2.3 Detection Model 28
3.2.4 Evolution Scheme 28
3.2.4.1 Initial States 28
3.2.4.2 Transitions 29
3.2.5 Life-cycles 30
3.2.5.1 Cluster Life-cycle 30
3.2.5.2 Node Life-cycle 31
3.3 Scenarios and Countermeasures 32
3.3.1 Malicious Node Attack in Smart Home 32
3.3.2 Malicious Application Attack in Shopping Mall 34
3.3.3 Denial of Service Attack in Military Field Surveillance 35
3.3.4 Applying Authorization as Countermeasures 36
3.4 Autonomic Security Architecture 37
3.4.1 Policy-based Approach 37
3.4.2 Functions vs. Autonomic Maturity 38
3.4.3 3-level Architecture 38
3.5 Summary 40

II Design 41

4 Generic Attribute-Based Access Control (G-ABAC) Approach 43
4.1 Access Control Requirements 44
4.1.1 Policy-neutrality of Specification and Administration 44
4.1.2 Decentralization of Access Control Validation 44

4.1.3 Efficiency of Authorization Enforcement 45
4.1.4 Flexibility of Policy Configuration 45
4.1.5 Other Requirements 45
4.2 A Short Survey of Access Control Models 45
4.2.1 Discretionary Access Control (DAC) 46
4.2.2 Mandatory Access Control (MAC) 46
4.2.3 Role-based Access Control (RBAC) 47
4.2.4 Attribute-based Approach 47
4.2.5 Summary 48
4.3 Contribution 49
4.4 G-ABAC Definition 50
4.4.1 Entities 50
4.4.2 Attributes of Entities 51
4.4.3 Rules 52
4.4.4 G-ABAC Overview 53
4.4.5 G-ABAC Policy Definition 54
4.4.6 Specification of Existing Policies using G-ABAC 54
4.4.7 G-ABAC Specification 59
4.4.7.1 G-ABAC Specification Model 59
4.4.7.2 Expression Format 59
4.5 Administration Support 60
4.5.1 Attribute Mutability and Revocation 61
4.5.2 Policy Customization 61
4.6 Summary 62

5 Autonomic Security Policy Framework **63**
5.1 ASPF Design Requirements 64
5.1.1 G-ABAC Policy Support 64
5.1.2 Scalability 64
5.1.3 Consistency 64
5.1.4 User-Friendliness 65
5.1.5 Context-awareness 65
5.1.6 Other Requirements 65
5.2 A Short Survey of Policy Administration Mechanisms 65
5.2.1 Access Control Administration Models 66
5.2.2 Policy-based Frameworks 67
5.2.3 Summary 68
5.3 ASPF Overview 69
5.4 ASPF Design 70
5.4.1 Overall Design 70
5.4.2 ASPF Core Model 71
5.4.2.1 Resource Model 71
5.4.2.2 Security Model 72
5.4.2.3 Autonomic Model 72

5.4.3 ASPF Extended Model . . . 74
5.4.3.1 Cluster Extended Model . . . 74
5.4.3.2 Node Extended Model . . . 75
5.4.4 ASPF Implementation Model . . . 76
5.4.4.1 Cluster Implementation Model . . . 76
5.4.4.2 Node Implementation Model . . . 77
5.4.4.3 A Double Control Loop for Self-Protection . . . 78
5.5 Authorization Architecture . . . 79
5.6 Execution Support Mechanisms . . . 81
5.6.1 Delegation Approach . . . 81
5.6.2 Dynamic Reconfiguration . . . 81
5.6.3 Authorization Policy Enforcement . . . 81
5.7 Summary . . . 82

6 Virtual Security Kernel (VSK) 83
6.1 Requirements . . . 84
6.1.1 G-ABAC Support . . . 84
6.1.2 Efficiency . . . 85
6.1.3 Run-time Control . . . 85
6.1.4 Other Requirements . . . 85
6.2 Related Work . . . 85
6.2.1 OS Architecture . . . 85
6.2.2 Dynamic Reconfiguration . . . 87
6.2.3 Authorization Enforcement . . . 90
6.2.3.1 Single Policy Enforcement . . . 90
6.2.3.2 Policy-neutral Architectures . . . 91
6.2.4 Summary . . . 92
6.3 VSK Overview . . . 92
6.4 VSK Architecture . . . 94
6.4.1 Virtual Kernel (VK) . . . 95
6.4.2 Access Control Monitor (ACM) . . . 96
6.5 Execution Support . . . 98
6.5.1 Kernel Reconfiguration . . . 98
6.5.2 Revocation . . . 98
6.5.3 Multiple Policy Approach . . . 99
6.5.4 Functionality Extensions . . . 99
6.6 Summary . . . 100

7 Adaptation Policy Specification 101
7.1 Design Requirements . . . 102
7.1.1 Intuitive Representations . . . 102
7.1.2 Self-managed . . . 102
7.1.3 Enabling Trade-off between Multiple Concerns . . . 102
7.1.4 Easy for Integration . . . 102

7.2 Related Work . . . 103
7.3 Main Features . . . 103
7.4 DSL Design Principle . . . 104
7.4.1 The DSL Approach . . . 104
7.4.2 Main Actors of Autonomic Security Management . . . 105
7.5 A DSL for Self-protection . . . 105
7.5.1 Taxonomy of Attacks . . . 106
7.5.2 Taxonomy of Countermeasures . . . 107
7.5.3 Generic Adaptation Policies (GAP) . . . 107
7.5.4 An Example of GAP Specification . . . 108
7.6 From DSL to Run-Time Representation . . . 109
7.6.1 Autonomic Adaptation Policies (AAP) . . . 110
7.6.2 A Sample Translation . . . 111
7.6.2.1 GAP Definition . . . 111
7.6.2.2 GAP → AAP Translation . . . 112
7.7 Summary . . . 113

III Validation **115**

8 VSK Validation **117**
8.1 VSK Implementation . . . 117
8.1.1 Overview of Implementation Framework . . . 117
8.1.2 VSK Architecture Overview . . . 119
8.1.3 Execution Space . . . 120
8.1.4 Control Plane Implementation . . . 121
8.1.5 Authorization Policy Implementation . . . 123
8.2 VSK Evaluation . . . 124
8.2.1 Authorization Overheads of ACM . . . 124
8.2.2 Authorization Validation and Enforcement Overheads of VSK . . . 125
8.2.3 Comparison with Microkernel . . . 126
8.2.4 Reconfiguration Overhead of VSK . . . 127
8.2.5 Kernel Occupation Rate . . . 128
8.2.6 VSK Qualitative Evaluation . . . 129
8.3 Summary . . . 131

9 Validation of the End-to-end Framework **133**
9.1 A Basic ASPF Implementation . . . 134
9.1.1 Cluster-side ASPF . . . 134
9.1.2 Node-side ASPF . . . 136
9.1.3 Summary . . . 136
9.2 Second ASPF Implementation . . . 136
9.2.1 Platform Overview . . . 136
9.2.2 ASPF Implementation . . . 137

9.3 Evaluation of the End-to-End Framework 138
9.3.1 End-to-End Response Time 139
9.3.2 Resilience 140
9.3.3 Security Evaluation 141
9.4 Summary 142

10 Cloud Computing Validation **143**
10.1 Cloud computing Environments 143
10.2 Towards Self-Protecting Clouds 144
10.3 Cloud Self-Protection Scenario 144
10.4 ASPF Core Model 145
10.4.1 Resource Model 145
10.4.2 Security Model 146
10.5 Extended Models 148
10.5.1 Machine Extended Model 148
10.5.2 VSB Extended Model 149
10.5.3 System Extended Model 149
10.6 Authorization Architecture 150
10.7 Summary 150

11 Conclusion **153**
11.1 Summary of Contributions 153
11.2 Perspectives 154
11.2.1 Self-protection for Other Types of IT Systems 154
11.2.2 Framework Extensions to other Concerns 155

Chapter 1

Introduction

1.1 Pervasive Systems

Pervasive system was firstly introduced in 1991 by Mark Weiser in [147] to describe integration of back-end computing and communication to daily activities. A pervasive system is an environment in which human can ubiquitously interact with equipments embedded with capabilities of computing and communication. Therefore, information treatment by these equipments can guide and ameliorate daily activities of human being (e.g., usage of Global Position System to help driving).

Advances in embedded systems and wireless communications turned pervasive systems into a reality. Miniaturization of computing equipments, particular that of embedded systems, makes human-machine interaction more invisible. Recent innovations in chip design and nano technology allow fewer consumption for embedded systems which facilitate their life-cycle management. Secondly, greatly improved wireless communication capabilities connect these embedded devices in an efficient and hidden manner. These two major technological advances have permitted the emergence of pervasive systems corresponding to the design of Mark Weiser.

A pervasive system in this thesis refers to a network (consisting of terminals) which is:

- distributed and decentralized: terminals are physically distributed in one area and are connected through wireless communication. Despite their physical localization, wireless communication makes them logically available to others. Furthermore, the distributed architecture usually calls for decentralized coordination of terminals to meet execution environments.

- dynamic and open: terminals, especially mobile terminals, can join and leave pervasive networks at any time which characterizes openness of pervasive systems. The openness makes system architecture dynamic, i.e., system topology highly depends on time. The dynamic feature of system architecture calls for a highly flexible system modeling to support various conditions.

- large-scale and complex: the size of such systems may become huge with hundreds or thousands of terminals. Coordination of these terminals arrives a high level of

complexity. It handles with these terminals, their connections, makes a part of them work together, etc. Instead of traditional solutions for a small number of terminals, large-scale pervasive systems should apply an efficient solution.

Three working scenarios as a smart home network, a shopping mall network and a wireless sensor network are given in Chapter 3 in order to illustrate these major features.

1.2 Motivation For Self-protecting Systems

Pervasive systems are more open, dynamic, and large-scale than traditional distributed systems. They unravel a whole new landscape of rapidly changing threats and of heterogenous security requirements, calling for strong and yet highly flexible security mechanisms. Managing protection "by-hand" in such a setting becomes far too complex. Thus, the *autonomic approach* to security management [47] is a major step forward to address those issues, a system now being able to protect itself without or with minimal human intervention.

In this thesis, we are interested in the setting of self-protection for pervasive systems, addressing realizability of making a pervasive system self-protecting. Such systems should allow their terminals and network-side servers counteracting against threats ranging from hardware or OS attacks to network attacks. Unfortunately, ubiquitousness of threats and dynamism of pervasive systems make self-protection hardly easy. Within the autonomic approach, we apply autonomous control theories for conventional protection mechanisms, extending the infrastructure towards a framework where threats can be autonomously eliminated.

1.2.1 Some Background on Autonomic Computing

IBM firstly initiated the principle of autonomic computing defining as system self-management, freeing administrators from low-level task management while delivering more optimal system behavior [90]. Autonomic computing constitutes an effective set of technologies, models, architecture patterns, standards, and processes to mitigate the management complexity of dynamic computing systems using feedback control. It meets some IT system tendencies as:

Increasing Complexity of IT Systems IT applications together with their execution environments seem in a dramatic growth in terms of complexity, and manual administration can no more be effective. A huge IT system consisting of hundreds or thousands of applications calls for a terrible number of configurations for each administration which is out of control of human. Maintainability becomes a major bottle-neck. Autonomic computing provides tools to autonomously administer systems by enabling administrators to focus on high-level strategies rather than low-level operational tasks.

Continuous Evolution of Software Systems Software systems are under constant development, can never be fully specified, and are subject to constant adjustment and

adaptation [141]. This involves two parts: evolution of design requirements which leads to system updates; and that of executing environment which results in system adaptation. The former asks for an extensible infrastructure to meet evolutional design requirements. The latter illustrates the fact that the execution environment is not known a priori at design time and, hence, the application environment cannot be statically anticipated [87]. This drives software adaptation. Autonomic computing takes care of both evolutions by autonomously reconfiguring its functionalities, architectures, and administration policies.

Increasing Overhead of Administration Administration overhead has already become a crucial factor for complex IT systems. The industry spends billions of dollars to maintain their systems. Configuration by hand may also induce disfunctions and dramatic loss. Autonomic computing is seen as a way of reducing total cost of ownership of complex IT systems by allowing reconfiguration and optimization driven by feedbacks on systems behavior. Formal verification and checking in autonomic computing also guarantees correctness of configurations and manipulations.

1.2.2 Self-protection Challenges

Self-protection was defined as one of the four main properties of autonomic computing [90]. It drives us to add autonomic functionalities in existing or emerging protection frameworks. In this thesis, we define a self-protection framework over which applicative systems can be implemented. As a matter of fact, the whole execution system, ranging from the hardware level to the application level, is protected by the framework. However, several challenges exist in the setting of self-protection.

End-to-End Security for the Framework Such open pervasive systems become increasingly vulnerable to malicious activities covering different aspects from the hardware level to the application level. This is also called ubiquitousness of threats. Therefore, an effective protection framework should cover all these aspects and defend against threats from all the levels.

Flexibility of Control over Applications A self-protection framework usually acts as a separated framework cooperating with an applicative system. It protects applications by applying some supplementary mechanisms. Therefore, this calls for run-time control over applications. We need a flexible platform enabling dynamic reconfiguration over the applicative system.

Efficiency of Protection Mechanisms Overhead of existing protection frameworks remain high. This overhead includes two parts, respectively that of computing and communication. The former refers to computing resources allocation for complementary checking control. The latter targets communication traffics of protection coordination. An efficient protection framework should thus mitigate all these two kinds of overhead.

Feasibility for Self-management Systems In several domains such as context-aware system or cloud computing, whole systems run in an autonomic manner without user intervention. Consequently, their protection should become autonomic. We cannot manually mitigate threats of systems which are autonomously executing.

1.3 Approach and Contributions

1.3.1 Approach

Despite recent progress in autonomic computing or security enforcement, self-protection remains relatively little addressed. Its design as well as implementation is far from a mature autonomic framework, lacking:

1. a generic architecture of reference for organization of building blocks;
2. a fundamental but extensible approach of security enforcement;
3. a simple design pattern to regulate and complete autonomic features;
4. an efficient paradigm to support run-time enforcement and control.

Four main approaches are applied corresponding to these lacks:

An End-to-End Protection Architecture An end-to-end protection architecture provides a generic structure which may be used for various systems. We call a protection framework end-to-end if it can protect against both terminal-level and network-level attacks. Terminal-level local attacks refer to threats against OS or applications of a terminal. Network-level global attacks usually menace a whole network or a part of the network. A conventional isolated local protection system cannot counteract against global attack since it does not have a global view about the whole system. This asks for an end-to-end protection architecture for pervasive systems.

An Access Control Approach Access control is chosen as fundamental meanings to enforce security in our proposed framework. It is usually a basic security module implemented in OSes which restricts access to resources by computing permissions. Additionally, other security functionalities such as trusted management and privacy also depend on access control. Access control usually involves a policy and an enforcement infrastructure. The policy specifies access permissions, and the infrastructure illustrates their validation.

An Autonomic Computing Deployment Autonomic computing can be built at the hardware level (e.g., self-diagnosing hardware [12]), OS level (e.g., self-configuration upgrades or security management [83]), middleware level (e.g., self-healing infrastructure), or application level (e.g., self-tuning application [102], or optimization of performance and service levels). Our framework regulates security at two levels, using separate autonomic feedback loops for self-protection of the network and of terminal: (1) a terminal-side

control loop operates settings of local security modules for protection and optimization; (2) and a network-side control loop achieves a global counteraction and improvement in coordinating a set of terminals in the network.

A Convenient Run-time Control One approach to cope with continuous evolutions (see Section 1.2.1) is to break the traditional division among development phases by suspending some activities from design time to run-time. That is, a software system includes a static infrastructure is fixed at design time and a set of dynamic software components which can be updated and replaced are used at run-time. The policy-driven approach has successfully demonstrated its flexibility and generality for this requirement [138]: fixed system functionalities are governed by a set of variable policies. As the context changes, other policies may be selected to activate within system functions better adapted to its new environment. Furthermore, a run-time platform is integrated which facilitates flexible control over these policies.

1.3.2 Main Contributions

The previous design approaches identify some basic design guidelines to build a self-protection framework. Various technologies can be selected to achieve these guidelines. The main contributions explain developed and implemented mechanisms in our protection framework. The proposed framework consists of:

- A three-layer abstract architecture for self-protection: a three-layer self-protection architecture is defined. A lower *execution space* provides a running environment for applications, a *control plane* supervises the *execution space*, and an *autonomic plane* guides the control behavior of the *control plane* in taking into account system status, risk evolution, administrator strategy and user preferences.

- An attribute-based access control approach: the proposed approach (called Generic Attribute-Based Access Control) applies attribute-based formulization for authorization which improves both policy-neutrality to specify a wide range of access control policies and flexibility to enable fine-grained manipulations on policies.

- A policy-based framework for authorization to realize autonomic security management: the policy-based approach has shown its advantages when handling complex and dynamic systems. An Autonomic Security Policy Framework (ASPF) provides a consistent and decentralized solution to administer authorization policies in large-scale distributed pervasive systems. The integration of autonomic features also enhances user-friendliness and context-awareness.

- A terminal-side security kernel for access control enforcement: the distributed authorization policies defined previously are enforced by an OS-level authorization architecture. This efficient OS kernel called VSK controls resource access in a dynamic manner to reduce authorization overhead. This dynamic mechanism also enables to support different authorization policies.

- A Domain-Specific Language (DSL) for adaptation policy specification: all adaptations of our end-to-end self-protection framework are controlled by high-level strategies called adaptation policies. A DSL to specify such policies is given which takes into account several aspects for adaptation decisions.

1.4 Outline of the Document

This PhD work starts with a detailed introduction of the execution context in `Part I`. The definition of major design choices to construct an autonomic framework is given in Chapter 2. This also explores existing solutions to determine limitations and drawbacks. A system modeling coping with dynamism and flexibility of pervasive systems is given in Chapter 3. Moreover, this model proposes an extensible means to describe multiple aspects such as risk level, QoS, performance,etc. Three working scenarios as well as their threats and corresponding countermeasures are also presented in this chapter.

Contributions are elaborated in `Part II`. In Chapter 4, we introduce G-ABAC, a promising attribute-based access control approach. Although G-ABAC is not a really formulized access control model, it explores attribute-based paradigm to specify a wide range of authorization policies. It also shows advantages to support various administration models. Based on G-ABAC, an administration framework (ASPF) achieving management of distributed policies in pervasive systems is proposed. On one hand, ASPF can be recognized as a framework which realizes administration of G-ABAC access control policies. On the other hand, it extends traditional authorization architecture in integrating autonomic functionalities to simplify management. The terminal-side enforcement of G-ABAC policies is realized by VSK which is a flexible and lightweight OS-level authorization architecture. It can support access control policies specified by G-ABAC. Additionally, it enables dynamic reconfiguration of access control policies and can validate requests in combining decisions of two access control policies. All these features make VSK an efficient and effective terminal platform for the self-protection framework. ASPF and VSK are main contributions of this thesis, and will be respectively elaborated in Chapter 5 and Chapter 6. A Domain-Specific Language (DSL) for specification of adaptation policies is described in Chapter 7. This DSL allows to capture trade-offs between security and other concerns such as energy efficiency during the decision making phase. A translation mechanism to refine the DSL into a run-time representation, and to integrate adaptation policies within legacy frameworks is also illustrated.

The framework validation is given in `Part III`. Based on our previous results [95] of the *E2R* project, the terminal-side implementation is achieved and evaluated in Chapter 8 within the *Mind* project and *Pronto* project. The evaluations [81, 83] show the efficiency and flexibility of the OS-level authorization architecture. Afterward, Chapter 9 realizes, ASPF, the network-side implementation. Moreover, the evaluations of the end-to-end framework [82, 14] are achieved within the *SelfXL* project. These evaluations illustrate the efficiency of the two autonomic loops in terms of response time and resilience. Finally, another design validation to apply the main approaches in the field of cloud computing is presented in Chapter 10. It is part of an in progress project which copes with the security aspect of cloud computing.

Part I

Research Context

Chapter 2

An Overview of Autonomic Computing

Autonomic computing is an efficient paradigm to simplify complexity of IT systems and to facilitate end-user manipulations. This chapter gives an overview, ranging from basic concepts to implemented frameworks, of autonomic computing. In Section 2.1, we review some research areas and describe common features of systems applying autonomic computing. Section 2.2 presents some taxonomies for such systems. A short survey of some existing autonomic frameworks is given in Section 2.3 which examines these frameworks with regard to the defined taxonomies. Finally, Section 2.4 proposes concept guidelines to design our self-protection framework.

2.1 Introduction

Autonomic computing is inspired from a biological metaphor, and is derived from the autonomic nervous system that governs functions of human body without any conscious perception or effort [90]. Its main objective is to minimize human intervention and to enable users to concentrate on their areas of interest. Thus, the autonomic approach lets systems self-manage themselves to execute low level operations in order to achieve high level tasks.

2.1.1 Research Areas

Autonomic computing has been applied to different domains, including IT systems (*autonomic systems*), communication and networks (*autonomic networking*), and protection (*autonomic security*).

Autonomic Computing: *Autonomic computing* was originally designed as a control mechanism for the management of software applications. This approach [90, 86, 110] usually consists of four steps: (1) monitoring and collecting information from a variety of sources such as sensors or specific detection systems; (2) aggregating this information and

reasoning out a high level representation of the system context; (3) making adaptation decisions corresponding to the environment evolution; (4) triggering a set of reactions to perform the decisions.

Autonomic System: An *autonomic system* is the application of autonomic computing to IT systems. It is usually divided into a set of *autonomic elements* [148] which manage their own behavior in accordance with administration strategies. Thus, an *autonomic element* must be self-managed which establishes and maintains collaborations with other *autonomic elements* to meet execution requirements, either by adjusting their execution settings, or by reconfiguring running applications.

Autonomic Networking: *Autonomic networking* [66] aims to simplify the management of communication infrastructures by minimizing manual interventions - particularly for large-scale networks where numerous individual entities are connected and interact with each other. It improves manageability by continuously reacting to system or context evolutions through the use of some self-adaptive protocols or algorithms [118].

Autonomic Security *Autonomic security* [47] enhances user-friendliness of security enforcement technologies, which leads to build run-time frameworks with implicit protection. Background self-protection frameworks usually derive and adapt security configurations or protection mechanisms with respect to high-level security policies.

2.1.2 Common Features of Autonomic Systems

Control Loops: *Autonomic computing* applies control loops to automate infrastructure management. A control loop consists of a set of components that work together to drive systems towards desired states. IBM proposed a reference model called MAPE-K (*Monitor, Analyze, Plan, Execute, Knowledge*) to model the control loop [85]. In this model, sensors or probes *monitor* the execution context to collect system state data [96]. Then, this data is *analyzed* to abstract system status and predict potential evolutions. The *planning* step enables to decide how to *execute* adaptations on the current system and enrich system *knowledge* for later decisions [110].

This model provides a good overview of the main activities in the control loop. In order to design an *autonomic security system*, we believe that the five activities should be implemented through well-identified building blocks. Diversity of system architectures makes realizations of the MAPE-K loop very different from one to another, e.g., distributed sensors may cooperate with a central entity which realizes the *analyze-plan-execute* step; or distributed sensors can be controlled by decentralized entities which make decisions in a P2P manner. Section 2.2.3 gives a detailed classification of such architectures.

Complex Structure: Autonomic computing is generally applied to large-scale systems with complex organizational structures. Therefore, complexity of these systems makes self-management out of control. One promising and popularly accepted approach is to divide these systems into smaller parts called *autonomic cells* [90, 51]. An *autonomic*

cell fulfills requirements concerning performance, reliability, availability and security. It manages itself in adjusting its internal parameters, state, structure or execution behavior. Usually, an *autonomic cell* is also recognized as an atomic unit that offers services to other cells. They can collaborate together to achieve higher level services, and autonomously maintain global design requirements during execution. Hence, the complexity is reduced through the decomposition.

Separation of *Autonomic Manager* from *Managed Resources*: An *autonomic cell* is also viewed as being composed of one or more *managed elements* that perform basic functions with an *autonomic manager* to coordinate configuration, inputs, and outputs of the managed elements. The main activity of the *autonomic manager* constitutes the control loop, often referred to the MAPE-K model. It captures measurements from *managed resources* via sensors or probes and then if necessary tunes the resources through effectors.

The separation of control mechanisms (the *autonomic manager*) from execution operations (the *managed resources*) simplifies system architecture. It also improves reusability of control mechanisms for heterogeneous resources. The mechanisms dedicated to the organization of low-level resources with high-level *autonomic managers*, to the coordination and synchronization of *autonomic managers* will be discussed in Section 2.2.4.

Policy-driven Approach for Behavior Control: Alternatively, the distributed architecture of pervasive systems and the *autonomic cell* design may induce inconsistencies or conflicts. An *autonomic system* should make all its cells work towards a common goal. A popular and effective solution is to apply the policy-driven approach. Locally, an *autonomic manager* applies a policy to guide or optimize behavior of its *managed resources*. Globally, all local policies are derived from one abstract master policy.

We believe that this approach can simplify organization and administration of *autonomic systems*. In a dynamic context, the whole policy enforcement infrastructure remains constant while policies are re-evaluated. A number of existing policy classes are presented in Section 2.2.5.

Self-description of Managed Resources: *Managed resources* should provide a high-level description about their states and environments (reflection), i.e., based on raw data of probes or sensors, an *autonomic system* should abstract high-level representations of system and environment states. Unfortunately, non-determinism of execution and diversity of environment [109] make it difficult to reach such a self-description. Non-determinism induces unforseen system states. Diversity leads to imprecise and incomplete knowledge of context. A self-describing system should thus both capture existing *managed resources* and be more flexible and extensible to cope with uncertain events or cases and propose deterministic descriptions.

Dynamic Reconfiguration: An *autonomic manager* uses effectors to enforce decisions through predefined interfaces for setting modification, system topology evolution, application execution control and change of execution behavior. Dynamic reconfiguration

improves run-time control over *managed resources*. A classification of dynamic reconfiguration levels will be elaborated in Section 2.2.6.

2.2 A Taxonomy of Autonomic Systems

To build an autonomic system, several design decisions should be made. We need to clarify application domain of such a system, if it is used for performance optimization, healing or for protection. We need to foresee the *autonomic maturity degree*. We should determine its global architecture. If the system separates *autonomic managers* from *managed resources*, enforcement paradigm by *autonomic managers* to *managed resources* needs to be determined. If the system applies the policy-based approach, we also need to determine which kind of policies to be used. Finally, *dynamic reconfiguration* is another choice which enables different degrees of run-time control over the system. In this section, a taxonomy of autonomic systems is given. It clarifies main categories of design decisions for autonomic systems and describes potential choices for each category.

2.2.1 Self-* Properties

This section presents a taxonomy for different technologies to achieve autonomic computing. IBM specifies four main properties of *autonomic systems*: self-configuration, self-optimization, self-healing and self-protection [90].

Self-configuration: As defined by IBM [51], *self-configuration* realizes dynamic adaptations in an evolutional environment. Following high-level objectives, a *self-configuring* system performs a series of elementary configuration actions based on executing platform requirements or user preferences. The dynamic adaptation helps ensuring continuous efficiency of the IT infrastructure, resulting in greater flexibility.

Self-optimization: The objective of a *self-optimizing* system is to maximize performance or Quality of Service (QoS) with respect to design requirements. Performance improvement is realized by automatically monitoring and tuning resources which bring the system to one of desired states. Particularly for large-scale systems, this property becomes a promising challenge since optimizations for such systems should take into account different concerns such as security or energy consumption.

Self-healing: A *self-healing* system can discover, diagnose and react to disruptions. Since fault may occur in the system, the *self-healing* approach launches reactions to failures or early signs of potential failures. It guarantees that the functions of the system are correctly fulfilled.

Self-protection: A *self-protecting* system can anticipate, detect, identify and protect against a number of threats. It allows the whole system to consistently enforce security and privacy policies. Especially in the pervasive context where every device may become

an access point to external networks, the protection should be context-aware and efficient given the evolution of the environment [80, 128].

2.2.2 Autonomic Maturity

A well-designed *autonomic system* aims at improving and automating system management. IBM classifies the maturity degree of autonomic management functionalities into five levels: basic, managed, predictive, adaptive and autonomic [75].

Basic Level: The *basic level* defines a system which is manipulated by highly skilled staffs. No control or checking mechanisms is provided from, faulty operations from users that may induce disruptions. In order to guarantee safety, user interfaces of such systems are limited: only a set of predefined and verified operations are permitted. IBM believes that most of current IT systems are at this level.

Managed Level: A *managed level* system includes a monitoring sub-system to collect execution or context information. This information may usually be abstracted into a high-level representation to capture the system state. Users may benefit from this information to achieve more relevant operations. For example, an intrusion detection system may cooperate with an existing application. When a threat is detected, users can stop the application to reduce risks.

Predictive Level: The *predictive level* provides some rules which indicate reactions in accordance to recognized system status. It compares the state information with predefined templates, suggests corresponding configurations, and proposes actions which will be carried out by users.

Adaptive Level: A system at the *adaptive level* takes into account not only monitoring information, but also performance requirements and user preferences for adaptation decisions. It aggregates and reasons on all this information and produces an adaptation decision. The decision can be automatically carried out with adequate reconfiguration actions. A system at this level may also be called a self-adaptive system.

Autonomic Level: Finally, a fully *autonomic* system can dynamically adjust its administration strategies according to its system status and circumstances. When the system evolves or new elements occur in the system, obsolete strategies are discarded. We believe that an *autonomic level* system should update its administration strategies based on the evolution of the system state and context.

2.2.3 Autonomic Architectures

The autonomic architecture refers to arrangement of independent and collaborative *autonomic cells* and realization of MAPE-K loops. It exposes organization and interactions of the control loops. Generally, a central control loop is easier for system administration.

However, due to the distributed feature of pervasive systems, multiple control loops are usually integrated. Whether to select one or several control loops is an important design choice for an *autonomic system*.

Centralized Architecture: Centralized architecture features a single *autonomic manager* which is responsible to manage the entire system. The manager has a global view of all *managed resources*, and all run-time events of resources are captured by this manager. Therefore, processing and administration are centralized.

The centralized control ensures consistency of distributed adaptations and makes the system easy to supervise. However, the main shortcoming of such an architecture is the complexity of its structure and limited scalability. Particularly for large-scale systems, a huge number of resources may be controlled by the *autonomic manager* which leads to a significant administration, computing and communication overhead.

Decentralized Architecture: A decentralized system delegates control of resources to local managers. Each of these managers is attached to one *autonomic cell* to realize the MAPE-K model. They are able to communicate with each other in a *P2P* manner to synchronize and reason about global coordination.

This type of architecture fills in some limitations of the centralized design. The computing and communication complexity is reduced by delegating its tasks to each decentralized manager, and scalability is also enhanced. Furthermore, responsiveness of such architecture is much higher. However, decentralization may induce inconsistency of global properties or behaviors, since local optimizations do not necessarily optimize global behaviors.

Hybrid Architecture: The hybrid design combines the advantages of the centralized and decentralized architectures while limiting their respective drawbacks. It uses local managers for each *autonomic cell*, and is equipped with a centralized manager to coordinate these local managers. In this paradigm, a centralized manager is a superior authority to local managers [51]. A local manager must always take into account adjustments imposed by his supervisor. The essence of this design is that high-level managers deal with abstract strategies while low-level managers carry out operational manipulations. This architecture reduces the complexity of higher-level managers while maintaining their global vision.

2.2.4 Autonomic Enforcement Paradigm

The autonomic enforcement paradigm describes arrangement and implementation of *autonomic managers* with their *managed resources*. It is an essential design choice for *autonomic systems*.

Tight-coupling Paradigm: In the case of *autonomic systems* which do not have a significant separation between *autonomic managers* and *managed resources*, control mechanisms are usually embedded in resources to realize the four autonomic steps which is called

tight-coupling. Typical examples are autonomic algorithm for routing tables to optimize routing and self-organizing protocol for networking.

This paradigm is found in early autonomous systems [86] which integrate autonomic activities into applications (resources). These applications become more complex because they should not only carry out their functional services, but also take care of realization of the MAPE-K model. Maintainability is a shortcoming: once the same applications are deployed in another system, control functionalities need to be re-implemented.

Agent-based Paradigm: Autonomic computing was initially inspired from the agent-based approach [149] where agents control behavior, actions and internal states of executing applications. They collaborate together to achieve some services or tasks with minimal human intervention. With the *agent-based paradigm*, each agent implements an autonomic control loop that is responsible for adaptation via defined administration interfaces.

The separation of control from execution simplifies management of resources and improves maintainability. Unfortunately, since agents are highly coupled with specific applications, any modifications of applications demand updates on the corresponding agent. The dependency between applications and agents remains a constraint, we can hardly find generic agents for heterogeneous applications.

Container-based Paradigm: With the *container-based* paradigm, each application runs inside a container which is in charge of functional and non-functional control. This paradigm is usually implemented through Component-Based Software Engineering (CBSE) [40], where applications can be decomposed into components and their interactions are captured through interfaces. Containers encapsulating components can provide standard reconfiguration interfaces for later manipulations.

This paradigm provides a complete separation between control logic and execution logic. Various types of resources can be implemented within homogenous containers. With the help of CBSE, back-end compilers can autonomously insert containers for each application. One drawback is that local containers do not have an overall view, i.e., each container only takes care of its application. However, a fully *autonomic system* would call for global coordination between these containers.

Plane Separation Paradigm: The *plane separation* paradigm goes one step further towards a completely separated and totally controlled discipline. It regroups containers into a dedicated plane to facilitate global control. Service Level Agreements (SLAs), user preferences and control strategies are used in this plane, and will be translated to low-level configurations for each container. Containers achieve these configuration operations through predefined interfaces of managed resources.

2.2.5 Adaptation Policy Approaches

An autonomic control loop coordinates a system using high-level goals expressed in format of policies [91]. A policy consists of rules telling how to react, e.g., a rule determines types

of decisions and actions to perform. Therefore, a policy is a set of considerations that are designed to guide the behavior of managed resources [51].

Integrated Algorithm: Initial autonomous systems do not really apply the policy-driven approach, but they apply some algorithms or protocols which integrate autonomic functions. For instance, in autonomic networking, routing algorithms can self-optimize performance.

If-Then-Else (ITE) Policy: Early adaptation policies are usually in the form of *if-then-else (ITE)* rules. With this approach, systems are modeled and compared with pre-established templates to decide whether to launch reactions and which reaction to launch. A set of reactions may be attached to each state. System adaptations are then carried out by applying reactions according to different states.

Event-Condition-Action (ECA) Policy: An ECA policy [137] applies events to launch adaptations. A rule of an ECA policy takes the form "when *event* occurs and *condition* holds, then execute *action*". Comparing to the *if-then-else* rules, a rule of an ECA policy may be applicable not only for one state, but also for a set of states. Unfortunately, for a system of large-size, the ECA policy may hold a huge number of rules. Their management, especially resolution of rule conflicts is still little addressed.

Goal Policy: The *goal* policy approach makes systems choose configurations towards desired states. It is based on graph theory where optimized state transitions are identified in order to reach desired states. However, these policies are limited to systems which have a complete and precise view of their status, i.e., policy decisions rely on deterministic system modeling. For open and dynamic systems which have dynamic structures, this approach can hardly be applied.

Utility Function Policy: In order to overcome the drawback of *goal* policies, it is possible to define an *utility function* that attaches a utility value for each state. User preferences and performance requirements which are independent from system overall architecture can be taken into account with the function. Within one state, the policy guides execution behavior by maximizing values of the *utility function*.

Ontology: *Ontologies* are particularly helpful to structure or characterize adaptation policies which allow better communication and reuse of defined concepts. They are extensible and also enable to reason on adaptations. Unfortunately, a unified ontology which takes care of different aspects of *autonomic systems* is missing.

Different specification approaches are discussed in this section, an adaptation policy may apply one or several approaches to specify its rules. For example, a framework which integrates *action policies*, *goal policies*, and *utility function policies* is described in [91]. The framework also shows how to translate *utility function policies* to *goals policies*, *utility function policies* to *action policies*, and *goal policies* to *action policies*.

2.2.6 Dynamic Reconfiguration

The last step of the MAPE-K model performs reactions through effectors in order to enable reconfiguration over *managed resources*. Dynamic reconfiguration mechanisms bring flexibility to the manipulation of *managed resources*. This sub-section gives different granularity levels of dynamic reconfiguration.

Setting Update: One simple interpretation for control over running applications (*managed resources*) is to modify their parameters. This approach is also called *parameter adaptation* [110]. Since the execution of these applications relies on the parameters, setting updates indirectly exercise control over application execution.

Application Configuration: Within this approach, applications are execution units to be manipulated. Operations such as starting or stopping application execution, enabling or disabling some services are recognized as configurations. Contract to the previous solution, the *application configuration* allows a direct control over running applications.

Architecture Evolution: For a system consisting of several applications, an *autonomic manager* should administer both individual applications and global architecture. The *architecture evolution* reconfiguration involves creation and destruction of applications as well as creating or removing bindings between applications.

Behavior Control: Finally, non-functional aspects should also be reconfigured if an *autonomic system* attempts to completely control over its managed applications. Some aspects like security, QoS, performance can be indirectly controlled through the usage of strategies. One applied strategy is translated into some low-level reconfigurations that can be achieved through the previous three classes of reconfiguration.

2.3 Some Existing Frameworks

An overview of some existing autonomic frameworks is given in this section which illustrates advantages and drawbacks for design choices.

AutoMate: AutoMate [11] is a middleware which addresses self-configuration, self-optimization and self-protection for grid computing. It defines an architecture of three layers respectively: an autonomic system layer for fundamental functionalities and inter-component communication mechanisms; an autonomic component layer to manage and control components; and an autonomic application layer for the optimization of component execution. In addition to these three layers, several *engines* take care of specific objectives like access control, rule reasoning, and context-awareness.

A system built on AutoMate consists of autonomic components which are units of execution and control. The whole framework is based on a decentralized architecture since there does not exist a central authority to control distributed components. Agents

of different concerns are associated to each component in order to treat corresponding objectives. This may induce some conflicts between decisions taken by different agents. Separation between policy and mechanisms in AutoMate allows abstracting control over the grid infrastructure. Based on the ECA policy, events launch adaptations to handle various contexts. Since AutoMate aims to develop a generic framework of autonomic computing, adaptations are *setting updates*. For instance, for the access control engine, it can dynamically adjust Role Assignments and Permission Assignments to fit different environments.

Rainbow: Rainbow [46] is a generic autonomic architecture that separates a fixed part (*adaptation infrastructure*) from a variable part (*system-specific adaptation knowledge*). The *adaptation infrastructure* implements common functionalities for an autonomic framework. The *system-specific adaptation* is based on the following knowledge: types and properties of components, behavioral constraints, and strategies of adaptation.

Such a framework addresses *self-configuration* to enhance system execution performance and to adapt strategy modification or context evolution. It divides the whole system into three layers: a *system layer*, a *translation layer* and an *architecture layer*. Based on captured information from the *system layer*, the *translation layer* abstracts information to a formalized representation. Afterward, an adaptation decision reconfigures the running system through effectors of the *system layer*. The *centralized* control in the *architecture layer* maintains a complete representation of the system, and compares it with pre-defined template models. This comparison can be seen as the application of *if-then-else* policy to fit different execution states. Since executing systems are totally separated from the *architecture layer*, it eliminates the dependency of such a general framework from various applications. Finally, effectors are implemented as APIs to achieve reconfiguration. To be generic, no specific reconfiguration platforms are defined in the framework. We believe that reconfiguration is of the *setting update* level.

Intel AES System: Intel Autonomic Enterprise Security (AES) system [12] aims at providing an end-to-end framework to defend enterprise IT infrastructures. This system consists of: a self-defending platform protecting individual OSes, a distributed detection and inference system detecting network-scale attacks, and an adaptive framework delivering security policies as countermeasures to network threats. When the detection system identifies threats in the network, it launches an adaptive feedback to produce low-level security policies which will be enforced by the Inter ATM [4] end-points.

Addressing the security self-protection objective, *Intel AES* realizes *adaptive* feedback to enforce protection of IT systems. It is equipped with a *centralized* server at the network-level which collects events from the detection system and delivers customized security policies through the whole IT infrastructure including network and terminal platforms. The adaptation feedback does not directly act over *managed resources*. Instead, it produces security policies to guide their protection behavior. The run-time control thus corresponds to an update of *settings* (security policies) in the infrastructure.

Jade: Jade [49, 140] is a middleware for self-management of distributed applications. It encapsulates and administers legacy applications using the *Fractal* component model [41] to provide a uniform means of application deployment and configuration. Complex environments are managed by different points of view extending the autonomic vision.

The overall architecture of Jade is divided into two layers: a *management layer* and a *legacy layer*. The *management layer* includes common autonomic functionalities together with two specific managers addressing *self-optimization* and *self-healing*. The abstraction of autonomic managers improves extensibility towards other application domains. The *management layer* serves as a *centralized* server to administer managed resources modeled as components in the *legacy layer*. Jade follows the *plane separation paradigm* which puts the *control logic* in a completely separated plane. Adaptation policies were first specified as *integrated algorithms* for *autonomic managers*. A subsequent approach is based on *Domain-Specific Languages (DSL)* which are "*languages tailored to a specific application domain*" [106]. The *Tune* [38] framework provides highly abstracted languages for the specification of autonomic policies. The resulting framework fulfills all requirements of the *autonomic* maturity level. Finally, because of the use of *Fractal*, the architecture of the legacy layer can be dynamically reconfigured by adding, removing components or bindings, that is, the *architecture evolution*.

Auto-Home: Auto-Home together with H-OMEGA [37] is a run-time for residential applications. It applies the Service-Oriented Architecture (SOA) design to facilitate management of dynamism and heterogeneity of domestic devices. These devices are modeled as service-oriented components and administered by the underlying H-OMEGA run-time.

Auto-Home provides a *self-configuration* gateway-side server to manage in-house devices. By gathering context information from managed applications, it achieves *adaptive* control. The framework involves a set of autonomic managers organized in a hierarchy (the *hybrid architecture*). Interaction points can be dynamically inserted into existing applications for interoperability. With the help of the underlying H-OMEGA platform, these interaction points play the role of containers which take care of non-functional aspects of applications. Several types of managers of distinguished roles are identified and implemented in the framework. Each manager applies a specific *algorithm* to treat one aspect. Since non-functional aspects can be reconfigured through the interactions points, this framework provides the *behavior control* as dynamic reconfiguration.

Reaction after Detection (ReD): The original framework of ReD was proposed in [57, 64, 63] to provide countermeasures to intrusion detection alerts. Based on existing intrusion detection systems, attacks are analyzed and abstracted in format of *Intrusion Detection Message Exchange Format (IDMEF)* [62]. Appropriate countermeasures are then proposed with respect to different attacks. A *Policy Instantiation Engine (PIE)* instantiates security policies as countermeasures to eliminate reported attacks. These policies are then enforced through reconfigurations by *Policy Enforcement Points (PEP)*.

The framework was extended as a *self-protection* framework called *Reaction after Detection* (ReD). Three types of reaction loops are defined: low level reactions by PEP;

	Self-* Properties	Maturity	Architecture	Enforcement	Policy	Reconfiguration
AutoMate	Self-configuration Self-optimization Self-protection	Adaptive	Decentralized	Agent	ECA	Setting
Rainbow	Self-configuration	Adaptive	Centralized	Plane	If-then-else	Setting
Intel AES	Self-protection	Adaptive	Centralized	Integration	Goal	Setting
Jade	Self-optimization Self-healing	Autonomic	Centralized	Plane	Ontology	Architecture
Auto-Home	Self-configuration	Adaptive	Hybrid	Container	Algorithm	Behavior
ReD	Self-protection	Autonomic	Hybrid	Agent	ECA+Ontology	Setting

Table 2.1: Autonomic Frameworks Overview

middle level reactions by *Reaction Decision Point*; and high level reactions by PIE. This separation of control loops enhances scalability and corresponds to the *hybrid* architecture. We believe that RdD belongs to the *agent-based control paradigm* by which a set of ReD nodes (agents) are deployed through the whole network. One significant improvement of ReD is the usage of ontologies to enable the mapping process between detection and reaction [59]. It applies OrBAC-based ontologies to describe alerts and security policies. A policy (set of rules) allows the mapping from formalized IDMEF alerts to OrBAC security policy instances. These ontologies together with ECA rules allow reaching the *autonomic* level of maturity by providing semantic description and reasoning. Finally, generated policy instances are enforced by PEPs which translate policies to low level configurations such as closing a port of a firewall. Thus, ReD belongs to the *setting update* reconfiguration level.

2.4 Summary

To construct an autonomic framework, several design choices should be made. We should determine the application domain of such a framework: whether it is designed for *self-configuration*, *self-optimization*, *self-healing* or *self-protection*. We should also foresee its maturity level. Based on execution context, we should select the administration architecture (centralized, decentralized, or hybrid). The enforcement paradigm is another design decision, as the choice of the adaptation policies (*if-then-else*, *event-condition-action*, etc). Finally, some dynamic reconfiguration mechanisms should be provided.

Table 2.1 compares the designs of the previous frameworks. For the *self-protection* of pervasive systems, we figure out that existing solutions may be improved. *Self-protection* is not treated by some of these framework. For the frameworks of *self-protection* (AutoMate, Inter AES, and ReD), they address one or two aspects of security and do not provide an end-to-end framework to cover all levels of system. Most of the frameworks are at the *adaptive* level of maturity. Based on the system state or context evolution, reactions may be triggered for adaptation. However, a framework at the *autonomic* level can also adjust itself to optimize its administration. The two frameworks of the *hybrid* architecture in the table (Auto-Home and ReD) illustrate an efficient architecture for distributed systems, particularly for large-scale systems. This approach is thus selected for

our end-to-end *self-protection* framework. To be a generic framework for various types of systems, the *plane-based* approach of the *enforcement paradigm* provides a completely separated control plane which coordinates *autonomic managers* and supervises *managed resources*. This approach will be applied to our framework. Adaptation policy class is another important design choice. Ontology can not only specifies different classed of policies, but also provide reasoning. The adaptation policy of our framework will thus be based on this approach. Finally, reconfiguration mechanism supports different levels of flexibility for run-time control. Our framework will achieve the *behavior control* level which has a full control over running systems. In the later part of this thesis, we will elaborate the concept and implementation of these design choices.

Chapter 3

System Modeling

Before starting the design of an end-to-end self-protection framework, we characterize pervasive systems in this chapter. A pervasive system embeds computing and communication capabilities in daily devices and makes them transparent to end-users. Therefore, devices can achieve some computing tasks and collaborate together to construct networks sometimes called pervasive networks, and users uses these devices with the help of computing and communication tasks running in the background.

In this chapter, we describe pervasive systems, their execution behaviors as well as security functionalities. An end-to-end self-protection architecture for such systems will be built based on this modeling.

Section 3.1 gives our definition of pervasive systems. The system has an infrastructureless architecture, dynamic topology, and multiple concerns such as security and QoS which should be taken into account for execution control.

Section 3.2 proposes a formulization of the defined systems. This contains an architecture model, a system state model, and a system detection model. It also illustrates run-time evolutions of the systems and life-cycles of their main entities.

Section 3.3 elaborates three working scenarios in order to show application domains of the self-protection framework. It discusses security issues, explains basic security functions, and identifies three classes of attacks such as malicious node attack, malicious component attack and Denial of Service (DoS) attack. Subsequently, it gives some ideas about countermeasures against these attacks.

Section 3.4 briefly introduces principles of the self-protection framework, its three-layer architecture, and organization of autonomic functions.

3.1 Our Definition of Pervasive Systems

A security framework depends on underlying systems to be protected. This section gives our definition of the working pervasive systems which permits to clearly and formally separate its internal parts from its external environments. This section also presents a system modeling that takes into account three aspects: system architecture, run-time topology evolution, and multiple concerns.

3.1.1 Infrastructureless Architecture

The structure of pervasive systems can be dynamically modified by internal evolutions (e.g., migration of devices) or external changes (e.g., context evolution). The definition of system structure should support these evolutions and changes. One approach is to minimize essential building blocks of such an infrastructure.

Definition 3.1 *We define a pervasive system architecture infrastructureless if it is built by a minimal predefined infrastructure with the following principles:*

- *the system has a minimal pre-established infrastructure.*
- *the system separates network-level entities from device-level entities.*
- *the cluster definition abstracts network-level entities.*
- *the node definition abstracts device-level entities.*

Our system model asks for a minimal pre-established infrastructure: a network-side server installed with our self-protection ASPF framework (described in Chapter5) acts as the minimal pre-established infrastructure. Different kinds of devices equipped with VSK OS (described in Chapter6) can dynamically join in the system and collaborate with other devices.

Traditionally, two different approaches, the hierarchical approach and ad-hoc approach, are applied to organize distributed systems. The hierarchical approach divides a system into several sets and each set into subsets, etc. This approach enforces the centralized control over the whole system and simplifies its management. It was proven as an efficient solution for administration of large-scale systems. A typical example is the *Domain Name System (DNS)* [9] which is used to bind *Fully Qualified Domain Names (FQDN)* and network information such as IP addresses. FQDN are uniquely built as a path from a root domain to a sub-domain and finally a name that belongs to the sub-domain. All domains are independently administered by their pre-established infrastructures. However, this approach requires a completely hierarchical infrastructure to handle with each level. The ad-hoc approach considers all entities in same level, there does not exist any prioritized entities to administer others. Selection algorithms may be used to casually choose one entity as a temporal authority of administration. This approach avoids implementing a complete infrastructure, but it induces supplementary computing and communication overheads. We propose a hybrid solution that divides a system into two levels: a network level and a device level. Network-level entities (clusters) administer device-level entities (nodes) to achieve centralized control and to avoid computing and communication overheads. To adapt large-scale systems, this architecture can be extended by adding supplementary levels (see Assumption 3.1).

Network-level entities are modeled as clusters which consist a set of devices. By Assumption 3.2, service composition from the node level to the network level and decomposition from the network level to the node level are out of the scope of this thesis. Moreover, clusters cannot directly interact with others. But they may intersect through some shared devices (see Assumption 3.3).

Device-level entities are modeled as unified nodes. Heterogeneous devices are abstractly described as nodes and are managed interchangeably. A node can be shared by at maximum two clusters as indicated in Assumption 3.3.

Assumption 3.1 *We assume that a pervasive system is divided into the network and node levels. Scalability can be improved by adding supplementary abstraction levels that applies the same modeling principle. But the extension is out of the scope of this thesis.*

Assumption 3.2 *We assume that service composition and decomposition are not taken into account in our framework since its main objective is security enforcement. A framework is supposed to provide these functionalities, but they are not detailed in this thesis.*

Assumption 3.3 *We assume that a node is at least connected to a cluster, and can at most be shared by two clusters simultaneously. The former relation guarantees the determinism of node possession. The latter simplifies the design and implementation of the self-protection framework. Sharing a node by more than two clusters applies the same design principle but is not realized.*

3.1.2 Dynamic Topology

Run-time topology of a pervasive system may evolve according to availability and localization of its entities (cluster or nodes). For example, if a node is destructed or removed, the topology of the attached cluster or clusters is changed.

Definition 3.2 *We define a pervasive system topology dynamic if its run-time architecture permits the following evolutions:*

- *a new cluster can be created.*
- *an existing cluster can be destructed.*
- *a new node can be created and attached to one cluster.*
- *an existing node can be destructed.*
- *a node can be migrated from one cluster to another.*
- *a node already attached to one cluster can join a second cluster.*
- *a node attached to two clusters can be released from one.*

A cluster can be dynamically created or destructed during execution. Once the cluster is destructed, no more operations about this cluster will be demanded.

A new node can be created in the system and join one cluster. Otherwise, physical disappearance of one node is considered as destruction, and may be monitored by a supervision system (see Assumption 3.4).

One node which is attached to a cluster can join another to enable collaboration between these two clusters. When the collaboration is finished, the node can be released from one cluster.

Since this thesis only addresses security issues, these evolutions are not realized in our proposed framework (see Assumption 3.5).

Assumption 3.4 *We assume that a node supervision system is provided which monitors availability and localization of all nodes. However, realization and administration of the supervision system are not given in this thesis.*

Assumption 3.5 *We assume that a management framework is on collaboration with the self-protection framework to achieve the specified evolutions. But the realization of all these evolutions are not described in this thesis.*

3.1.3 Multiple Concerns

In an autonomic framework, execution and context information is analyzed to propose reactions. This information should be integrated into the system modeling in order to have an overall and complete view about system state.

Definition 3.3 *We define concern as execution or context information corresponding to one aspect. Different concerns such as risk level, energy consumption, availability, and computing capability can be integrated in the system modeling.*

Corresponding monitoring systems are supposed to supervise these concerns (see Assumption 3.6). Following concerns may be taken into account for a pervasive system:

- risk level represents vulnerability degree. Different from conventional risk level definition which combines menace probability with damage loss, a simplified version of risk level addresses security effect of threat. In an open and dynamic environment, threats may come from internal parts (e.g., an internal node) or external parts (e.g., an external attacker) which menace not only availability, but also confidentiality and integrity.

- energy consumption defines consumed energy as effect of security function. Different security functions provoke different energy consumption. This concern may be used to find out a trade-off between strong security function and minimal energy consumption.

- availability of a node declares whether it is ready for use. During the execution, a node may disappear from the system due to physical destruction or user turn-off. A supervision system timely checking node availability (see Assumption 3.4) is supposed to exist.

- computing capability shows hardware settings of devices. Because of heterogeneity, a run-time configuration of pervasive systems should take into account hardware

settings of each device. For resource-limited devices like sensors, a simple but efficient security enforcement needs be proposed. For devices of high computing capability like laptops, some strongly secure functions may be applied.

As the whole system is divided into the cluster level and node level, aggregation of concern information between these two levels is another important issue. Different characters of nodes are modeled as node states. Aggregation mechanisms compute cluster level concern values from those of the node level. As explained in Assumption 3.7, we assume that this kind of aggregation is provided.

Assumption 3.6 *We assume that a monitoring system is provided for each concern, but their achievement and administration are not described in this thesis.*

Assumption 3.7 *We assume that aggregation mechanisms for concern information from the node level to the cluster level is supported. But their realization is not presented in this thesis.*

3.2 Abstract Modeling

A pervasive system in our definition is organized as a set of clusters which contain several nodes. Each cluster controls its execution settings such as its run-time topology and state information. Nodes have a variety of characteristics, ranging from hardware performance to software modules. In this thesis, we address the execution of a single cluster, but our approach can be extended to multiple and concurrent clusters.

A system model is defined as $M = (A, S, D)$ where A is a system architectural model, S is a state model involving states of concerns, and D is a detection model having different types of detection elements.

3.2.1 Architecture Model

The architecture model is $A = (C, N, nAss, cPos)$ with:

- $C = \{c\}$ is a set of clusters in the system.
- $N = \{n\}$ is a set of nodes in the system.
- $nAss : N \rightarrow 2^C \backslash \{\emptyset\}$ represents mapping of nodes to their parent clusters.
- $cPos : C \rightarrow 2^N$ represents mappings of clusters to their attached nodes.

3.2.2 State Model

The state model monitors and abstracts different concern of a running system with $S = (V, nSta, cSta)$:

- $V = V_1 \times V_2 \times ... \times V_n$ represents a combination of n concern dimensions. For example, risk level may be one concern dimension V_1 which has four levels such as *very hostile*, *hostile*, *neutral*, and *friendly* with $V_1 = \{\text{very hostile, hostile, neutral, friendly}\}$.
- $nSta : N \rightarrow V$ is a function that associates concern states to nodes.
- $cSta : C \rightarrow V$ is a function that associates concern states to clusters.

3.2.3 Detection Model

Different from the system state modeling which abstracts stable evolutions, a detection model manages time-critical system changes which need a timely treatment. It is defined as $D = \bigcup_{m \in M} D^m$:

- M is a set of detection aspects.
- $D^m = \{d^m\}$ represents one detection aspect that includes a set of detection events. One monitoring system is assumed to be provided for each detection aspect (see Assumption 3.6). For instance, an intrusion detection system is used in the working scenario for the security aspect.

3.2.4 Evolution Scheme

Evolution of a pervasive system may modify cluster-node relation. In this sub-section, we define all accepted transitions for topology evolutions.

3.2.4.1 Initial States

An initial state is hardware and software settings that should be fulfilled before any transition.

Cluster Initial State Before starting any manipulation on one cluster, we specify its initial state as:

- concern aspects are already determined, and related monitoring systems capture and maintain corresponding information for the self-protection framework.
- ASPF, a framework that administers self-protection (see Chapter 5), is installed on a server which takes into account the concern aspects to coordinate self-protection.

Concern dimensions involve internal or external information that a system should take into account for self-protection. They are usually defined by a system architect at the design phase. Once different concerns are determined, corresponding monitoring systems that supervise these concerns should be provided to support the self-protection framework.

ASPF (see Chapter 5) is a framework on the server side that manages self-protection mechanisms of clusters and their nodes. To realize manipulations on clusters and nodes, ASPF should be installed.

Node Initial State A node also calls for some preparations before any transition. We specify the initial state of a node as:

- a VSK OS is installed on the node.
- the node is categorized to one cluster.
- the node profile that describes its characteristics is stored in ASPF.

Within our end-to-end framework, a VSK OS should be installed in each device. VSK protects terminals by enforcing authorization policies which are administered by server-side ASPF. More details about VSK and ASPF will be respectively introduced in Chapter 6 and Chapter 5.

Since a node belongs to at least one cluster, each node should thus be categorized to one cluster at the beginning.

Moreover, since any manipulations of nodes depend on their characteristics, node profiles need to be installed in ASPF.

Finally, since this thesis only addresses protection mechanisms, we do not show how to realize the cluster and node initializations (see Assumption 3.8).

Assumption 3.8 *We assume that the initialization of each cluster or node is already established.*

3.2.4.2 Transitions

Cluster-level Transitions: Transitions are classified into cluster-level transitions and those of the node level. Cluster level transitions represent all actions that can occur to each cluster. We have:

- *cluster creation* enabling the creation of a new cluster in the system.
- *cluster destruction* enabling the destruction of an existing cluster.
- *cluster state evolution* enabling the cluster level state evolution driven by node aspect evolutions. For instance, when a node is compromised, the risk level of its associated cluster or clusters will be increased.

A cluster can be dynamically created or destroyed. Additionally, the state of a cluster may evolve during its execution.

Node-level Transitions: Node level transitions represent all previewed actions for each node. We have:

- *node insertion* enabling a new node to join the system.
- *node removal* enabling an existing node to leave the system.

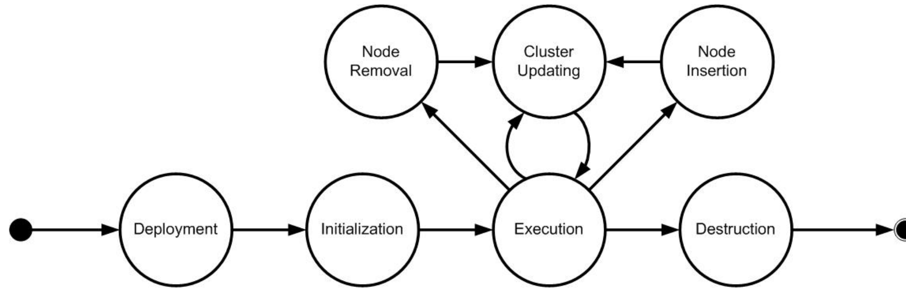

Figure 3.1: Cluster Life-cycle

- *node migration* enabling a node to leave one cluster and joint another.
- *node sharing* enabling a node which is already attached to one cluster to join a second cluster.
- *node sharing removal* enabling a node which is shared between two cluster to leave one cluster.
- *node state evolution* enabling states evolution of a node at run-time.

3.2.5 Life-cycles

Life-cycles illustrate different phases of clusters or nodes and transitions between these phases.

3.2.5.1 Cluster Life-cycle

As shown in Figure 3.1, a cluster may have several phases as:

- *Deployment*: a cluster is administered by an authority which is a software entity residing on the server side. Once the cluster authority is created, the corresponding cluster is thus in the *deployment* phase. The authority manages all nodes which are attached to it, and maintains a *profile* for each node.
- *Initialization*: based on execution settings, some security modules are customized and installed to each node of the cluster. This is the *initialization* phase of the cluster.
- *Execution*: after the *initialization*, nodes of the cluster are protected by the customized security modules. They collaborate together to achieve some tasks, this is the *execution* phase of the cluster.
- *Node Integration*: when a node wants to join the cluster (by the *node insertion* or *node sharing* transition), the cluster is in the *node integration* phase. It checks the node through some authentification mechanisms (see Assumption 3.9).

- *Node Removal*: when one node is destructed, turned off, or leaves the cluster, the cluster is thus in the *Node Removal* phase.

- *Cluster Updating*: after the *node integration* or *node removal* phase, the cluster settings may be updated. During the *cluster update* phase, new security modules are customized and installed into each node of the cluster.

- *Destruction*: when all nodes of the cluster are removed, the cluster is thus turned to the *destruction* phase, and the corresponding software entities in ASPF will be removed.

Assumption 3.9 *We assume that authentification mechanisms are provided for the verification of new nodes joining clusters. But their realizations are not detailed in this thesis.*

3.2.5.2 Node Life-cycle

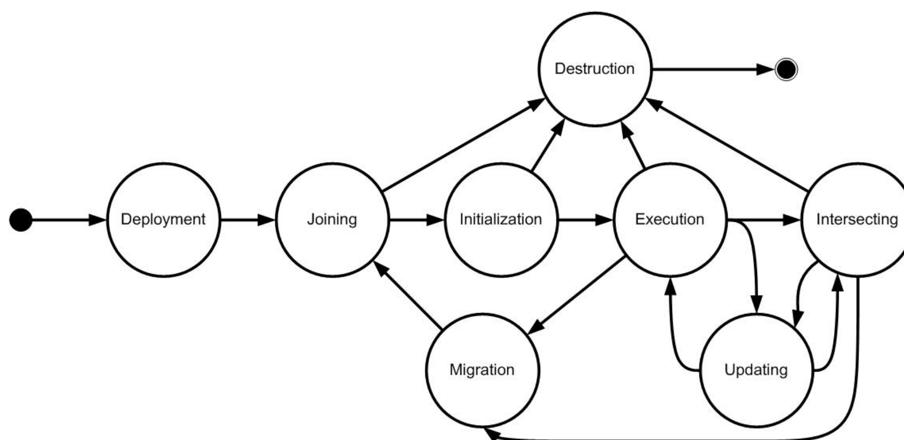

Figure 3.2: Node Life-cycle

As shown in Figure 3.2, a node can have various phases as:

- *Deployment*: a node installed with a VSK OS is in the *deployment* phase. Within this phase, a node proxy is created in the ASPF framework, and the node is ready to join in different clusters.

- *Joining*: authentification of the node is the *joining* phase.

- *Initialization*: for the integration of the node in one cluster, the *initialization* phase installs one or several security modules which depend on requirements of the cluster and profile of the node.

- *Execution*: in the *execution* phase, the node can collaborate with other nodes in the same cluster to realize tasks.

- *Intersecting*: within our framework, a node can interact with at most two clusters at the same time by adopting simultaneously two security modules. This is the *intersection* phase which indicates inter-cluster collaboration.
- *Updating*: when the cluster updates its settings or the context of the cluster evolves, the node is turned into the *updating* phase in which it needs to update its security modules.
- *Migration*: a node can migrate from one cluster to another, this procedure is the *migration* phase. When the node shared between two clusters leaves one cluster, it also returns to this phase.
- *Destruction*: a supervision system periodically checks availability of nodes (see Assumption 3.4). In the case where the node disappears by the supervision system, it is in the *destruction* phase.

3.3 Scenarios and Countermeasures

Security includes three main objectives such as the confidentiality, integrity and availability. In this section, we first present three scenarios of pervasive systems together with corresponding classes of attacks. Then, we propose some countermeasures handling with these three classes of attacks.

3.3.1 Malicious Node Attack in Smart Home

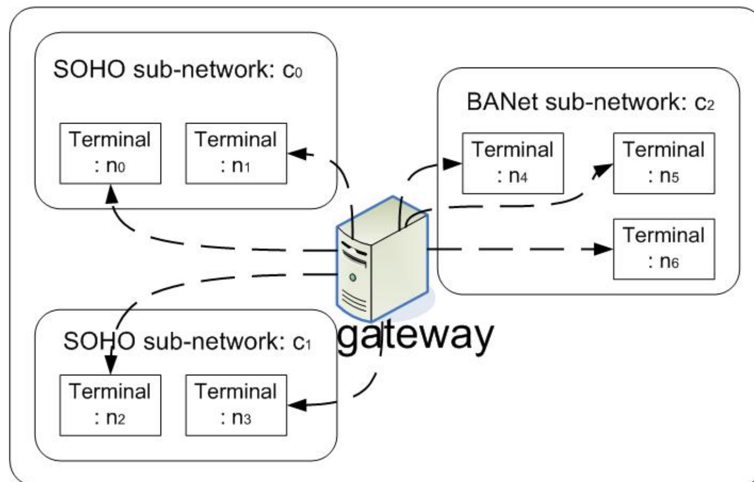

Figure 3.3: Smart Home Scenario

Smart Home Scenario: A working scenario of smart home is firstly analyzed. We address a simple domestic environment where several sub-networks cooperate together: two Small Office/ Home Office (SOHO) sub-networks for storage and process of important

files, and a Body Area Network (BANet) monitoring health-care aspect of user. All these sub-networks are connected and administered through a gateway (see Figure 3.3).

Formally, we model the smart home as $M = (A, S, D)$. The architecture model $A = (C, N, nAss, cPos)$ with:

- $C = \{c_0, c_1, c_2\}$ respectively for the two SOHO sub-networks and BANet.
- $N = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$ for all mobile terminals in these three sub-networks.
- $nAss(n_0) = \{c_0\}$, $nAss(n_1) = \{c_0\}$, and $cPos(c_0) = \{n_0, n_1\}$: two nodes are included in the SOHO network c_0.
- $nAss(n_2) = \{c_1\}$, $nAss(n_3) = \{c_1\}$, and $cPos(c_1) = \{n_2, n_3\}$: two nodes are included in the SOHO network c_1.
- $nAss(n_4) = \{c_2\}$, $nAss(n_5) = \{c_2\}$, $nAss(n_6) = \{c_2\}$, and $cPos(c_2) = \{n_4, n_5, n_6\}$: three nodes are included in the BANet c_2.

For the state model $S = (V, nSta, cSta)$, we take into account two aspects, the risk level V_0 and energy consumption V_1 with:

- $V = V_0 \times V_1$.
- $V_0 = \{$very hostile, hostile, neutral, friendly$\}$.
- $V_1 = \{$critical, high, normal, low$\}$.
- for a given moment, we assume that the node states are.
  - $nSta(n_0) = (friendly, low)$.
  - $nSta(n_1) = (friendly, low)$.
  - $nSta(n_2) = (friendly, low)$.
  - $nSta(n_3) = (friendly, low)$.
  - $nSta(n_4) = (friendly, low)$.
  - $nSta(n_5) = (friendly, low)$.
  - $nSta(n_6) = (friendly, low)$.
- and the cluster states are:
  - $cSta(c_0) = (friendly, low)$.
  - $cSta(c_1) = (friendly, low)$.
  - $cSta(c_2) = (friendly, low)$.

An intrusion detection system consists the detection model $D = D^0 = \{d_{attackDetected}, d_{attackRecovery}\}$. This system contains only two events: one assigning an alert of attack $d_{attackDetected}$, and the other for attack recovery $d_{attackRecovery}$.

Malicious Node Attack Class: A malicious node attack class addresses attacks where the control of a node in the system is gained by an attacker. Then, the attacker may access to sensitive files or information through this node. For example, in our working scenario, a mobile terminal of the less-protected BANet is firstly compromised by an attacker. With the evolution of the system topology, this node may migrate to one SOHO sub-network. It can thus access to sensitive file of this SOHO sub-network. Consequently, the confidentiality of the SOHO is no more guaranteed.

3.3.2 Malicious Application Attack in Shopping Mall

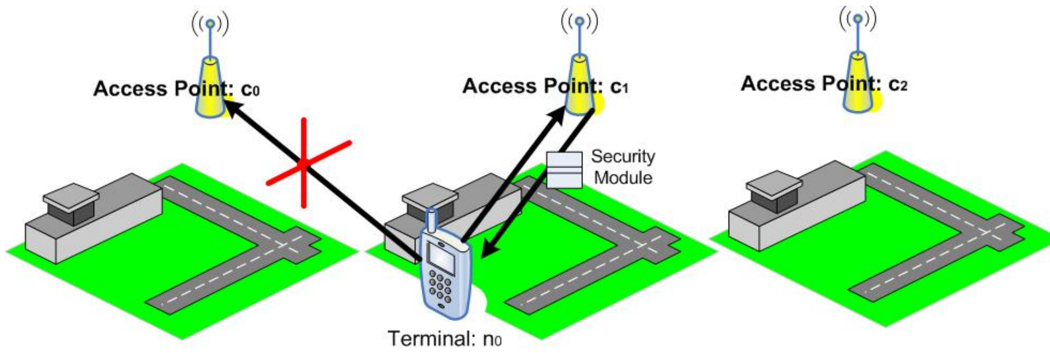

Figure 3.4: Shopping Mall Scenario

Shopping Mall Scenario: In the second scenario, considering a shopping mall where mobile terminals connect to *Internet* through different access points. Each access point calls for a specific security module. When one terminal moves from one WLAN to another, the old security module may no longer be adaptable for the new WLAN. A new security module needs to be downloaded and installed. The updated terminal can then connect to *Internet* through the new access point by the new security module.

Formally, we model the shopping mall as $M = (A, S, D)$. The architecture model $A = (C, N, nAss, cPos)$ with:

- $C = \{c_0, c_1, c_2\}$ for three WLANs;
- $N = \{n_0\}$ for one mobile terminal;
- for a given moment t_0, the terminal connect to the WLAN c_0 with: $nAss(n_0) = \{c_0\}$, $cPos(c_0) = \{n_0\}$, $cPos(c_1) = \emptyset$, and $cPos(c_2) = \emptyset$;
- for another moment t_1, the terminal connect to the WLAN c_1 with: $nAss(n_0) = \{c_1\}$, $cPos(c_0) = \emptyset$, $cPos(c_1) = \{n_0\}$, and $cPos(c_2) = \emptyset$;

Malicious Application Attack Class: Due to non-trusted access points, downloaded security module may be compromised. Malicious security modules may be installed that either affects confidentiality by accessing sensitive files or decreasing integrity by modifying files or data. This class of attacks is thus called malicious application attack class.

3.3.3 Denial of Service Attack in Military Field Surveillance

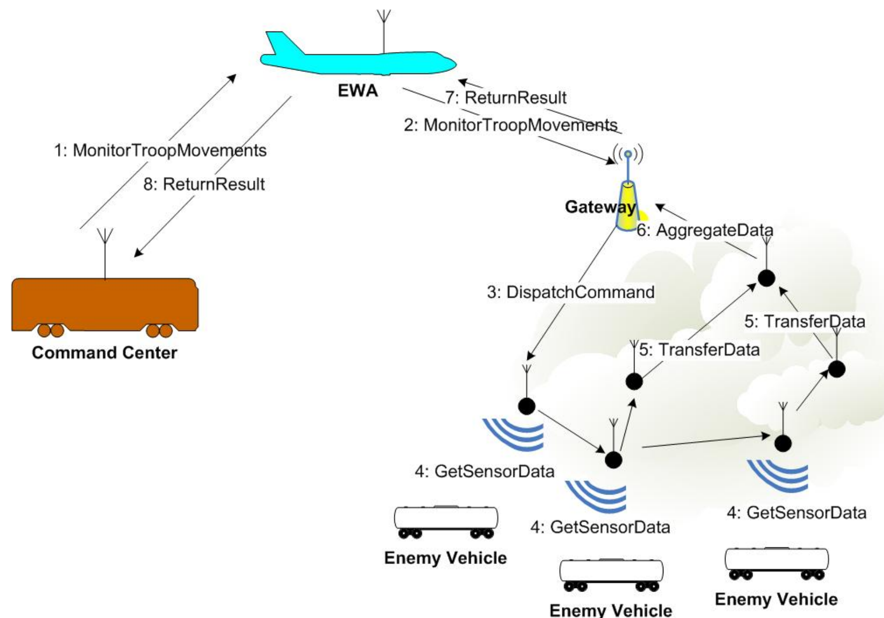

Figure 3.5: A military surveillance scenario

Military Field Surveillance Scenario: The third military surveillance scenario is in the context of Wireless Sensor Network (WSN). Its objective is to track the enemy's movements (e.g., gather information on vehicle positions and velocities) by a pre-deployed WSN [145] (see Figure 3.5). In a war field, a command center aims to monitor movements of the enemy. It sends a *MonitorTroopMovements* command to an Early Warning Aircraft (EWA), which flies above the WSN and transfers this command to a WSN gateway. The gateway then dispatches the command to all nodes in the WSN. Each sensor gathers information on vehicles in the field. An aggregated result is then returned to the gateway. In the end, the gateway sends back the result to the command center via the EWA. Since the formal modeling of a such WSN is similar to the previous two formal modeling, we do not present it here.

Denial of Service (DoS) Attack Class: The DoS attack class compromises availability. In the assumption that a self-protection solution has already been implemented to the WSN, it can automatically protect itself against external attacks. The protection is usually a series of reactions achieved by all sensors in the network. Because of energy limitation of some sensors, an enemy can launch many blind attacks which expend energy of sensors. This is one kind of DoS attacks against the self-protection system. The availability of the network is thus affected.

3.3.4 Applying Authorization as Countermeasures

Applicable countermeasure of our self-protection framework are selection, customization and enforcement of access control policies to defend different classes of attacks. Access control is usually a basic security module applied not only to OSes but also to distributed systems. It restricts access to resources by computing permissions. Other security functionalities such as trusted management and privacy also depend on access control.

An access control sub-system embodies an *access control policy specification*, an *access control enforcement infrastructure*, an *administration model of policy specification* and an *administration model of policy deployment*. The *access control policy specification* provides a formal definition of essential concept elements. The formalization enables proof of properties on the system being designed. An access control policy is defined by the specification which contains a set of authorization rules. Each rule indicates whether an active entity (subject) is granted to perform an operation such as read, write to a passive entity (object). Throughout this thesis, we use the terms *access control policy* and *authorization policy* interchangeably. The *access control enforcement infrastructure* allows low level functions to implement controls imposed by access control policies. The *administration model of policy specification* defines potential configurations on authorization policies. The configurations may change security levels of an object for a Domain Type Enforcement (DTE) policy or add a new role to a subject for a Role-Based Access Control (RBAC) policy. Since such configurations may compromise the projected system, an administration model usually restricts and checks manipulations. Finally, the *administration model of policy deployment* defines the deployment of authorization policies for distributed systems. In this thesis, we apply the Policy-Based Access Control approach [35], where a master policy is customized, refined and deployed in the distributed system. Thus, we define an *access control policy specification*, an *access control enforcement infrastructure* and an *administration model of policy deployment*, but we do not provide any *administration model of policy specification*.

Authorization Policy: As the pervasive system is divided into the cluster and node levels, proposed authorization policies have also a separation of these two levels. A cluster level authorization policy involves potential access permissions of all nodes in the cluster. It is initially defined by a security administrator. As sensitive information is shared between different nodes of a cluster, for the reason of consistency, only one authorization policy is applied to a cluster at any moment, and each node of this cluster applies an authorization policy which is derived from the cluster-level authorization policy.

Bodes may have different capabilities such as computing capabilities or memories, a policy of a node is a simplified version of the cluster policy. The node-level authorization policy is customized for all subjects and objects in the node. For dynamic reconfiguration, the loading, replacement and tuning of node-level authorization policies are enabled. As a node can be shared between two clusters (see Assumption 3.3), each node is able to apply two authorization policies (VSK can support two authorization policies simultaneously). More details about the policy implementation will be introduced in later chapters.

The authorization of pervasive systems involves authorization of applications in one node, authorization of applications in different nodes of one cluster, and authorization of

applications in different nodes of different clusters. Our framework only addresses the first type of authorization (see Assumption 3.10). An extension which takes into account the second and the third types of authorization is given in the context of cloud computing (see Chapter 10).

Assumption 3.10 *We assume that only authorization of applications in one node is taken into account in our framework. Authorization of applications in different nodes is not taken into account.*

Countermeasures against Attack Classes: The malicious node attack class can be defended by applying authentification mechanisms, and by providing a consistent and distributed access control administration model. With this administration model, all applied authorization policies of sub-networks are managed by one authority.

The malicious application attack class can be counteracted by a flexible and dynamic authorization policy administration model. This model allows tuning of security parameters of new applications on their execution behavior. Therefore, all access of new applications will be strictly controlled.

In terms of the availability by the DoS attack class, the separation of a detection system from decision-making seems to be a good solution. All received events will be firstly analyzed by the detection system before launching an adaptation. Thus, no more energy will be consumed by protection enforcement before decision-making. More detailed about the detection system will be elaborated in later chapters.

3.4 Autonomic Security Architecture

With the increasing complexity of protection systems, administration of corresponding security functions together with applied authorization policies becomes an crucial issue. Autonomic computing [90] proposes a promising approach to simplify administration. The protection framework manages itself without or with minimal human interventions and can autonomously react to its state or context evolutions.

3.4.1 Policy-based Approach

One main need for such an autonomic system is to develop an efficient but simple administration framework that supports cooperative and coherent interactions among multiple autonomic components to satisfy a common system-wide objective. The policy-based approach fulfills this requirement [105]. Highly abstracted policy governs behavior of execution where low-level operations are realized autonomically [137]. On the other hand, access control framework also attempts to be based on policies: all resource accesses are controlled by an authorization policy [35, 132]. Referring to the objective of this thesis, proposing an autonomic security framework, the policy-based approach is chosen as a principal basis which facilitates access control.

A policy of such a framework is a collection of rules that define decisions for potential conditions. Hence, it is an administrative means to deal with situations that are likely

to occur. The separation of policy from its enforcement infrastructure enables dynamic change of administration strategy without modifying underlying infrastructure. Therefore, the policy-based framework is able to dynamically control run-time behavior of pervasive systems without affecting its underlying implementation. Two different types of policies are used in our framework, authorization policies serve as predictive rules to control access and adaptation policies for context-aware adaptation over system execution.

3.4.2 Functions vs. Autonomic Maturity

From the perspective of autonomic functions, IBM divides autonomic maturity into five levels: *basic level*, *managed level*, *predictive level*, *adaptive level* and *autonomic level*. Within this definition, we can classify existing autonomic frameworks in following their degrees of maturity. Based on underlying levels, each level provides supplementary control functions that makes systems more autonomic. For example, a framework is classified as *predictive level* only if both *managed level* and *predictive level* functions are fulfilled. In order to achieve the *autonomic level* which is one of the objectives of this thesis, our framework should realize all functions that correspond to these five levels.

The lowest *basic level* can be viewed as execution of applications without any control mechanisms. The *managed level* provides fundamental control mechanisms and the *predictive level* applies some predictive rules beyond the *managed level*. The *adaptive level* takes into account context-awareness for execution optimization. Finally, the *autonomic level* proposes some semantic autonomic strategies to guide adaptation behavior.

3.4.3 3-level Architecture

In the context of pervasive systems with our system modeling, the end-to-end autonomic security framework realizes these five levels by three planes (see Figure 3.6). The *execution space* is an execution environment of applications that plays the role of the *basic level*. The *control plane* regroups both the *managed level* and the *predictive level* which controls the *execution space* by applying predictive rules. In our case, authorization policies are predictive rules to control access of resources. On the top, the *autonomic plane* coordinates adaptation with semantic adaptation policies. This sub-section briefly describes these three planes, and more details about realization of each layer will be introduced in later chapters.

Execution Space: The *execution space* is an environment in which all applications can be launched to achieve some tasks. All software systems without control mechanisms can be seen as an *execution space*. However, since upper levels will enforce control by achieving manipulations, the *execution space* should provide a uniformed representation of all applications with a standard manner of control. *Component-Based Software Engineering* (CBSE) [40] appears as a good solution since it encapsulates all applications and resources in format of components. Standard interfaces are defined and used to control these components. Furthermore, other advantages like dynamic reconfiguration are also provided by CBSE.

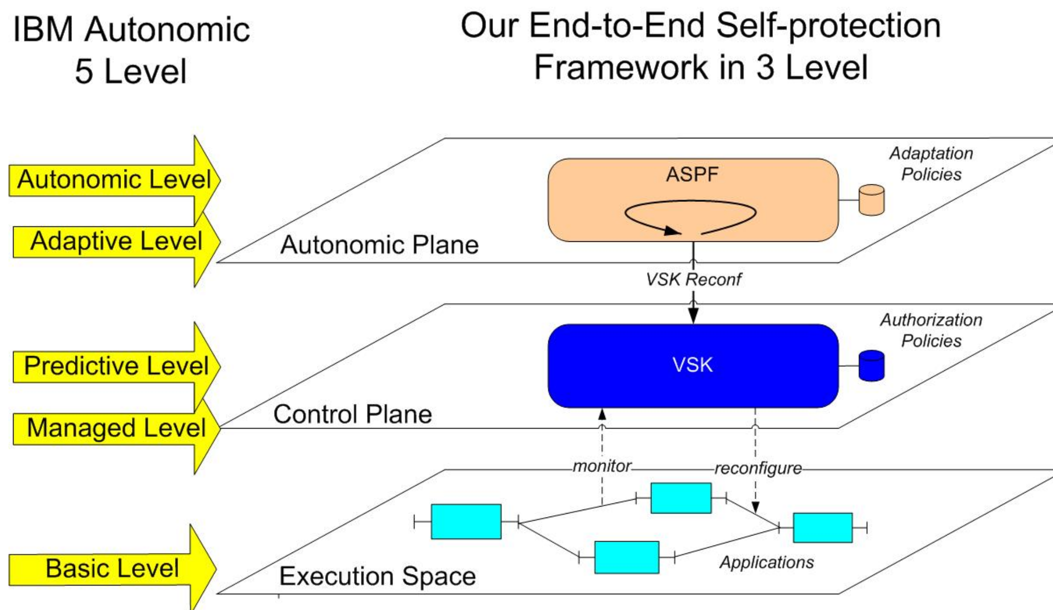

Figure 3.6: 3-Level Autonomic Security Architecture

Control Plane: The *control plane* combines the *managed level* and *predictive level* of the IBM autonomic maturity model. In order to realize the former one, supervision and reconfiguration should be proposed that allow both monitoring and manipulations on the *execution space*. For the *predictive level*, some kinds of predictive rules need to be used that indicate reactions for potential conditions. Since our framework focus on self-protection, authorization policies are applied as a set of predictive rules. An authorization policy is composed of several rules which represent authorization permissions about subjects on objects with operations. Once a component in the *execution space* attempts to access to another component, an access request is firstly captured by the *control plane*. Then, a predictive decision module makes a decision for the request. A reconfiguration module achieves a set of reactions according to the decision.

Autonomic Plane: In an access control framework, predictive authorization rules are usually fixed at the beginning which can not be modified during execution. However, design requirements demand system to be more context-aware where the predictive rules can evolve in accordance to context evolution [100]. For example, one access request can be granted if the system is in a relatively secure context and should be denied in the opposite case. The *adaptive level* proposes functions to perform this predictive policy update based on context information in following an adaptation policy. At the end, to distinct an adaptive system from an autonomic system, semantic adaptation policies should be provided that guide adaptation behavior of the self-protection framework. In brief, the *adaptive level* achieves adaptation but the *autonomic level* guides adaptation behaviors on function of user preferences, execution condition, etc.

3.5 Summary

In this chapter, we motivated and described the model of our working pervasive systems which is a basis for our self-protection framework. As detailed in Section 3.1, the infrastructureless architecture, dynamic topology and multiple concerns are the three main features. A formal definition of these pervasive systems was given in Section 3.2 in which we also specified a set of recognized transitions both at the cluster and node levels. Their life-cycles were also elaborated. In Section 3.3, three working scenarios were identified with some classic attack classes which address different security objectives respectively the confidentiality, integrity, and availability. Some countermeasures were also proposed against these attack classes. Finally, an overview architecture about our self-protection framework was presented in Section 3.4, it is a 3-level policy-based architecture that achieves all necessary functions for an autonomic system. In later chapters, we will describe this framework in detail.

Part II

Design

Chapter 4

Generic Attribute-Based Access Control (G-ABAC) Approach

As described in the previous chapter, access control policies are in the center of the end-to-end framework. The *autonomic plane* selects the most adequate access control policy among a set of potential ones. Subsequently, it customizes and deploys the selected policy through the whole system. The *control plane* enforces customized policies in all nodes. For the administration of access control policy specification, some existing access control models define their own administration models [126, 113] to control or restrict manipulations. Although some access control models are quite expressive in terms of policy specification [98, 116], few models provide full support for various *administration models of policy specification* which is crucial for mobile terminals to be integrated into different networks. The conventional policy-neutral approach [24, 79, 136] should thus be extended to enhance not only expressivity of policy specification but also flexibility for various *administration models of policy specification*.

Although the main contributions of this thesis are the deployment (see Chapter 5) and enforcement (see Chapter 6) of access control policies, we present in this chapter an approach called Generic Attribute-Based Access Control (G-ABAC) that can express a wide set of existing access control policies. This approach facilitates the validation and configuration of access control policies at the OS level for embedded systems. Its support of *administration models of policy specification* is more generic: a large range of administration models can be integrated in the ASPF framework which applies G-ABAC. This chapter gives the description of G-ABAC together with its support for various administration models.

Section 4.1 identifies design requirements for access control policies in the context of pervasive systems: policy-neutrality of specification and administration, decentralization of access control validation, efficiency of authorization, and flexibility of policy configuration.

Section 4.2 discusses some existing access control models with regard to their structure such as: access matrix models, multi-level security models, role-based and attribute-based access control models. However, all these models hold one or several shortcomings given

the considered design requirements.

Section 4.3 reviews advantages of applying the attribute-based approach for authorization in pervasive systems. It highlights the importance to separate administration of attributes from specification of access control policy. G-ABAC also enables run-time configuration of policies and efficient authorization validation.

Section 4.4 defines G-ABAC in detail. It formulizes basic elements and their relations. Several examples illustrate how G-ABAC achieves policy-neutrality by specifying a number of existing access control policies.

Section 4.5 describes administration supports for G-ABAC. Dynamic reconfiguration of policy enables attribute mutability which avoids inconsistency due to attribute modification. Session support allows to control the whole life-cycle of authorization: before, during and after permission validation.

Section 4.6 concludes this chapter by comparing G-ABAC features to the design requirements. Afterwards, it briefly motivates the need for an administration framework described in Chapter 5.

4.1 Access Control Requirements

Within a pervasive system, different access control policies may be applied depending on the situation. One solution is to adopt a generic access control approach that can express a variety of authorization policies in this open and dynamic environment. Since existing access control policies [73, 89] are usually managed by heterogenous *administration models of policy specification*, their diversity calls for a unique approach for such management. In this section, some essential design requirements of a generic access control approach are listed.

4.1.1 Policy-neutrality of Specification and Administration

The use of multiple authorization policies calls for a policy-neutral paradigm in terms of policy specification and administration. Since existing policies use a large number of concepts for access control validation (e.g., MLS [27, 32] defines security levels, RBAC [73] uses the role concept), a generic approach needs to provide some basic concepts by which different authorization policies can be expressed and managed.

Requirement $\mathcal{R}$ 4.1 *The access control approach should be policy-neutral to express and manage a wide variety of authorization policies.*

4.1.2 Decentralization of Access Control Validation

In a pervasive environment which is highly distributed, the access control validation should be designed accordingly. Conventional centralized validation by one authority does not seem adequate. Particularly, when the system becomes large-scale, a huge set of permissions needs to be maintained with a significant performance overhead for access control

validation, and also increasing the complexity of policy administration. Therefore, a decentralized paradigm for access control validation is needed.

Requirement $\mathcal{R}$ 4.2 *The access control validation should support distributed systems through a decentralized paradigm.*

4.1.3 Efficiency of Authorization Enforcement

Within an end-to-end self-protection framework, access control policies are enforced by OSes of terminals. Some types of models are not adequate for resource-limited terminals, and thus should be simplified to make their enforcement efficient. For example, an access enforcement which should invoke a monitoring system to get contextual information is not lightweight. An efficient access enforcement should compute access permissions based on information stored in the kernel. Using external information for OS-level authorization may cause a supplementary mode switch overhead and is considered as non lightweight. Thus, a lightweight yet effective access control approach is needed.

Requirement $\mathcal{R}$ 4.3 *The authorization should be efficient for OS-level enforcement of any kind of embedded terminals.*

4.1.4 Flexibility of Policy Configuration

As access control policies are applied to pervasive systems which are infrastructureless and highly dynamic, context changes may lead to some updates of policies at run-time. The flexibility of policy configuration leads to two requirements: definition of fine-grained concept elements and independence among these elements. The former calls for a fine granularity of policies. The latter means that such concept elements should be independently separated. Therefore, the flexibility of policy configuration should enable adjustment of these concept elements without impacting others.

Requirement $\mathcal{R}$ 4.4 *Policies specified by the access control approach should be flexible to support configurations of policy concept elements.*

4.1.5 Other Requirements

Other requirements such as life-cycle management of authorization, integration of context information in access control validation are not among the main design requirements of G-ABAC. However, some implementations described in Chapter 5 show how G-ABAC may be extended to achieve these requirements.

4.2 A Short Survey of Access Control Models

In this section, we present an overview of some existing access control models from the perspective of the policy specification (the structured concept to describe policies), that is, how to express access control permissions in using different concepts like *domain* and

type in DTE, *role* in RBAC. We believe that the adminstration of policy specification is managed by administration models and it not treated in this thesis. We notably examine the Discretionary Access Control (DAC) models, Mandatory Access Control (MAC) models, Role-Based Access Control (RBAC) models, and finally access control models using the attribute-based approach.

4.2.1 Discretionary Access Control (DAC)

The access control mechanism is defined as DAC if an individual user can set an access control mechanism to allow or deny access to an object [33].

Access Matrix DAC is firstly implemented through an access matrix. The access matrix is a subject-object-operation matrix which indicates permissions of subjects to perform operations on objects as defined in the Lampson model [97]. Access Control Lists (ACL) is most basic implementation of the access matrix in OSes. With ACL, a specific access control list is attached to each resource (object). When a user wants to perform some operations on an object, the OS checks whether the user and his operations are included in the access control list of the object. Capability-based systems [99] are another implementation where the access matrix is on the subject side. Lists of capabilities are tokens containing a permission for a subject to access an object and a relevant operation. As a category of DAC, an resource owner has complete control over all his resources and can assign permissions of his own resources by modifying the access matrix.

The decentralization requirement ($\mathcal{R}$ 4.2) is partially satisfied since some permission tables may be attached to either subjects or objects. Due to the too basic structure of the model, it is hardly used to express other policies as is $\mathcal{R}$ 4.1. The huge size of permission tables in large-scale systems reduces efficiency of access control validation ($\mathcal{R}$ 4.3): an access request needs to be validated by looking through the whole table. The model has the same drawback for the requirement $\mathcal{R}$ 4.4, one update of permission calls for modifications through all ACL or capability lists.

4.2.2 Mandatory Access Control (MAC)

As defined by M.Bishop [33]: when a system mechanism controls access to an object and an individual user cannot alter that access, the control is a Mandatory Access Control (MAC). Hence, all resources are controlled by the OS. Unlike DAC, it is not possible for MAC policies non-administrator users to change access control strategy over resources.

Multi-Level Security (MLS) MAC is firstly implemented through MLS which classifies subjects and objects using lattices [125] and base access control validation on comparison of security levels. In MLS models such as Bell LaPadula (BLP) [27] and Biba [32], all subjects are classified into security levels called *security clearances*, all objects are classified into security levels called *security classifications*. For each access request from a subject to an object with an operation such as read or write, a comparison of the corresponding security clearance and security classification is performed.

Within the access matrix model described previously, subjects and objects are identified by their identities. Access control validation is based on these identities. This is a key drawback for large-scale, distributed systems, since a huge number identities should be managed. MLS separates identities from authorization validation to improve efficiency ($\mathcal{R}$ 4.3) and flexibility ($\mathcal{R}$ 4.4). But the use of the lattice concept limits policy-neutrality ($\mathcal{R}$ 4.1).

Domain and Type Enforcement (DTE) DTE is another DAC implementation. In the DTE model [24, 79], subjects and objects are respectively grouped to domains and types. An authorization validation of a subject, an object and an action is performed by checking the corresponding domain, type and action in the *Domain Definition Table* (this table contains all allowed operations between domains and types).

From the policy specification perspective, DTE simplifies the access matrix by assigning domains and types to subjects and objects in order to improve efficiency ($\mathcal{R}$ 4.3) and flexibility ($\mathcal{R}$ 4.4). Decentralization ($\mathcal{R}$ 4.2) of the DTE authorization is not described in exiting works, but we believe that it is possible to realize.

4.2.3 Role-based Access Control (RBAC)

The RBAC category introduces the notion of role to represent a set of similar subjects having some common access rights. Each role grants a number of permissions. Roles can then be assigned to users on the basis of their specific job responsibilities and qualifications [127, 73]. Users are assigned with roles according to their functions performed in a company or organization to determine whether access will be granted or denied. Therefore, a single access permission to a role can be profited by all assigned subjects of the role.

In order to avoid scalability limitation, RBAC simplifies the access matrix model by assigning roles to subjects. RBAC also enables dynamic assignment of roles and permissions. The usage of roles presents benefits of decentralization ($\mathcal{R}$ 4.2) and flexibility ($\mathcal{R}$ 4.4). It is thus widely applied for context-aware access control models. In such models, the concept of condition makes authorization related to some context information like time or location. Environmental roles are proposed to (de)activate roles of a location in GEO-RBAC [31, 54]. Spacial Role-based Access Control (SRBAC) [114] and Dynamic Role-based Access Control (DRBAC) [154] are extensions of RBAC that are able to dynamically adjust permission and role assignments depending on context information. RBAC enables to achieve a certain level of policy-neutrality ($\mathcal{R}$ 4.1) as it permits to express a number of authorization policies [115]. But the usage of roles together with organizational and context information may reduce efficiency of authorization at the OS level ($\mathcal{R}$ 4.3).

4.2.4 Attribute-based Approach

In the attribute-based approach, an OS examines attributes of subjects and objects to determine whether an access request is granted. Therefore, users are identified by their characteristics (*attributes*) rather than identities.

Generalized Role-based Access Control (GRBAC) GRBAC [53] is an extension of the traditional RBAC model that applies roles not only to subjects, but also to objects and to the environment. In this type of model, the role can also be considered as a security attribute for subjects, objects and the environment. When a subject wants to access an object, corresponding permissions based on their roles should be computed.

Organization-Based Access Control (OrBAC) In OrBAC [89], in order to abstract different kinds of subjects, objects, and actions, the concepts of *role*, *view*, and *activity* are respectively introduced, which may be viewed as particular types of security attributes. These abstractions completely separate concrete security entities from access permissions. Modification of attribute assignments is hidden from authorization validation which improves flexibility ($\mathcal{R}$ 4.4). Policy-neutrality ($\mathcal{R}$ 4.1) is fulfilled since OrBAC is able to express other authorization policies like RBAC. Furthermore, its organization-oriented concept allows to achieve decentralization ($\mathcal{R}$ 4.2).

Attribute-Based Multipolicy Access Control (ABMAC) In the ABMAC model [98], access decisions are made according to attributes of requestors (subjects), resources (objects), and the environment. One advantage of ABMAC is its support of multiple policies at the same time: various polices such as DTE, MLS, or RBAC can be simultaneously installed and enforced by ABMAC. A combining algorithm makes a final decision according to results returned by each policy. Policy-neutrality ($\mathcal{R}$ 4.1) and flexibility ($\mathcal{R}$ 4.4) are thus improved. Decentralization ($\mathcal{R}$ 4.2) is hardly addressed since ABMAC needs a centralized control over multiple policies. Efficiency ($\mathcal{R}$ 4.3) can be hardly met because of the complexity of ABMAC authorization validation which was initially designed for grid computing.

Attribute-based Privacy Policy Privacy [6] restricts the usage of user information by avoiding disclosure without explicit consent of users. The attribute-based approach is applied for privacy. An anonymous credential system allows a user to selectively prove statements about her identity attributes while keeping the corresponding data hidden [43]. The PrimeLife framework [21] applies privacy-aware access control in order to disclose only limited information about users. It uses the XACML language and architecture [77] for privacy policy specification and access enforcement.

4.2.5 Summary

Different authorization models have been reviewed in this section. From the policy specification perspective, ACL- or capability-based systems implement the access matrix which requires to maintain a large representation of *subject-object-operation* relationships. The management of such a matrix in distributed contexts is a challenge, with an important management overhead. MLS models reduce this overhead by assigning security levels to subjects and objects to compute permissions. But in large-scale systems, the partial order between security levels may become complex. DTE groups subjects and objects to domains and types for simplification. Role-based models introduce the notion of role to

	Policy-neutral	Decentralization	Efficiency	Flexibility
Matrix (ACL/Capability)	-	+	-	-
MLS (BLP/Biba)	-	?	++	+
DTE	-	?	++	+
RBAC (SRBAC/DRBAC)	+	+	-	++
GRBAC	++	+	-	++
OrBAC	++	+	?	++
ABMAC	++	-	-	++
PrimeLife	++	-	-	++

Table 4.1: Access Control Models Comparison

separate permissions from subjects in to improve flexibility. Finally, models applying the attribute-based approach simplify subject-object relations by describing entities through their attributes. Other promising models have also been proposed such as PBAC [132] or RAdAC [100]. However, they are not examined in this section. We believe that their main contributions deal not with access control specification, but with administration models of policy specification.

As show in Table 4.2.5, each model has its benefits and drawbacks (we use the symbol "–" to represent no fulfilled requirements, "+" and "++" for partially and fully satisfied requirements, and "?" for no mentioned requirements). GRBAC and OrBAC show advantages in the policy-neutral ($\mathcal{R}$ 4.1), decentralization ($\mathcal{R}$ 4.2) and flexibility ($\mathcal{R}$ 4.4) requirements. Since the target authorization policy will be enforced at the OS level, efficiency ($\mathcal{R}$ 4.3) needs to be improved. What is lacking is an access control approach that covers all the previous design requirements.

4.3 Contribution

In this chapter, a Generic Attribute-based Access Control (G-ABAC) approach is proposed that provides a common means to express, specify and instantiate various authorization policies such as DTE, MLS, etc. Unlike existing models, our G-ABAC combines the following features:

1. Integrating the attribute-based approach in access control:

   Different concept elements like security levels in MLS or roles in RBAC can be formulated as attributes. Authorization validation uses these attributes to compute permissions. Separation of attributes from concrete entities like subjects or objects fulfills the policy-neutrality ($\mathcal{R}$ 4.1) and flexibility ($\mathcal{R}$ 4.4) requirements;

2. Separating attribute administration from access control policy definition:

   ABAC access control permissions depend on security attributes which are usually managed by a related administration model. Our G-ABAC is generic in the sense

that it is independent from and transparent to administration models. This separation supports both the policy-neutrality ($\mathcal{R}$ 4.1) and flexibility ($\mathcal{R}$ 4.4) requirements;

3. Enabling dynamic reconfiguration of authorization policies:

   G-ABAC allows modification of policies and update of security attribute values at run-time. Since a system built on G-ABAC may execute in different contexts, security adaptation can be achieved by tuning applied authorization policies, e.g., replacing one access permission by another or just modifying security attribute values. The proposed reconfiguration mechanism improves both flexibility ($\mathcal{R}$ 4.4) and efficiency ($\mathcal{R}$ 4.3);

4. Proposing policy customization based on node profiles:

   One main distinguishing feature about pervasive systems is heterogeneity: different kinds of terminals can coexist in one system. As such, the security consistency requires a master authorization policy to be enforced in all nodes. However, the diversity of nodes may imply authorization policies of different complexity. Our authorization model provides a customization mechanism to generate specific but consistent policies depending on profile of each node. This mechanism overcomes the problem of node-level authorization efficiency ($\mathcal{R}$ 4.3). Moreover, customization also improves the decentralization requirement ($\mathcal{R}$ 4.2).

Other features like integration of organization hierarchy and support of sessions are also supported by our framework, either as extensions of the main approach or as additional specific modules.

4.4 G-ABAC Definition

G-ABAC is organized into two parts: (1) a definition that describes basic G-ABAC elements and relations between these elements; (2) some administration supports for the G-ABAC elements and relations. A description of the G-ABAC basic elements is given next.

4.4.1 Entities

Entities are fundamental abstractions of G-ABAC. The four main entities are:

- Subject

  Subjects are active entities for access validation. They may represent processes in a capability-based system, users in RBAC or requestors in ABMAC. We use S to represent the set of subjects;

- Object

  Objects are passive entities. They can be resources in DTE, memory in a capability-based system or services in ABMAC. All entities on which access operations may apply may be categorized as objects. We use O to represent the set of objects;

- Action

  Actions represent access control operations on objects. Conventional access actions are read, write and execute in Linux. Other operations such as insert or remove for Database access control may also be considered as actions. A represents the set of actions;

- Context

  unlike basic access control models, emerging models apply additional information such as organizations in OrBAC, location in SRBAC/DRBAC and user information in ABMAC. One solution is to abstract all these types of information into one entity called *context*. The context model DNG-ng proposed in [139] allows the modeling of different informations in a single model. We use C to represent the set of contexts.

4.4.2 Attributes of Entities

In the attribute-based approach, each entity has one or a set of attributes that define its characteristics. Unlike the conventional definition of attributes which has only one value, our concept of attribute may hold a finite set of values. E.g., when using G-ABAC to specify RBAC policies, the *role* concept is defined as a subject attribute. Since one user can be assigned several roles, a set of role values can be assigned to the *role* attribute.

- Subject Attribute

  For each subject, a set of attributes can be associated that characterize different aspects of the subject. The attributes can be of any type such as roles in RBAC, domains in DTE, or security clearances in MLS.

  We define the set of subject attribute values as: $ATT_S = ATT_S_1 \times ATT_S_2 \times ...ATT_S_i... \times ATT_S_l$ where ATT_S_i represents a finite set values of the subject attribute s_i and l is the number of subject attributes. The function $att_s : S \rightarrow 2^{ATT_S}$ returns attribute values for each subject;

- Object Attribute

  An object can also hold one or a set of object attributes to describe its specific characteristics. The attributes can be types in DTE, security classifications in MLS or views in OrBAC. An object attribute may have a finite set of values.

  We define the set of object attribute values as: $ATT_O = ATT_O_1 \times ATT_O_2 \times ...ATT_O_i... \times ATT_O_m$ where ATT_O_i represents a finite set of values for one object attribute and m is the number of object attributes. The function $att_o : O \rightarrow 2^{ATT_O}$ returns attribute values for each object;

- Action Attribute

  An action attribute specifies categories of access actions that can be performed. For example, OS access operations as read, write, and execute can be regrouped into

one action attribute. In cloud computing, access actions are categorized into two attributes: Virtual Machine actions which access virtualized resources and hypervisor actions to directly access physical resources.

We define the set of action attribute values as: $ATT_A = ATT_A_1 \times ATT_A_2 \times ...ATT_A_i... \times ATT_A_n$ where ATT_A_i represents a finite set of values for one action attribute and n is the number of action attributes. The function $att_a : A \rightarrow 2^{ATT_A}$ returns attribute values for each action;

- Context Attribute

  Context attributes describe elements of the context of an entity. For example, location may be viewed as a context attribute, and can be assigned with multiple values such as city, GPS position, etc.

  Formally, we define the set of context attribute values as: $ATT_C = ATT_C_1 \times ATT_C_2 \times ...ATT_C_i... \times ATT_C_p$ where ATT_C_i represents a set of values for one context attribute and p is the number of context attributes. The function $att_c : C \rightarrow 2^{ATT_C}$ returns attribute values for each context.

4.4.3 Rules

A key difference between G-ABAC and existing ABAC models is the abstraction of rules in G-ABAC. Access permissions based on subject, object, action or context attribute values are formulated as rules which are administered as independent entities. The rules of G-ABAC are static since they do not evolve during execution.

In OrBAC, some context-aware constraints can be defined to (de)activate permissions according to context evolutions. This allows adaptation of permissions for different environments. But from the perspective of policy administration, this mechanism asks for administration models to support run-time update for permissions. In order to fulfill the policy-neutrality requirement, G-ABAC should be expressive to specify different policies and be flexible to support associated administration models of policy specification. The *rule* concept improves flexibility. For context evolutions, attribute value may be updated. Therefore, another rule will be selected based on new values. But the rules themselves always remain unchanged.

Efficiency is another important motivation for the rule concept. For OS-level access control, the complex structure of an access control policy is a main challenge. Low-level authorization modules cannot enforce complex access control policies without performance loss. For example, in SELinux [104], only three simple policies DTE, MLS, RBAC (simplified version) are implemented. The static and simplified structure of rules reduces the overhead of authorization administration.

Unfortunately, G-ABAC is limited in terms of performance for some access control models. The rule concept simplifies administration, but on the other hand it may increase the overhead of access validation. With the static rule concept, all rules based on attribute values are fixed. When the system state or execution context evolves, updated values may be assigned to entities. Access validations may thus check all rules to find out whether its attribute values are associated to another permission. In the case where attributes hold

many values, a large set of rules exist. Hence, a huge rule table should be maintained and used. This induces a significant access validation overhead which will be evaluated in Chapter 8.

4.4.4 G-ABAC Overview

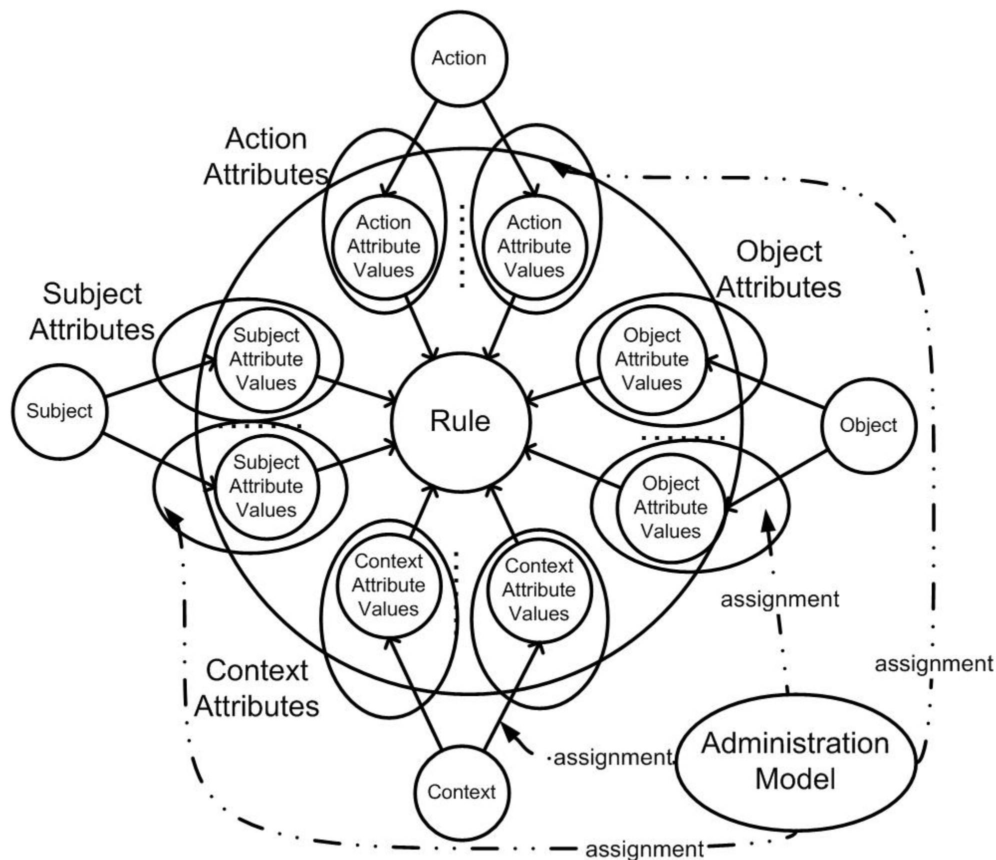

Figure 4.1: G-ABAC Approach

G-ABAC is defined as entity-attribute-rule relation shown in Figure 4.1. Each entity (subject, object, action, or context) can hold several attributes. Each attribute can be assigned with a finite set of values. We define the rules as static relations between attribute values. An administration model controls assignment of values to attributes. One advantage of the separation of attribute usage from its administration is the improvement to support several administration models. Some powerful administration models with complex and dynamic concepts (like constraint or sessions) can be easily embodied to administer G-ABAC policies. For example, in RBAC, the session concept controls the (de)activation of roles to users. To specify RBAC using G-ABAC, sessions may be represented by adding or removing values to the role attribute.

4.4.5 G-ABAC Policy Definition

Definition 4.1 *A G-ABAC policy is defined as:*

- S, O, A, *and* C*: the sets of subjects, objects, actions, and contexts;*
- ATT_S, ATT_O, ATT_A, *and* ATT_C*: the sets of subject attributes, object attributes, action attributes, and context attributes;*
- $RULES : ATT_S \times ATT_O \times ATT_A \times ATT_C$*, a permission rules table based on values of subject attributes, object attributes, action attributes, and context attributes;*
- $grant(s, o, a, c) \Leftrightarrow \exists sa \in att_s(s), \exists oa \in att_o(o), \exists aa \in att_a, \exists ca \in att_c(c), (sa, oa, aa, ca) \in RULES$*: an access request (s,o,a,c) can be granted for the subject s with the action a on the object o in the context c, if and only if there exists a permission rule for the corresponding subject attribute values sa, object attribute values oa, action attribute values aa and context attribute values ca.*

4.4.6 Specification of Existing Policies using G-ABAC

Some examples are given in order to show the expressivity of G-ABAC to describe access control policies, ranging from the simple ACL policy, to the MLS policy and the OrBAC policy.

ACL Policies: An ACL policy is defined as:

- A set of subject identities: SID;
- A set of object identities: OID;
- A set of access operations: AO;
- An assignment of subject identities to subjects: $sid : S \rightarrow SID$;
- An assignment of object identities to objects: $oid : O \rightarrow OID$;
- An assignment of access operations to actions: $ao : A \rightarrow AO$;
- For each object, an Access Control List is defined: $acl : OID \rightarrow 2^{SID \times AO}$;

An access request (s,o,a) with $s \in S$, $o \in O$, $a \in A$ can be granted if and only if $(sid(s), ao(a)) \in acl(oid(o))$.

We translate the ACL policy in G-ABAC by:

- Defining subject identities as subject attributes: $ATT_S = SID$;
- Defining object identities as object attributes: $ATT_O = OID$;
- Defining access operations as action attributes: $ATT_A = AO$;

- Defining sid assignment as subject attribute assignment: $att_s = sid$;
- Defining oid assignment as object attribute assignment: $att_o = oid$;
- Defining ao assignment as action attribute assignment: $att_a = ao$;
- Neglecting the context attributes: $ATT_C = \emptyset$;
- Defining a set of rules $RULES = \{(sid, oid, ao, *)|sid \in SID, oid \in OID, ao \in AO, (sid, ao) \in acl(odi)\}$.

Therefore, an access request (s,o,a) with $s \in S$, $o \in O$, $a \in A$ can be granted if and only if $(att_s(s), att_o(o), att_a(a), *) \in RULES$.

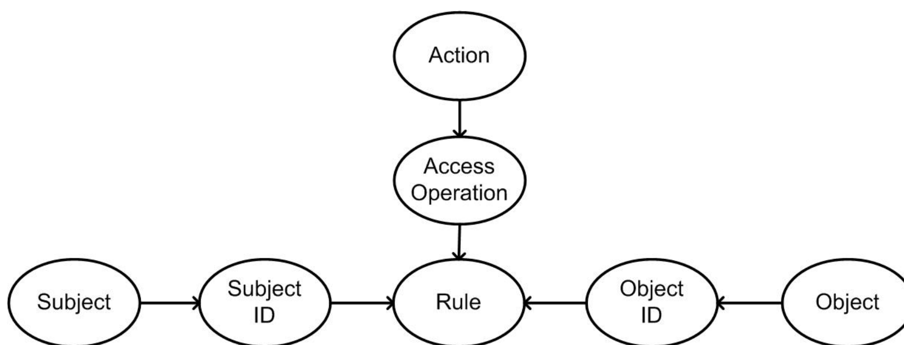

Figure 4.2: Access Control List Policies in G-ABAC

In the case of ACLs, individual identities are represented as subject and object attributes. Permission rules are defined based on subject identities, object identities, and access operations as shown in Figure 4.2. If the identity of a subject together with a requested object identity and an operation exists in the rule list, there is thus a permission which grants the access request.

MLS Policies: This example shows the implementation of a Bell LaPadula (BLP) policy using G-ABAC. A BLP [27] policy is defined as:

- A set of security clearances $SCLEAR$;
- A set of security classifications $SCLASS$;
- A set of access operation $AO = \{read, write\}$;
- A dominance relation $\geq$ between security clearances and security classifications;
- An assignment of security clearances to subjects: $clear : S \rightarrow SCLEAR$;
- An assignment of security classifications to objects: $class : O \rightarrow SCLASS$;
- An assignment of access operations to actions: $ao : A \rightarrow AO$.

A read access request (s,o,read) with $s \in S$, $o \in O$ can be granted if and only if $clear(s) \geq class(o)$. A write access request (s,o,write) with $s \in S$, $o \in O$ can be granted if and only if $class(o) \geq clear(s)$.

We translate the BLP policy in G-ABAC by:

- Defining security clearances as subject attributes: $ATT_S = SCLEAR$;
- Defining security classifications as object attributes: $ATT_O = SCLASS$;
- Defining access operations as action attributes: $ATT_A = AO$;
- Neglecting context attributes: $ATT_C = \emptyset$;
- Defining the *clear* assignment as the subject attribute assignment: $att_s = clear$;
- Defining the *class* assignment as the object attribute assignment: $att_o = class$;
- Defining the *ao* assignment as the action attribute assignment: $att_a = ao$;
- Defining a set of rules $RULES = \{(sClear, sClass, ao, *) | sClear \in SCLEAR, sClass \in SCLASS, ao \in AO, (sClear \geq sClass, ao = read)$ or $(sClass \geq sClear, ao = write)\}$.

An access request (s,o,a) with $s \in S$, $o \in O$, $a \in A$ can be granted if and only if $(att_s(s), att_o(o), att_a(a), *) \in RULES$.

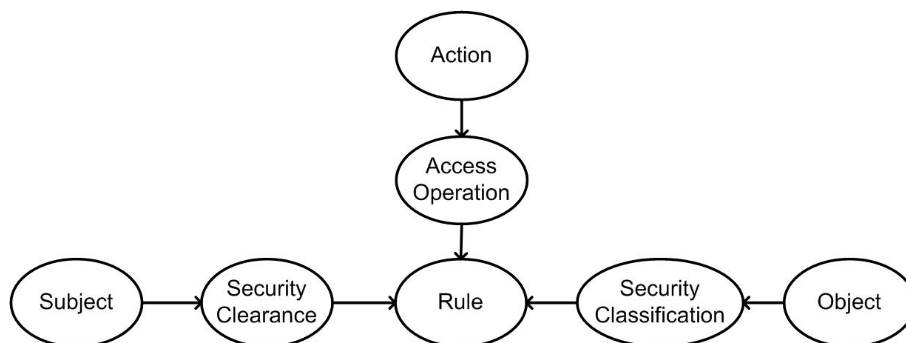

Figure 4.3: Multiple Security Level Policies in G-ABAC

To specify MLS-BLP policies with G-ABAC, security clearances and classifications are defined as attributes for both subjects and objects. Two access operations read and write are specified as the access operations for the action attributes. The rules are based on the values of subject attributes, object attributes and action attributes (see Figure 4.3). Permissions are computed by comparing the domination relationship. If the security clearance of a subject dominates the security classification of an object, a read permission rule is identified. If the security classification dominates the security clearance, a write permission rule is identified.

OrBAC Policies: An OrBAC policy is defined by:

- A set of organizations: ORG;
- A set of roles: R;
- A set of views: V;
- A set of activities: AT;
- A set of contexts: C;
- A *Employ* relationship: $employ : S \times ORG \rightarrow R$;
- A *Use* relationship: $use : O \times ORG \rightarrow V$;
- A *Consider* relationship: $consider : A \times ORG \rightarrow AT$;
- A *Hold* relationship: $hold : S \times A \times O \times ORG \rightarrow C$;
- A *Permissions* relationship: $permission : ORG \times R \times V \times AT \times C \rightarrow grant$.

An access request (org,s,o,a) with $org \in ORG$, $s \in S$, $o \in O$, $a \in A$ can be granted if and only if $permission(org, employ(s, org), use(o, org), consider(a, org), hold(s, a, o, org)) = \{grant\}$.

G-ABAC cannot translate an OrBAC policy to an equivalent policy in using the attribute-based approach. However, we propose in this paragraph a transformation to specify an "OrBAC-like policy".

- Defining roles as subject attributes: $ATT_S = R$;
- Defining views as object attributes: $ATT_O = V$;
- Defining activities as action attributes: $ATT_A = AT$;
- Defining the organization and context as the two dimensions of context attributes: $ATT_C = ORG \times CX$. In order to distinguish different context definitions in OrBAC and G-ABAC, in later part, we use the term "context (the symbol C)" to represent that of OrBAC, and "context entity (the symbol CX)" to represent that of G-ABAC. By G-ABAC, the organization and context of OrBAC are considered as two dimensions of the context entity;
- Defining the subject attribute assignment $att_s : S \rightarrow 2^R$, a function which returns all potential roles of the subject for different organizations;
- Defining the object attribute assignment $att_o : O \rightarrow 2^V$, a function which returns all potential views of the object for different organizations;
- Defining the action attribute assignment $att_a : A \rightarrow 2^{AT}$, a function which returns all potential activities of the action for different organizations;

- Defining one context entity attribute assignment $att_c_1 : C \rightarrow ORG$, a function which returns one organization;
- Defining another context entity attribute assignment $att_c_2 : C \rightarrow CX$, a function which returns one context information;
- Defining a set of rules: $RULES = \{(r, v, at, org, cx) | r \in R, v \in V, at \in AT, org \in ORG, cx \in CX, permission(org, r, v, at, cx) = grant\}$.

An access request (s,o,a,c) with $org \in ORG$, $s \in S$, $o \in O$, $a \in A$, $c \in C$ can be granted if and only if $\exists r \in att_s(s), \exists v \in att_o(o), \exists at \in att_a(a), (r, v, at, att_c_1(c), att_c_2(c)) \in RULES$.

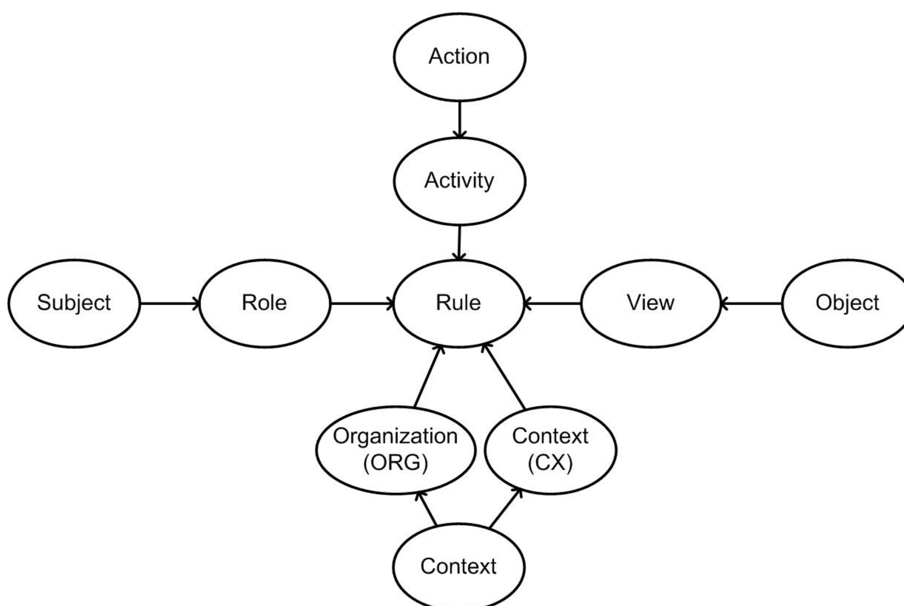

Figure 4.4: Organization-Based Access Control Policies in G-ABAC

To describe OrBAC policies with G-ABAC, the role attribute is assigned to subjects, views to objects, activities to actions, and the organization and context to the context entities of G-ABAC (see Figure 4.4). For an access request (s, o, a, c), if its role, together with its view and activity, context entity (including organization and context information) exists in the rule table *RULES*, the access request can be granted. In the opposite case, it is denied.

Unlike the definition of OrBAC [89], the organization concept is modeled as a context attribute in the transformation. In fact, the relationships of OrBAC such as *organization-subject-employ-role*, *organization-object-use-view*, *organization-action-consider-activity*, and *organization-subject-action-object-hold-context* are simplified to *subject-role*, *object-view*, *action-activity*, and *context-context entity*. Instead, the organization concept is defined as another dimension of the context entity. This transformation simplifies the *entity-attribute* assignments. But on the other hand, the rules becomes somewhat complicated.

OrBAC introduces not only permissions, but also prohibition, obligation, and recommendation concepts. Since permission and prohibition are similar in terms of policy specification, we give only an example of permission in this sub-section. Obligation and recommendation call for supplementary life-cycle management which cannot be realized in our framework.

4.4.7 G-ABAC Specification

4.4.7.1 G-ABAC Specification Model

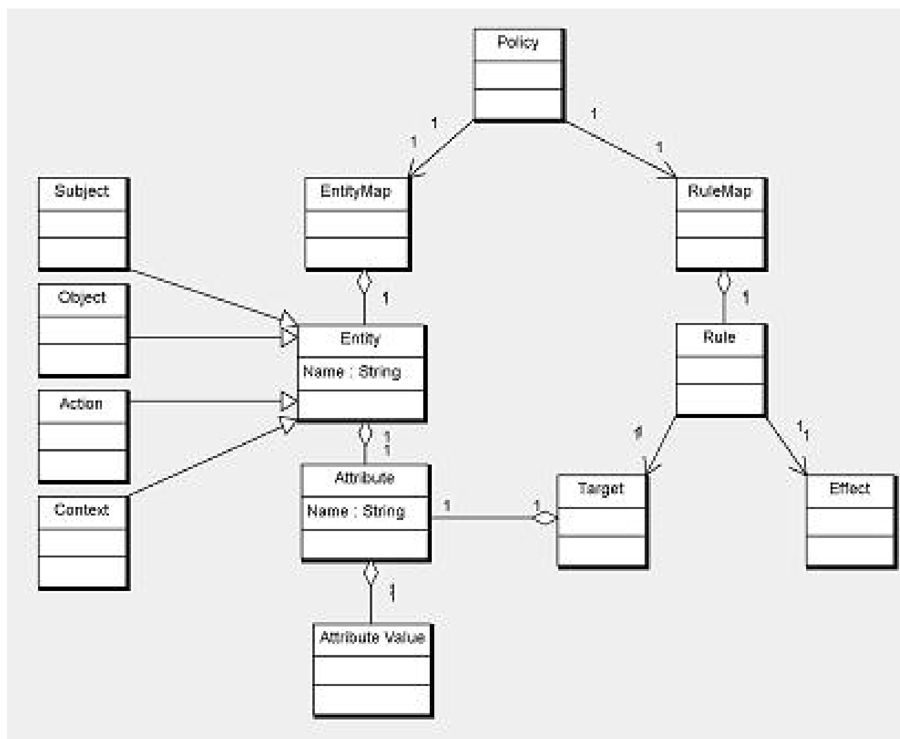

Figure 4.5: G-ABAC Specification Model

The specification of a G-ABAC policy consists of an *entity map* and a *rule map* (see Figure 4.5). The *entity map* is a set of entities that couple system basic entities to their attributes. Entities may be subjects, objects, actions, or contexts. Examples of corresponding attributes include security domains, resource types, read/write operations, localization and time information. A rule contains a target described by several attribute values and an effect (grant or deny).

4.4.7.2 Expression Format

With a large number of frameworks or applications that either use XML to model their data or to present data relations or restrictions, specification of authorization policies in

XML becomes an important requirement. A sample DTE policy based on on our policy model is shown below, granting write authorizations to a private file ".bashrc" for process "bash" which is in a trusted domain of Linux.

```
<?xml version="1.0"?>
<DTEPolicy>
  <EntityMap>
    <Subject Name="bash">
      <Attribute Name="domain">
          <Value>Trusted</Value>
        </Attribute>
    </Subject>
    <Object Name=".bashrc">
        <Attribute Name="type" >
          <Value>Private</Value>
        </Attribute>
    </Object>
    <Action Name="OSAccess">
        <Attribute Name="operation">
          <Value>write</Value>
        </Attribute>
    </Action>
  </EntityMap>
  <RuleMap>
    <Rule>
      <Target>
        <Attribute name="domain">
            <Value>Trusted</Value>
          </Attribute>
        <Attribute name="type">
            <Value>Private</Value>
          </Attribute>
        <Attribute name="operation">
            <Value>write</Value>
          </Attribute>
      </Target>
      <Effect>grant</Effect>
    </Rule>
  </RuleMap>
</DTEPolicy>
```

The result is a quite expressive formalization, while still remaining policy-neutral. As in XACML [77], specific authorization policies may be implemented by defining corresponding attributes and rules. For instance, RBAC policies are implemented by defining some profiles [19]. Similarly, we think that context-aware or history-based policies may be implemented by adding specific context attributes, but we do not investigate on it.

4.5 Administration Support

An administration model usually allows to specify policy management strategies. Different administration models can guide execution to various usages. This section presents some administration supports with which various administration models can be implemented. More details about different administration models of G-ABAC will be given in the next chapter.

4.5.1 Attribute Mutability and Revocation

An entity in G-ABAC is assigned with attributes which are properties that will be used for authorization. Attribute mutability is an important issue for G-ABAC where an attribute value is modifiable during the system execution. Unlike UCON [116] which supports mutable attributes as a consequence of access, G-ABAC together with revocation mechanism of the underlying VSK OS (see Chapter 6) enable attribute mutability at any phase of authorization. An attribute of any type (subject, object, action, or context) may be modified by a administration model before, during, or after access validation. Following access decisions will then be based on the updated attributes values.

Thanks to the separation of attribute usage from attribute administration, permission rules remains constant while attribute assignments are changed. Another rule related to the updated attribute values will thus be selected to handle with new access requests. Underlying system (Chapter 6) proposes some solutions to implement this feature. As the result of attribute mutability, attribute update can be realized during the whole life-cycle of system, not just before an authorization, but also during and after authorization.

4.5.2 Policy Customization

Policy customization enables to refine a master authorization policy into several decentralized but consistent policies for distributed nodes. In our approach, a node profile is specified that characterizes the settings of each node. With the profile information, a global policy can be automatically translated into local ones that are suitable for nodes. For instance, a node profile in our implementation is represented as:

```
<?xml version = "1.0" ?>
<NodeProfile>
  <Platform reference="MSP430F2003">
    <CPUType>16bit</CPUType>
    <CPUSpeed>16MHz</CPUSpeed>
    <Flash>1kb</Flash>
    <RAM>128b</RAM>
  </Platform>
  <Category name="sensor">
    <Subject>ZigBeeDriver</Subject>
    <Subject>UWBDriver</Subject>
    <Object>ProtocolStack</Object>
    <Object>UserPrivateFile</Object>
  </Category>
</NodeProfile>
```

It consists of two parts: a platform part that describes hardware properties such as CPU type, CPU speed, Flash size, and RAM size; and a software part which specifies its running applications like communication protocol, user files, etc. With this profile, a lightweight but effective local authorization policy can be generated based specific characteristics of a node.

4.6 Summary

This chapter described G-ABAC, a generic attribute-based access control approach, which permits to specify a wide set of existing authorization policies. In the context of pervasive systems, policy-neutrality, decentralization, efficiency, and flexibility are listed as the key design requirements. By applying the attribute-based approach and separating attribute administration from policy specification, G-ABAC becomes more flexible for both policy specification and administration. It enables dynamic reconfiguration and customization of policies in order to overcome efficiency and decentralization. Therefore, all the requirements are fulfilled with G-ABAC. This chapter also presented a definition of G-ABAC, illustrating how to implement several existing authorization polices.

However, the proposed authorization approach G-ABAC does not cope with the deployment for distributed systems. An Autonomic Security Policy Framework (ASPF) which deploy G-ABAC policies through the whole pervasive system will be introduced in the next chapter.

Chapter 5

Autonomic Security Policy Framework

The previous chapter permits to use a generic access control approach, G-ABAC, which allows to express existing access control policies. However, it lacks some deployment supports for G-ABAC policies. In this chapter, a policy deployment framework, called Autonomic Security Policy Framework (ASPF), is proposed which not only achieves deployment of different access control policies in distributed environments, but also embodies autonomic control loops to improve self-management capabilities. Based on this framework, different *administration models of policy specification* can be implemented.

Policy-based approach was originally developed to reduce the complexity of IT system administration, and the use of authorization policy guides protection behaviors of managed systems. Therefore, the integration of these two approaches improves administration and deployment of authorization policies and guarantees their consistency. With the help of autonomic computing, a policy-based framework becomes both user-friendly without human intervention and context-aware which can react to the context evolution.

Section 5.1 presents key design requirements for such an authorization policy management framework. This framework will support authorization policies specified by G-ABAC. It must be scalable to large-size by guaranteeing policy consistency. The framework needs to be more user-friendly for administration and can react quickly and efficiently to the context evolution.

Section 5.2 gives a short survey of existing access control administration models with different management solutions. With the policy-based approach, we examine existing frameworks. A summary that reviews all these two approaches in following the design requirements is given at the end of this section.

Section 5.3 shows main features of ASPF. ASPF applies the policy-based approach to authorization policy administration and deployment. It decentralizes access control validation to improve scalability. By integrating self-configuration and self-protection mechanisms into the framework, ASPF reduces human intervention and facilitates administration and deployment of authorization policies.

Section 5.4 describes three main models of ASPF: a core model, an extended model, and

an implementation model. It shows the extension of the core model to the extended model to describe functionalities of each type of resources. The refinement of the extended model to the implementation model organizes functionalities as implementation components. Finally, the realization of the autonomic vision is illustrated through these three models.

Section 5.5 compares the ASPF framework to the conventional access control architecture XACML, describes use of G-ABAC, and gives us a detailed description about complementary autonomic functionalities.

Section 5.6 discusses in detail some supplementary mechanisms like execution support, realization of delegation approach, and reconfiguration mechanism.

Section 5.7 concludes the chapter and indicates some requirements of the underlying OS that will be fulfilled in Chapter 6.

5.1 ASPF Design Requirements

5.1.1 G-ABAC Policy Support

Different types of authorization policies may be enforced in clusters and nodes. *Policy-neutrality* is thus mandatory to account for heterogeneous security domains by supporting several classes of authorization policies. Moreover, policies should be *dynamically reconfigured* (between different classes) when nodes move between security domains, or when the context changes. G-ABAC is proposed in the previous chapter that can express a wide set of existing authorization policies. Instead of developing a policy-neutral architecture being able to enforce various policies, a framework supporting G-ABAC can implement different access control policies expressed by G-ABAC. Hence, the policy-neutrality and flexibility requirements of ASPF can be replaced by the requirement to apply G-ABAC.

Requirement $\mathcal{R}$ 5.1 *As a policy-based framework, ASPF will support and administer policies expressed by G-ABAC.*

5.1.2 Scalability

Pervasive systems are highly open and dynamic: nodes can enter and leave a network at run-time. The numbers of connected nodes may thus vary greatly in time, scaling network capacity both up and down, while the infrastructure remains unchanged. Scalability is thus a major challenge for the underlying protection framework, which should support both small- and large-scale systems.

Requirement $\mathcal{R}$ 5.2 *ASPF should enable dynamical scaling within the same infrastructure.*

5.1.3 Consistency

At the device level, a single system component (e.g., the security kernel [17]) usually controls all access to resources and enforces authorization policies. However, at the network level, each node still applies its own policy, but some nodes may share resources. The

lack of a centralized module for enforcement of authorizations may lead to inconsistent network security policies. A solution for policy administration is thus required to guarantee consistency of distributed authorization policies.

Requirement $\mathcal{R}$ 5.3 *ASPF should enforce decentralized and distributed policies to guarantee consistency.*

5.1.4 User-Friendliness

Pervasive systems become increasingly complex, involving multiple users with different roles. Thus, the issue of system administration with minimal human intervention cannot be ignored. A security policy management framework should therefore simplify administration tasks and make system modifications transparent to other users.

Requirement $\mathcal{R}$ 5.4 *ASPF should be user-friendly for system administration and make modifications transparent to other users.*

5.1.5 Context-awareness

Openness and dynamism of pervasive networks induce rapid changes in the system context, calling for context-aware administration and protection. For instance, node availability may affect access privileges, as in ASRBAC authorization policies [15]. A node part of some clusters may have specific types of permissions that cannot be assigned to nodes in other clusters. Node migration between clusters may thus require update of access privileges. The management framework should thus select security functions based on evolution of the context.

Requirement $\mathcal{R}$ 5.5 *ASPF should be able to dynamically reconfigure its components according to the context evolution.*

5.1.6 Other Requirements

The security framework should also take into account requirements such as unified modeling of heterogeneous nodes, efficient protection mechanisms compatible with embedded constraints, or collaboration of decentralized security infrastructures.

5.2 A Short Survey of Policy Administration Mechanisms

Since the main objective of the framework is to develop a policy management framework which can support most of existing access control policies together with their associated administration models, an overview about access control administration models is presented in Section 5.2.1. Additionally, some existing policy-based frameworks are discussed in Section 5.2.2.

5.2.1 Access Control Administration Models

Early access control models does not really define a concrete administration model. Bell LaPadula [27] is a Mandatary Access Control (MAC) which uses security clearance and classification for authorization validation. But the assignment and modification of security levels (clearance or classification) in MAC are usually controlled by a central authority which enhances consistency ($\mathcal{R}$ 5.3). For large-scale, distributed and dynamic systems, the central authority does not seem adequate.

Capability-based system is of Discretionary Access Control (DAC) which delegates the right of authorization permission assignment to owners of objects. An owner of an objects can determine authorization privileges and is able to delegate his privileges to other users. This paradigm improves scalability ($\mathcal{R}$ 5.2). In contrast, the multiple ownership which is distributed through the whole system may provoke inconsistency of authorization policies.

ARBAC97 [126] is a popular administration model corresponding to RBAC96 [127]. It consists of three main components as URA97 for user-role assignment, PRA97 for permission-role assignment, and RRA97 for role-role assignment which can be recognized by the management of attributes for authorization policies ($\mathcal{R}$ 5.1).

As an extension of ARBAC97, Scoped Administration of Role-Based Access Control (SARBAC) [56] proposes administration scope as paradigm to manage the assignments of users and permissions to roles of RBAC. The administration scope [55] tags each role in the role hierarchy with a set of roles over which it has control. Hence, the control over complex role hierarchy can be reduced to the management of scoped administration ($\mathcal{R}$ 5.2).

As an improved version of ARBAC97, ARBAC02 [113] overcomes shortcomings such as multiple assignment, redundant assignment, restricted constraints. ARBAC02 uses an organization structure for user-permission pools, rather than roles as user-permission pools in ARBAC97 which enables the separation of user-permission pools to roles for the improvement of scalability ($\mathcal{R}$ 5.2). ARBAC02 also shows its advantages for supporting the administration of ACL-based policies or MLS policies which improves policy-neutrality ($\mathcal{R}$ 5.1) and efficiency.

However, ARBAC02 depends on the running infrastructure and lacks flexibility and extensibility. ASRBAC [15] is designed by combining SARBAC and ARBAC02 in an infrastructureless network. It proposes role categories as the concept of scope in SARBAC, and assigns these categories to entities of network ($\mathcal{R}$ 5.2). Another advent of ASRBAC is context-awareness by taking into account circumstance information for the administration of role assignment ($\mathcal{R}$ 5.5).

Because of the abstraction of subject, action, object respectively by role, activity and view, the administration model of OrBAC [89] should administer their assignments. AdOrBAC [58] achieves administration by providing URA for User-Role Administration, PRA for Permission-Role Administration, and UPA for User-Permission Administration ($\mathcal{R}$ 5.1). Scalability ($\mathcal{R}$ 5.2) and context-awareness ($\mathcal{R}$ 5.5) [59] are also taken into account in AdOrBAC.

Within a distributed, complex, and large-scale environment, decentralized management of authorization policies is missing. Policy-Based Access Control (PBAC) [35] proposes

a solution to harmonize Attribute-Based Access Control ($\mathcal{R}$ 5.1) in a distributed manner and provides coherent ($\mathcal{R}$ 5.3) and distributed access control policies ($\mathcal{R}$ 5.2). Rather than the previous administration models, PBAC holds a global control over these distributed policies.

Another access control model, RAdAC [112], advances one step by introducing environmental condition such as risk level into authorization process. Subject, object and permission assignments may be updated according to the executing circumstances. For example, a strong access control policy could be related upon a non-secure context. Context-awareness enables self-management of administration model ($\mathcal{R}$ 5.5).

5.2.2 Policy-based Frameworks

The policy-based approach uses highly abstracted policies to control behavior of executing systems, based on a framework to administer these policies. We believe that this kind of frameworks can be employed to authorization policy management. In this sub-section, we examine exiting policy-based frameworks in following our design requirements.

The IETF policy framework [152, 108] is widely accepted and consists of three main components: a *Policy Enforcement Point (PEP)*, a *Policy Decision Point (PDP)*, and a *policy repository*. PEP is responsible for enforcement of policy by executing necessary actions. PDP produces policy decisions and the *policy repository* is the component in which resists defined policies. Within this framework, centralized PDP is the main component for decision making which improves the consistency requirement ($\mathcal{R}$ 5.3).

The IETF framework was extended in the XACML [77] framework by additional *Policy Information Point (PIP)* and *Policy Administration Point (PAP)*. Access validation is based on information queried from PIP ($\mathcal{R}$ 5.1). PAP not only specifies policies but also manages them ($\mathcal{R}$ 5.3). It was implemented in the *Globus Toolkit release 4 (GT4)* [98]. The main difference of GT4 from IETF is its flexibility which contains a set of policies that may be suitable for different circumstances ($\mathcal{R}$ 5.5).

Policy Management for Autonomic Computing (PMAC) [13] is a policy management platform which uses *Autonomic Manger (AM)* to govern a set of *Managed Resources (MR)*. It reads states of MR using some sensor interfaces, evaluates relevant policies, and plans actions ($\mathcal{R}$ 5.4, $\mathcal{R}$ 5.5). When a MR receives directives from an AM, it can change its behavior in order to achieve policy guidance. For scalability ($\mathcal{R}$ 5.2), large-scale IT systems may be divided into different domains over which one AM manages one domain. Consistency of distributed policies ($\mathcal{R}$ 5.3) is also treated through a policy ratification mechanism.

A similar Policy-Based Architecture for Autonomic Communication (PBAAC) is proposed for autonomic communication [61]. A Shared *Information and Data Model* enables abstraction of both desired behavior and deduced current behavior. In addition, a comparison of these two models is established to develop a representation of behavioral orchestration ($\mathcal{R}$ 5.4, $\mathcal{R}$ 5.5). Different from conventional policy-based frameworks, PBAAC enables policy refinement by refining or decomposing high-level business policies to low-level network element policies ($\mathcal{R}$ 5.2). A policy conflict module is in charge of consistency of distributed policies ($\mathcal{R}$ 5.3).

		G-ABAC Support	Scalability	Consistency	User -friendlity	Context -awareness
Administration Models	MAC	-	-	+	-	-
	DAC	-	+	-	-	-
	ARBAC97	+	-	-	-	-
	SARBAC	+	+	-	-	-
	ARBAC02	+	+	-	-	-
	ASRBAC	+	+	-	-	+
	AdOrBAC	+	+	-	-	+
	PBAC	+	+	++	-	+
	RAdAC	-	-	-	-	++
Policy-based Frameworks	IETF	-	-	+	-	-
	XACML GT4	+	-	+	-	+
	PMAC	-	+	+	++	++
	PBAAC	-	+	+	++	++
	Ponder2	+	+	++	++	+

Table 5.1: Policy-Based Frameworks Comparison

Based on the security description language Ponder [60], an autonomic policy management framework for pervasive systems called Ponder2 [142] was proposed. With the Ponder language, existing authorization policies may be structured to reflect organizational structure ($\mathcal{R}$ 5.1, $\mathcal{R}$ 5.3). These policies are used in *Self-Managed Cells (SMCs)* to manage administrative domains of distributed pervasive systems ($\mathcal{R}$ 5.2). SMC also realizes an autonomic control loop ($\mathcal{R}$ 5.4) and enables context-aware interaction ($\mathcal{R}$ 5.5).

5.2.3 Summary

As shown in Table 5.1, an overview about existing access control administration models as well as existing policy-based frameworks is given. As the limit of administration models, all the five requirements are hardly treated by one existing administration model. On the other hand, the policy-based frameworks like PMAC or Ponder2 address policy management in fulfilling most of the design requirements. But the integration of authorization policies, especially G-ABAC based policies, is missing.

In conclusion, the diversity of authorization policies leads to the diversity of administration models of policy specification. Most of these administration models mainly treat creation, deletion, assignment of security attributes as described in DAC, MAC, ARBAC97, ARBAC02. Other conceptions like organizational hierarchy and context-awareness are also integrated into administration models as described in SARBAC and ASRBAC. Therefore, a policy management framework which is able to implement various authorization policies must be able to integrate all administration models associated. With the attribute-based G-ABAC approach, we believe that administration models should only handle with attribute assignment. Instead, the policy management framework takes charge of management and integration of supplementary modules like organizational hierarchy or context-awareness for access validation. As such, the administration of authorization policies is realized through updates of attribute values in taking into account evolution of supplemen-

tary concepts. For example, in order to achieve context-awareness with the BLP policy, ASPF offers the possibility to dynamically update security level based on context evolution. An administration model may check manipulations of security levels, and ASPF makes the administration model collaborate with a context monitoring system. However, the realization of such checking mechanisms refers to the restriction of information flows [42] or declassification [36] which is complex. Since we focus on the feasibility of self-protection framework, we do not cope with these checking mechanisms in this thesis.

The separation of the administration model of policy specification from the policy deployment guarantees both flexibility of administration models and efficiency for the underlying system. The objective of ASPF is to apply the policy-based approach into the administration and deployment of authorization policies based on G-ABAC.

5.3 ASPF Overview

Administration of authorization policies includes creation, deletion, and maintenance of access attributes and rules, and management of run-time constraints. To achieve this goal, ASPF applies the *autonomic approach* to make systems self-managed. Moreover, ASPF is *policy-driven*, i.e., security behavior of systems is entirely governed by policies. Main distinguishing features of the framework are the following:

1. Policy-based management of authorization:

   The policy-based approach is well adapted for administration of systems in open and dynamic environments: evolutions only trigger updates of applied policies, without changing enforcement mechanisms. In our case, we use authorization policies to control protection ($\mathcal{R}$ 5.1). ASPF enables to modify, deploy, and enforce them throughout the whole system ($\mathcal{R}$ 5.3).

2. Decentralized validation of authorizations:

   The scalability requirement ($\mathcal{R}$ 5.2) asks for a distributed infrastructure. A scalable distributed system avoids using a central authority to validate authorization. Our framework is based on a hybrid architecture using the concepts of *cluster* and *node*. Each node enforces a local authorization policy. Authorization policies of nodes inside a cluster are centrally controlled by a *cluster authority* which guarantees policy consistency between nodes. Policy synchronization between cluster authorities may be either centralized or decentralized ($\mathcal{R}$ 5.3). This architecture allows decentralized enforcement of authorization policies, while maintaining an efficient central control of policy deployment ($\mathcal{R}$ 5.2).

3. Integration of self-protection control loops:

   To satisfy the context-awareness requirement ($\mathcal{R}$ 5.5), ASPF regulates security using several self-protection feedback loops to select the authorization policy best fitting the system security context.

4. Integration of self-configuration control loops for policy deployment:

   To guarantee consistency of decentralized policies, and facilitate system administration, self-configuration control loops allow a system to configure itself with minimal human intervention. Modification of chosen authorization policies will thus be automatically propagated through the whole network ($\mathcal{R}$ 5.4) to guarantee consistent policy deployment ($\mathcal{R}$ 5.3).

Other contributions like extension of XACML authorization framework, usage of service-oriented approach, and implementation and support of governance policies are also achieved in ASPF.

5.4 ASPF Design

The policy-based framework applies policies to determine behavior of managed systems. The end-to-end self-protection framework involves two policy-based frameworks: ASPF is a policy-based system governed by an adaptation policy (see Chapter 7)to administer G-ABAC policies; underlying VSK is another policy-based system governed by G-ABAC authorization policies to control resource access. ASPF is thus a security management framework that governs authorization policies enforced by underlying VSK mechanisms. By the UML class diagram, this section describes the ASPF framework in an informal manner.

5.4.1 Overall Design

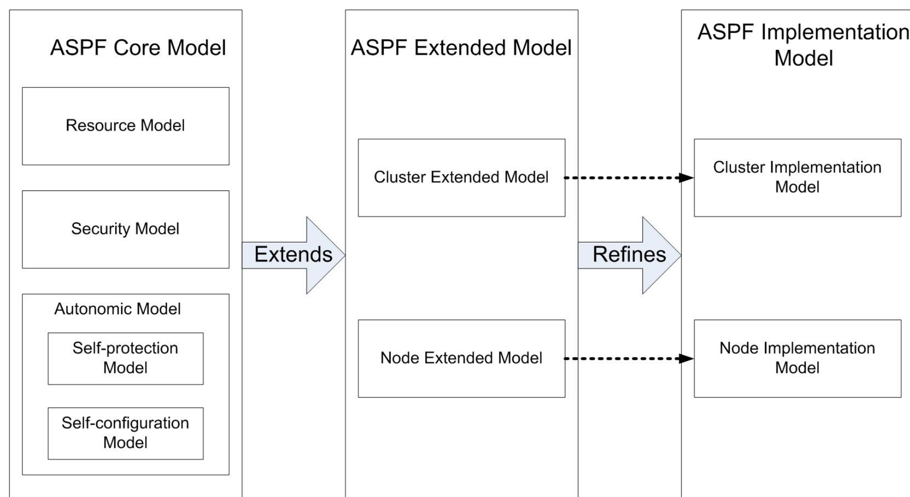

Figure 5.1: The ASPF Overall Design.

The ASPF design is organized into three models:

- A *core model* describes system resources, security, and autonomic functionalities.

- An *extended model* refines the security and autonomic models for each type of resource.

- An *implementation model* describes the realization of the extended model, organizing functionalities into components to be implemented.

Those models are defined in the three steps shown in Figure 5.1. The core model consists of a *resource model*, a *security model* and an *autonomic model*. These models are then refined into the *extended model* which involves a *cluster extended model* and a *node extended model* for cluster and node resources. Finally, these two models are refined into the corresponding implementation models.

5.4.2 ASPF Core Model

5.4.2.1 Resource Model

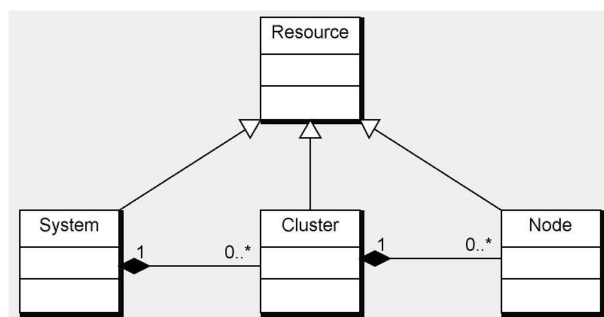

Figure 5.2: The Resource Model.

The *resource model* describes the structural organization of the system. The main concepts are those of *System*, *Cluster*, and *Node*, as shown in Figure 5.2:

- A *Resource* is the top-level concept which may be extended if the framework needs to be refined. It serves as coupling point with other models to describe different system functionalities.

- The *System* class represents the overall system to be protected (i.e., the pervasive network). It is organized into clusters.

- A *Cluster* is a coarse-grained structural unit including a set of nodes which collaborate to achieve some tasks, e.g., to provide a given service.

- A *Node* is the minimal structural unit. In pervasive networks, it represents a mobile device able to perform several functions and communicate with other nodes.

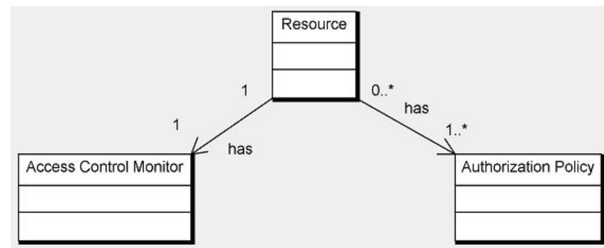

Figure 5.3: The Security Model.

5.4.2.2 Security Model

The *security model* specifies the authorization functionality to control access to *Resources*. The main concepts are those of *Access Control Monitor (ACM)* and *Authorization Policy* as shown in Figure 5.3:

- The *ACM* is a reference monitor which controls all access requests to resources.
- The *Authorization Policy* expresses conditions under which authorizations are granted or denied. It is specified according to the policy model previously described.

5.4.2.3 Autonomic Model

The *autonomic model* specifies how self-configuration and self-protection are achieved in the system. The *self-protection model* adapts authorization policies according to evolution of the context. The *self-configuration model* customizes authorization policies according to resource types, user preferences, or administrator-defined security governance strategies.

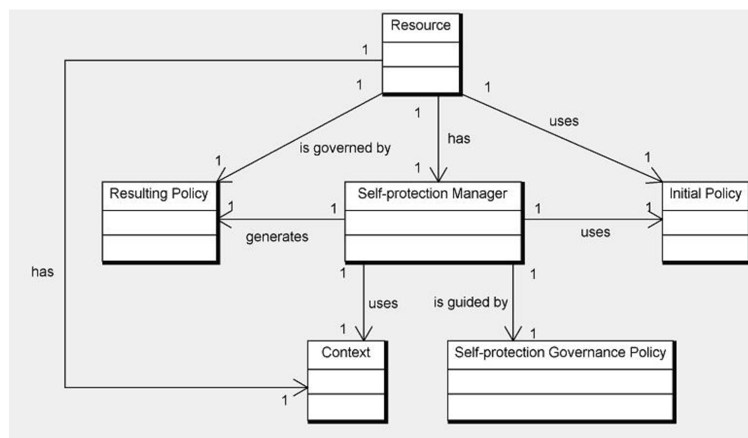

Figure 5.4: The Self-Protection Model.

The *self-protection model* describes how adaptations (selection of adequate security counter-measures) are launched at run-time, driven by evolution of the security context.

Adaptations are performed both at the cluster level and at the node level. The main element of the model are the following, as shown in Figure 5.4:

- The *Self-Protection Manager* controls and orchestrates all activities related to self-protection. Its main role is to monitor the context and update authorization policies.
- The *Self-Protection Governance Policy* captures the administration strategy for self-protection. It drives decision-making, specifying how to select the right authorization policy according to context information.
- The *Context* captures all information about the system environment which may influence such decisions.
- The *Initial Policy* is the current authorization policy, input for the context-aware security adaptation process.
- The *Resulting Policy* is the authorization policy output of the security adaptation process. This policy is generated by the *Self-Protection Manager*.

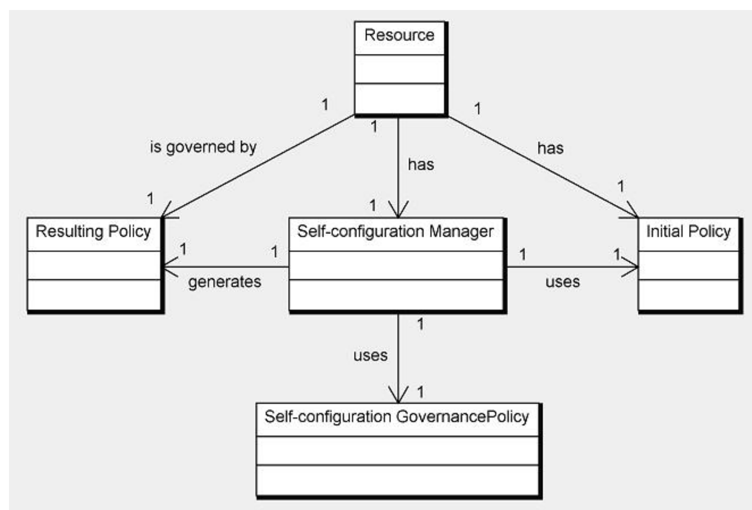

Figure 5.5: The Self-configuration Model.

Once a *Resulting Policy* has been generated, this new policy should be propagated through the whole network for enforcement. Similar to the MIRAGE framework [76] which analyzes, customizes and deploys security policies through networks, the *self-configuration model* of ASPF manages access control policies through large-scale distributed systems. Global policies issued from the *self-protection model* should be translated into local ones to be enforced by each node. The *self-configuration model* specifies this translation process. The main element of the model are the following, as shown in Figure 5.5:

- The *Self-Configuration Manager* is the component in charge of the self-configuration process. It generates a *Resulting Policy* based on an *Initial Policy* according to a *Self-Configuration Governance Policy.*

- The *Self-Configuration Governance Policy* contains the guidelines for the translation process. It may be specified by condition-action rules.

- The *Initial Policy* is policy output of the self-protection model, input for the self-configuration process. It typically represents the new network security policy.

- The *Resulting Policy* is a policy derived from the *Initial Policy*, customized for each resource. It typically represents the new node security policy, adapted to the node-specific setting, e.g., by filtering all network access control rules not involving directly that node to comply with node computational limitations.

5.4.3 ASPF Extended Model

The role of the ASPF core model is to describe the security framework independently from the type of large-scale system. However, to be useful in practice, the framework must be described in a concrete setting. This is the purpose of the *extended model* which specifies the security framework for a specific type of large-scale system such as pervasive networking or cloud computing infrastructures. We now present an extended model for the pervasive setting which was the core focus of our study. However, another extension for cloud environments is detailed in Chapter 10.

As pervasive systems are modeled as clusters and nodes, two extended models are defined to describe self-management of security at the cluster and node levels.

5.4.3.1 Cluster Extended Model

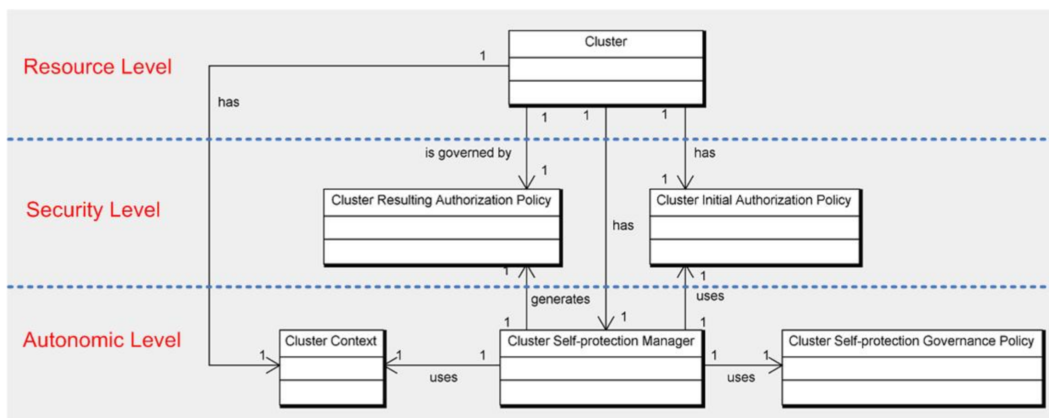

Figure 5.6: A Cluster Extended Model.

The main elements of the model are shown in Figure 5.6.

- The *Cluster Self-Protection Manager* captures the overall intelligence for self-protection of a cluster, coordinating the different necessary components.

- The *Cluster Context* class captures information about the context of a cluster. It may be specified using a more detailed context model such as DEN-ng [129] describing multiple dimensions of context.

- The *Cluster Self-Protection Governance Policy* captures the strategy to select the most adequate security function based on the cluster context.

- The *Cluster Initial Authorization Policy* is the starting point for the security adaptation process. It may be initially one of a set of predefined policies.

- The *Cluster Resulting Authorization Policy* is the result of the security adaptation process, and is generated by the *Cluster Self-Protection Manager* according to the current cluster status. That policy will then be applied to all nodes of the cluster.

The ASPF modular design into several models makes it more easy to select only the features necessary for the considered setting: compared to the core model, the cluster extended model only integrates the self-protection model. Authorization and self-configuration are left aside since: (1) policy enforcement is performed directly in the nodes; and (2) policy propagation towards nodes will be specified in the node extended model.

5.4.3.2 Node Extended Model

The main elements of the model are shown in Figure 5.7.

- The *Node Self-Configuration Manager* coordinates the components for self-configuration at the node level, i.e., to propagate adaptations decided at the cluster level. Such operations will be performed according to the *Node Self-Configuration Governance Policy*.

- The *Node Self-Protection Manager* orchestrates the components for self-protection of a node. Such operations will be performed according to the *Node Self-Protection Governance Policy* which describes reactions (i.e., authorization policies) to apply in security-sensitive situations, based on the *Node Context*.

- The *Node Resulting Authorization Policy* is the final output of the ASPF framework: after the adaptation process, both at the cluster and node levels, this policy will be installed inside the node for access control enforcement by the *Node Access Control Monitor*.

Overall, at the node level, self-management of security is a combination of self-configuration and self-protection: the result of the security adaptation process at the cluster level (*Cluster Resulting Authorization Policy*) is transformed into a *Node Resulting Authorization Policy* (self-configuration). Updates on the *Node Resulting Authorization Policy* will also be performed based on the node context (self-protection).

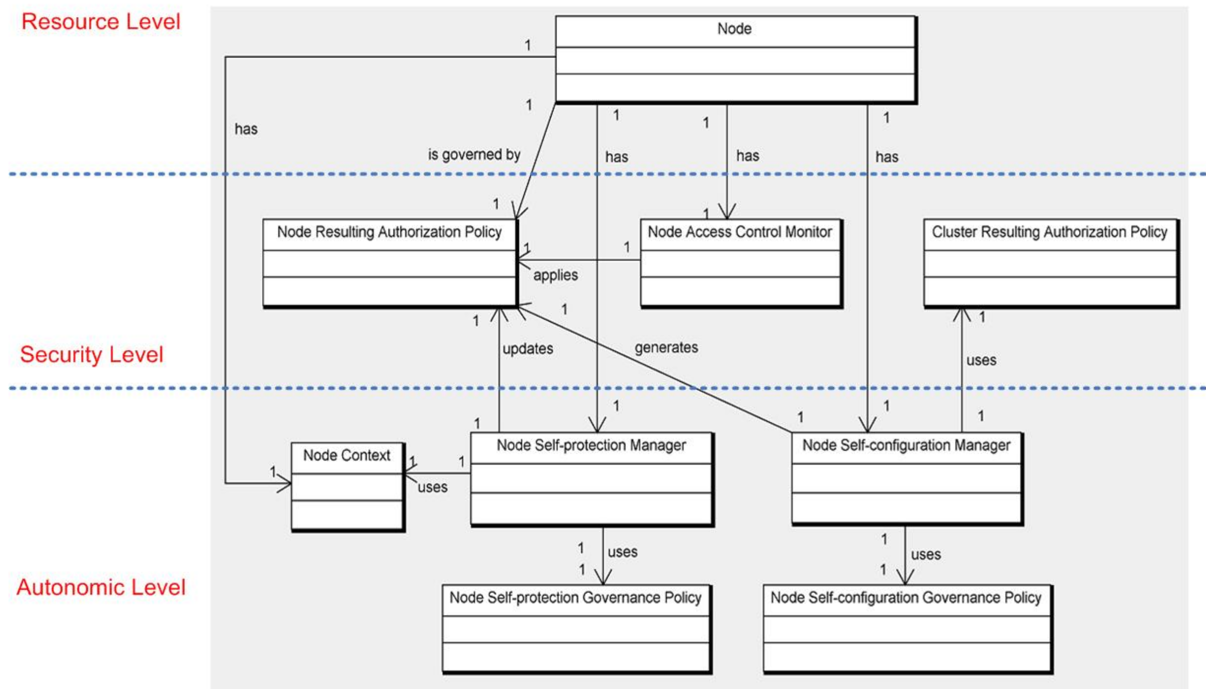

Figure 5.7: The Node Extended Model.

5.4.4 ASPF Implementation Model

The previous models are now refined at the implementation level, different implementation architectures being possible. In the sequel, we present an implementation model which fulfils the requirements presented in Section 5.1.

5.4.4.1 Cluster Implementation Model

The elements of the model are shown in Figure 9.1.

- The *Cluster Authority* component implements the *Cluster Self-protection Manager* class. It coordinates all self-protection tasks in the cluster.
- The *Cluster Context Monitor* provides a representation of the cluster security context. It aggregates low-level inputs from different sources (system/network monitoring probes, sensors,...), relying on context management infrastructures or intrusion detection systems.
- The *Cluster Authorization Policy Repository* contains a set of initial cluster authorization policies to enforce protection within different potential situations.
- The *Cluster Governance Policy Engine* generates security adaptation strategies to tune authorization policies to the environment, e.g., use DTE (resp. MLS) policies

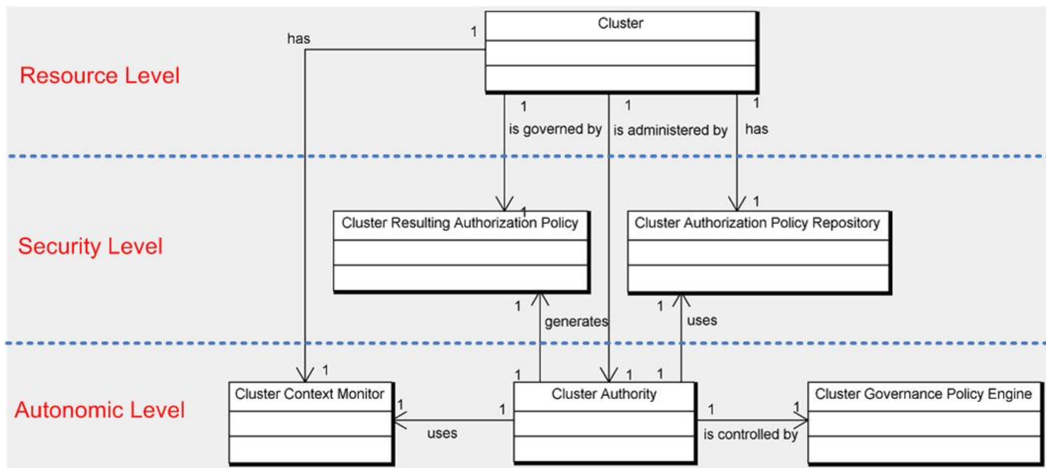

Figure 5.8: The Cluster Implementation Model.

in a friendly (resp. hostile) setting. It may also define new policies to cope with unknown situations.

- The *Cluster Resulting Authorization Policy* is the output of the cluster-level security adaptation process.

5.4.4.2 Node Implementation Model

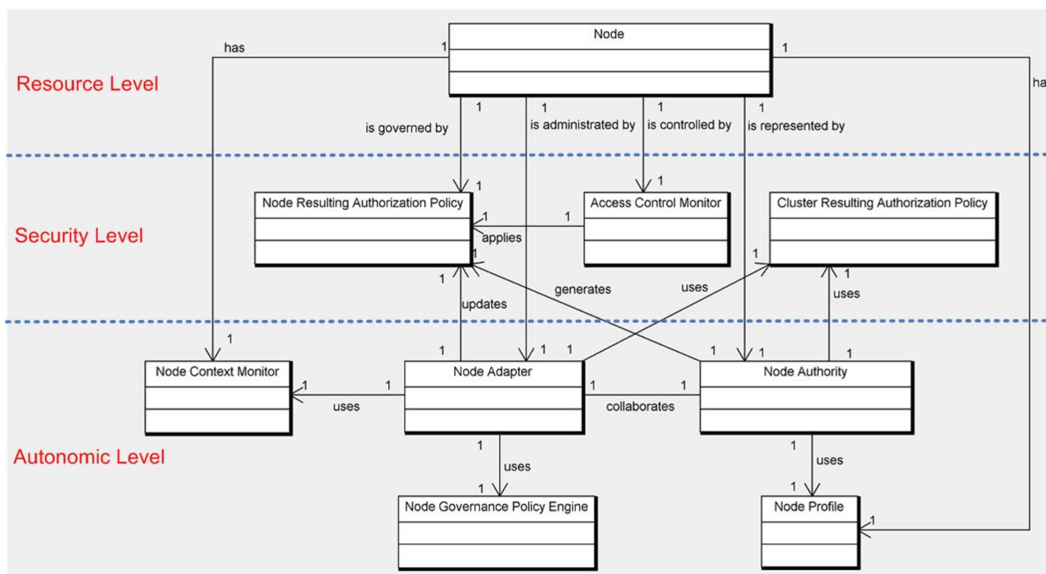

Figure 5.9: The Node Implementation Model.

The main elements of the model are shown in Figure 9.2.

- The *Self-Configuration Manager* and *Self-Protection Manager* functionalities are implemented by two components, the *Node Authority* and the *Node Adapter*. The *Node Authority* typically resides on a server at the cluster-level, while the *Node Adapter* is a component local to each node.

  The *Node Authority* is the main control point to administer security configurations and customize authorization policies.

  The *Node Adapter* is a local security controller in the node with two roles. It is a proxy for the remote *Node Authority*, executing its decisions and installing in the node authorization policies customized at the other endpoint. It also realizes node-level self-protection to adapt node authorization policies based on the node context.

- The *Node Profile* refines the *Node Self-Configuration Governance Policy* by describing the node capabilities (CPU, memory, storage...). As a cluster might contain many nodes, a large part of cluster policy rules might not be relevant for each node and should be filtered. In our design, node-level self-configuration is viewed as filtering the cluster authorization policy according to constraints described in this profile.

- Other components such as the *Node Context Monitor* or the *Node Governance Policy Engine* play the same roles as on the cluster side, but for the node setting.

- The *Node Resulting Authorization Policy* is the final output of the node-level security adaptation process. The corresponding access control rules may then be enforced in the node with a lightweight authorization overhead thanks to the underlying VSK OS.

5.4.4.3 A Double Control Loop for Self-Protection

ASPF regulates security at two levels, using separate feedback loops, both at the cluster and node levels. The previous implementation components interact as follows.

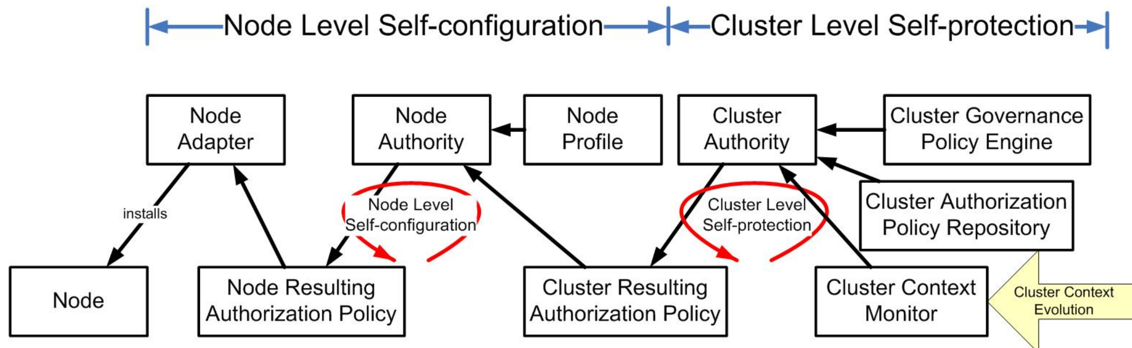

Figure 5.10: The Cluster-level Self-protection Control Loop.

Cluster-level Self-protection: This loop shown in Figure 5.10 aims to mitigate threats against a cluster. The *Cluster Context Monitor* aggregates security-relevant events to

reach a representation of the cluster security context. It notifies the *Cluster Authority* in case the context changes. The *Cluster Authority* then updates the *Cluster Authorization Policy*, according to the strategy specified in the *Cluster Self-Protection Governance Policy*. This operation may be performed by selecting a predefined stronger/weaker policy from the *Cluster Authorization Policy Repository*.

Nodes have severe resource limitations, for instance in terms of computing capabilities or power consumption. Execution must therefore be optimized. The chosen cluster policy is further interpreted by each node according to its specificities (CPU, memory, battery, etc.) captured in the *Node Profile*, generating a new node authorization policy (*Node Resulting Authorization Policy*). Policy rules not relevant for each node are notably discarded. This policy is then loaded into the node authorization sub-system for enforcement. This customization improves efficiency and scalability. It also makes the system more manageable by reducing the number of authorization rules.

For example, a malicious node attack occurs on a cluster, and it is detected by an intrusion detection system. With the predefined adaptation policy, a more secure policy needs to be selected and implemented instead of the previous policy. A more detailed example will be given in Chapter 7. Through the customization mechanism of the self-configuration, this policy will be translated into a *Node Resulting Authorization Policy* based on the *Node Profile*.

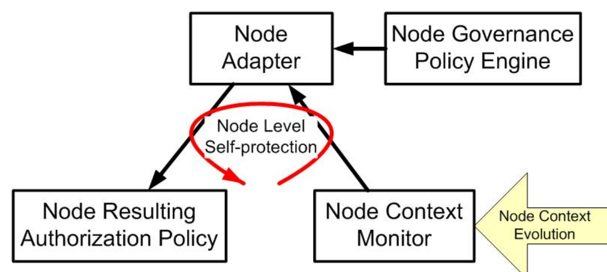

Figure 5.11: The Node-level Self-protection Control Loop.

Node-level Self-protection: A simpler loop also operates at the node level to defeat attacks on a single node as shown in Figure 5.11). Based on information about the node security context (captured by the *Node Context Monitor*), this loop tunes security attributes such as assigning a different role to a subject in order to reduce his privileges in a hostile environment, without modifying the rest of the node authorization policy. For instance, when a node is attacked, the security level of a highly sensitive resource could be increased from *Confidential* to *Top Secret* to minimize possibilities to access the resource.

5.5 Authorization Architecture

The ASPF framework integrates the self-configuration and self-protection models into the XACML authorization framework (see Figure 5.12). XACML defines four main components for policy enforcement (PEP), decision-making (PDP), administration (PAP), and

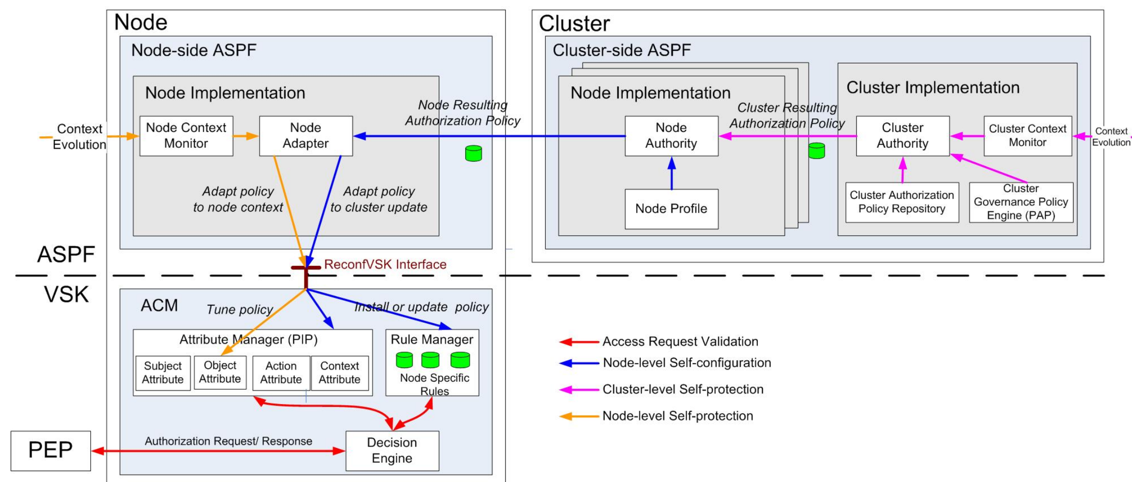

Figure 5.12: The Authorization Architecture.

management of attributes (PIP) [77]. Within the VSK OS architecture which is described in the figure and will be elaborated in the next chapter: PEP is achieved by the kernel, which enforces authorization on execution resources; PDP is the *Decision Engine* component (see Figure 5.12); PIP is the *Attribute Manager (AM)* component that provides additional information for access validation. The authorization policy is stored in the *Rule Manager (RM)* component.

Access requests to resources (located in the execution space) are forwarded to the *Decision Engine* and transformed to a G-ABAC compliant request. Attributes are fetched from the AM, and the request is validated against the authorization policy. The decision is then enforced by VSK which reconfigures the *execution space* to establish access to requested resources.

ASPF may be seen as enhanced PDP. Pure decision-making is extended with self-management capabilities to autonomically generate or tune the authorization policies contained in the VSK ACM based on policy sets written by a cluster network administrator.

Figure 5.12 also shows how ASPF realizes the two self-protection control loops described in Section 5.4.4.3. The cluster-level self-protection model together with the node-level self-configuration model achieve a global control loop which updates both rules and attributes of authorization policies according to cluster context and node profile information. The node-level self-protection loop tunes security attributes based on node context information. The overall architecture not only performs access control enforcement and decision-making. It also improves management of authorization policies, notably by enabling context-aware adaptations thanks to autonomic features.

5.6 Execution Support Mechanisms

The main design concepts are described previously. In order to perform these concepts, some execution supports are firstly described in this section. More details about these supports will be elaborated in Chapter 9.

5.6.1 Delegation Approach

Delegation approach uses local representatives to control remote and distributed entities. A local representative covers the whole life-cycle of controlled entities, ranging from their creation, initialization, to migration or deletion. Since our working system is a dynamic and open system, mobile nodes require a whole life-cycle control. The delegation approach guarantees some main behavior of remote entities by using local representative, and it is applied in the ASPF framework for node administration.

A node authority in a server can manage a remote node. The role of the server is to create and manage authorities of remote nodes. These local representatives are able to interact directly with the remote node to control their life-cycle. Operations of a node can be achieved by operations on its local representative. The design requirement of heterogeneity of equipments is overcome because ASPF interacts only with local representatives which have a unified interface in hiding various characteristics of remote nodes.

5.6.2 Dynamic Reconfiguration

In order to coordinate the life-cycle of each node, the server is able to administer the life-cycle of its representatives. It should dynamically create or withdraw a representative upon availability of nodes. When a new node joint the system, a representative corresponding to the node needs to be created in the server. Future manipulations on the node will be achieved through its representative. When this node is physically destructed or leaves the system, the representative should be removed from the server, and no more operations can be applied to the coupled node.

Moreover, as described in Chapter 3, manipulations as suspension or restart of applications (nodes) are needed since a node may be in several phases of life-cycle. However, this chapter only addresses the policy management, more details about the dynamic reconfiguration realization will be detailed in Chapter 8.

5.6.3 Authorization Policy Enforcement

Authorization policies are in the central of the ASPF framework where only policies are updated to realize adaptation or modification. Administration of policies asks for abilities to initiate, update, or replace remote policies that are installed in each node.

Policy initialization refers to the installation of a customized authorization policy into a node. Policy update modifies some policy attribute values without changing access control rules. Policy replacement changes both policy attributes and policy rules. The achievement of the policy administration depends on underlying platforms. More details about policy enforcement will be given in Chapter 6 and Chapter 9.

5.7 Summary

This chapter presented ASPF, a policy-based security management framework illustrating the design of an autonomic security manager to control authorization policy deployment. ASPF implements two self-protection loops, authorization policies being adapted according to security context variations both at the cluster and node levels. Policies are expressed with an attribute-based extension (G-ABAC) to support policies specified in multiple authorization models.

For such a large-scale, distributed policy management framework, the design requirements such as the G-ABAC based, architecture scalability, policy consistency, user-friendliness, and context-awareness are fulfilled through the use of the policy-based approach, decentralized authorization architecture, self-protection model and self-configuration model. Furthermore, the separation of the ASPF core model from the extended model and the implementation model improves the flexibility in applying the same models in different context.

However, the authorization policy administered and deployed by ASPF should be enforced by the underlying OS. The integration and implementation of G-ABAC based authorization policies at the OS level will be elaborated in the next chapter. The reusability of the ASPF model in the context of cloud computing will be described in Chapter 10.

Chapter 6

Virtual Security Kernel (VSK)

Based on the G-ABAC approach (Chapter 4), the policy deployment framework ASPF (Chapter 5) administers authorization policies through the whole pervasive system. In this chapter, a new OS authorization architecture called *Virtual Security Kernel (VSK)* is proposed which enforces G-ABAC authorization policies at the OS level. With evolution of the context, ASPF updates G-ABAC authorization policies. The updated policies are installed into VSK, and later access requests is thus validated in using the updated authorization policies.

VSK completely separates a minimal kernel control (the *control plane*) from execution resources (the *execution space*). The kernel performs efficient run-time reconfiguration of resources. It also manages authorization through a policy-neutral reference monitor, protection being non invasive thanks to an optimized access control strategy. The architecture is completely component-based, which allows flexibility both at resource level for customization, and in the kernel to support multiple authorization policies and enable their run-time reconfiguration.

Our architecture for self-protection framework is divided into 3 layers (see Chapter 3): (1) the *execution space* provides a run-time environment for components, either application- or system-level; (2) the *VSK Control plane* controls the *execution space*, both to enable application-specific customization and to guarantee security of resources; and (3) the *ASPF autonomic plane* performs automatic adaptation of authorization policies enforced in VSK. VSK consists of a *Virtual Kernel (VK)* and of an *Access Control Monitor (ACM)*. VK provides run-time management capabilities over components and their bindings to reconfigure the execution space. Access control to components is optimized by enforcing ACM decisions at binding creation time only, a binding being considered secure for subsequent access requests until the next change of authorization policy. Apart from these operations, VK remains hidden in the background to minimize interactions between kernel and execution space for performance optimization. ACM is a flexible decision engine allowing run-time selection of different authorization policies. ACM design is compliant with G-ABAC, by managing separately security attributes and rules, both of which may be dynamically updated. Furthermore, VSK is supervised by ASPF which triggers reconfiguration of authorization policies depending on context evolution.

Section 6.1 presents some design requirements for such an OS authorization architecture. We believe that a mobile terminal in our pervasive system should support G-ABAC based policies for collaboration with ASPF and for policy-neutrality. It needs to be efficient for resource-limited terminals. Finally, integration with ASPF asks for some run-time control supports.

Section 6.2 takes into account existing OS implementations in following the design requirements. It examines the OS architecture evolution, dynamic reconfiguration mechanisms, and authorization validations. A short comparison at the end of this section analyzes existing solutions of these three issues.

Section 6.3 outlines main features of VSK. VSK applies the *Configuration Manager* approach to the existing exokernel architecture. It is based on Component-Based Software Engineering (CBSE) to enforce policies specified by G-ABAC. One optimized access control solution, *One-time Check*, is proposed to reduce authorization overhead of VSK.

Section 6.4 elaborates the VSK architecture which consists of *Virtual Kernel (VK)* to supervise the *execution space* and *Access Control Monitor (ACM)* for authorization validation. An example illustrates functionalities of such a system.

Section 6.5 specifies some execution supports that should be fulfilled. Thanks to dynamic reconfiguration of the VSK kernel, authorization policies can be revocable. Dynamism of pervasive environments calls for co-existence of multiple authorization policies. At the end, the VSK kernel needs to be extensible to achieve supplementary functionalities.

Section 6.6 concludes the VSK authorization architecture. It illustrates integration of G-ABAC policies into the VSK OS and its collaboration with the ASPF policy framework. Hence, an end-to-end self-protection framework is realized.

6.1 Requirements

In order to build an OS architecture which adapts the ASPF framework for pervasive systems, several design requirements are listed. These requirements explain terminal-side functionalities for the end-to-end self-protection framework.

6.1.1 G-ABAC Support

Dynamism of pervasive systems allows mobile nodes to dynamically migrate between multiple clusters where each cluster has its own authorization policy. This calls for a policy-neutral OS which can enforce authorization policies of different classes. Because of policy-neutrality of G-ABAC (see Chapter 4), an OS supporting G-ABAC is able to specify a variety of authorization policies. Policy-neutrality can be achieved through the use of G-ABAC. Hence, G-ABAC is chosen as the approach to specify authorization policies of the OS level.

Requirement $\mathcal{R}$ 6.1 *For the reason of policy-neutrality, VSK will provide mechanisms to support policies built on G-ABAC.*

6.1.2 Efficiency

Since the architecture may be used for resource-constrained terminals, it should offer enough efficiency. The OS needs to execute applications in some specific platforms to meet resource limitations of embedded devices. This efficiency calls for full customization with limited performance overhead.

Requirement $\mathcal{R}$ 6.2 *VSK should be efficient to meet computing limitations of resource-constrained terminals.*

6.1.3 Run-time Control

Mobile devices of pervasive systems dynamically migrate between multiple environments, each with their own security settings. Such terminals should provide run-time control that enables dynamic OS reconfiguration for different environments.

Requirement $\mathcal{R}$ 6.3 *VSK should support run-time control for dynamism of pervasive systems.*

6.1.4 Other Requirements

Other design requirements such as portability of the VSK OS for different hardware platforms, customizability for OS services, and adaptability in responding to the context evolution are also taken into account in the VSK design.

6.2 Related Work

An overview of existing OS architectures together with their run-time control and authorization enforcement are given in order to find out an appropriate solution for OS authorization architecture of pervasive systems.

6.2.1 OS Architecture

The architecture of OSes has considerably evolved from *monolithic kernel*, *micro-kernel*, *hybrid kernel*, *exokernel*, to the *Configuration Manager* approach. In this section, different kinds of OS architectures are examined in chronological order.

Monolithic Kernels: A unique module which contains the whole OS were firstly proposed as *monolithic kernel*. The early *DOS*, *OS/360*, *OS/2* and *Multics* are monolithic kernels. Layered *monolithic kernels* were developed in which functionalities are organized hierarchically, and interactions only take place between adjacent layers. A typical example of layered *monolithic kernel* is the traditional *UNIX* OS which is divided in three layers: kernel layer, system call interface layer, and *UNIX* commands and libraries and user-written applications layer. Going one step further, each layer in the kernel was modularized as libraries or components for simplification. Some systems like the *BSD (Berkeley*

Software Distribution) series, the *SVR V* Release, *MAC OS (up to 8.6)*, and *Windows 9x* belongs to this category. The efficiency is guaranteed since no mode change overhead is introduced within one kernel space.

Micro-kernels: Because of the increasing complexity of OS, the structure of monolithic kernel became out of hand. The micro-kernel approach improves the organization of OS by exporting most of no essential services out of the kernel. As a solution for a smaller, lightweight, portable, extensible kernel, the micro-kernel design features a minimal OS kernel which does not provide most of OS services. Only necessary mechanisms such as low-level address space management, thread management, and inter-process communications (IPC) are implemented in the kernel.

Chorus [121] separates functions into a 3-level hierarchical OS for modularity and portability. A minimal micro-kernel called *Nucleus* integrates distributed processing and communication at the lowest level. Several sub-system servers can be implemented in the intermediate level which extend standard OS interfaces. Application programs with associated libraries run on the top level.

K42 [22, 93] "micro-kernelizes" *Linux* to achieve good performance, scalability, customizability and maintainability. It only provides some basic services like memory and process management, IPC infrastructure, etc. Other OS functions are implemented as user-level services. Customizability was introduced in *K42* to improve performance, a service implementation could be customized on demand.

CAmkES [94] is a component-based implementation of the *L4* micro-kernel. Its micro-kernel architecture is used to increase reliability and simplicity. It provides support for developing application components on top of a small micro-kernel.

Other micro-kernel systems like *Mach*, *EROS* [131] present a similar architecture: a micro-kernel providing basic services, and other functionalities being supported in the user mode. An invocation of a basic service (inside the micro-kernel) from a functionality service (outside the micro-kernel) calls for the mode change (from the user mode to the kernel mode). As a matter of fact, the performance and communication overheads of these cross-mode invocations are not negligible.

Hybrid Kernels: The hybrid kernel concept brings together the monolithic and micro-kernel design advantages. Based on the micro-kernel architecture, a hybrid kernel implements some applications in the kernel mode. This reduces the performance and communication overhead associated with message passing and context switching between the kernel and user modes.

SPIN [29, 30] installs important services in the micro-kernel in a dynamic manner. It provides an adaptable kernel that enables resources to be efficiently and safely managed by applications. This approach allows application-specific knowledge to be directly integrated in the management of application resources. In adding the application-specific services into the micro-kernel, cross-mode invocation and protection overheads are reduced.

VINO [134] focuses on the reuse and extension of kernel components. It organizes extensions in the kernel, rather than leaving them in user space for the performance im-

provement. At the same time, each application of *VINO* may select its own administration policy for customizability.

Different from micro-kernels, hybrid kernels reduces invocation and protection overheads by implementing some critical services in the kernel mode. Other hybrid systems such as the *Plan 9* kernel, *NT* kernel, or *MAC OS (after 8.6)* uses the same principle.

Exokernel: This approach eliminates the notion that an OS should provide abstraction on which applications are built. Instead, it addresses solely on securely multiplexing raw hardware [69]. It attempts to push the traditional abstraction into the application level. It allows untrusted software to implement traditional OS abstraction entirely at the application level. The only functions which remain in the kernel are devoted to OS protection. It consists of a kernel that multiplexes physical resources securely and of some application-level libraries that are implemented on the top of the kernel. These libraries are customized to each application in order to meet the best performance and functionality goals.

Virtual Machine Monitor (VMM) [135] is used to coordinate multiple guest OS environments and protect shared resources. It applies the exokernel approach for hypervisors with a thin layer for abstraction and access control. *Xen* [25] is a typical lightweight VMM between guest OS environments and hardware that merely provides basic control operations.

The exokernel approach provides more control to each application and features a tiny kernel. It sticks to the design principle to combine applications with OS services (applications can choose their needed OS services) in order to increase efficiency and security.

Configuration Manager (CM): The CM approach does not have any predefined kernel at all. Instead, services are implanted and managed by CM which separates the control plane from the execution plane: all checking and control actions take place during the initialization of applications, no control is performed during execution.

DEIMOS [48] is an extensible system which enables each application to build and customize its own execution environment. In *DEIMOS*, both traditional kernel functions and application-specific services are encapsulated as modules which can be loaded, configured, and unloaded on demand by CM. CM is also responsible for the management of interfaces of these modules. It creates and destroys bindings between interfaces on these modules, and manages conflicts between modules. As CM is the only system component able to establish bindings, it performs all checking actions during the binding creation. The non-kernel architecture reduces kernel performance overhead and provides customizability and adaptability.

6.2.2 Dynamic Reconfiguration

The *Configuration Manager* approach appears as a promising paradigm for future OS. However, it calls for dynamic reconfiguration which modifies a system at run-tim. Taking into account different phases of system life-cycle and different granularity of reconfiguration, we divide dynamic reconfigurable systems into five categories: *Build-Time Extensible*

Systems, *Library Extensible Systems*, *Dynamic Component Image Loading Systems*, *Dynamic Functional Code Loading Systems*, and *Dynamic State Variable Loading Systems*.

Build-Time Extensible Systems (BTES): The first proposition of dynamic reconfiguration is BTES which have just a modular architecture during the design phase. Different modules can be chosen off-the-shelf to build a target system.

Early releases of *Linux* can be considered as belonging to BTES. Modules are a collection of routines that perform system-level functions. For instance, a device module is a non-linked driver which contains device-specific system routines. However, the architecture of the system cannot be modified once it is compiled.

eCos [2] is a component-based framework to build Real-Time OS (RTOS). Reusability is the principal objective of *eCos*: most of application functionalities can be built by reusing existing components. A developer has more control over components by selecting reusable components for his system. He can remove unnecessary functionalities of selected components, modifies or configures components for different implementations. *eCos* uses compile-time control methods for its software components, along with selective linking provided by a GNU linker.

Library Extensible Systems (LES): LES contains code to extend a kernel at run-time. Code in format of libraries has already been compiled. Those libraries reside in memory without being integrating into the OS image file. This solution avoids implementing unnecessary OS modules which waste memory. Whatever needed, users rebuild and reboot the OS file to add new libraries. Most current UNIX-like systems and *exokernel* systems, can be considered as in this category.

OSKit [74] is able to construct not only OS services, but also the OS kernel without a predefined basic core structure. It provides a framework and a set of modularized library code for the construction of OS kernels, servers, and other core OS functionalities.

Dynamic Component Image Loading Systems (DCILS): One step further, DCILS is able to load a component binary image at run-time. A component binary image consists of three parts: architectural meta-data, functional code and a state variable. The architectural meta-data describes components, their interfaces together with their internal structure information. It is usually generated and manipulated by a component model. In terms of the dynamic reconfiguration at the OS level, since we consider general mechanisms, we suppose that its modification is achieved by the component model, and the update of the architectural meta-data is not included in the OS level reconfiguration mechanisms. The functional code is unchangeable code which describes operations that can be performed on the components. The state variable is the data structure that represents temporary states of the components. Therefore, all the newly installed components start with a freshly initialized state.

DYMOS [52] is an early system that enables a programmer to modify a program dynamically. A developer should modify and re-compile the source code (functional code)

to be replaced and then requests the system to change the old image. Three main steps are designed for the reconfiguration: edit, compile, update (when idle).

SPIN [29] allows OS services components to be dynamically loaded into its hybrid kernel to meet application requirements. For easy extension, it separates resource operations from services by interfaces.

Think [20] is a component-based framework which implements the *Fractal* [41] component model. A component of *Fractal* is both a design and run-time entity that constitutes a unit of encapsulation, composition and reconfiguration. It provides great flexibility for OS kernel construction, and is able to be implemented on different hardware platforms such as *ARM*, *AVR*, or *MPS430*. In *Think* [117], dynamic reconfiguration is performed in four steps: (1) identify the part of system to reconfigure; (2) suspend its execution; (3) modify the system (add, remove parts of the system); and (4) transfer the state to the new parts and resume the execution of the system.

In addition to LES, DCILS manages newly loaded component images. Other systems like *VINO*, *CAmkES*, *DEIMOS* are also dynamic image loading systems. Since after each reconfiguration, the initial state variable is applied to the reconfigured part which makes it return to an initial state. Some execution data may be lost or destroyed due to the state variable initialization.

Dynamic Functional Code Loading Systems (DFCLS): This kind of systems may replace the functional code without changing the state variable, that is, the state of reconfigured part remains the same when we modify operations which can be performed on the variable. This approach requires a fine grained interface for the functional code.

In *MMLite* [84], dynamic reconfiguration can be considered as replacing an implementation object by another in supporting the same *COM* interfaces. This mechanism consists in atomically changing an ordinarily constant implementation part of an object.

K42 [22, 93] uses a hot-swapping mechanism [26] for dynamic reconfiguration. Hot-swapping replaces an component instance with a new component instance that provides the same interfaces. Internal state from the old component is transferred to the new one and external references are re-linked. Hot-swapping allows component replacement without disrupting the rest of the system.

Contiki [67] may load or unload individual services at run-time. A service which is a process that implements a function can be dynamically replaced at run-time and must therefore be dynamically linked. The service has an internal state that may need to be transferred to the new processes. The kernel provides a way to pass a pointer to the new service process. The service can thus produce a state description that is passed to the new process.

Dynamic State Variable Loading Systems (DSVLS): Finally, DSTLS enables component state reconfiguration. In some cases, a component may need to tuned to another state without touching its implementation. This can be realized by changing or replacing the state variable. Due to the complexity of the component state variable, there are unfortunately no systems that support the dynamic state variable loading.

6.2.3 Authorization Enforcement

Using embedded systems in mobile, dynamic, and open contexts makes them more vulnerable. Authorization sub-system of each OS must provides strong protection for different contexts. OS authorization is usually achieved through two paradigm, either conventional single policy enforcement, or policy-neutral architecture.

6.2.3.1 Single Policy Enforcement

The single policy enforcement paradigm implements only one authorization policy in the system where the authorization sub-systems usually depends on the applied authorization policy.

ACL: *Access Control List* model was implemented in *DEIMOS* [48]. As the system evolves, *DEIMOS* can dynamically add or delete name-value pairs and rules in an ACL.

Capability-based: Capability-based access control policies [99] are enforced in *EROS* [131] that divides both applications and OS services into cleanly separated components, each of which resides in its own protection domain (a protection domain being the set of capabilities accessible to a sub-system). *SPIN* [30] implements capabilities directly using pointers. A pointer is a reference to a block of memory whose type is declared within an interface. There is no run-time overhead for using a pointer, passing it across an interface, or de-referencing it, other than the overhead of memory access. A system with capabilities removes the need for any access control list or similar mechanism by giving all entities in the system all and only the capabilities they will actually need.

MLS: *Multiple Level Security* [97] is used in *Multics* which adopts BLP [27] policies for OS protection. Security clearances of subjects are kept in a process-level table. Objects security classifications are directly associated with each resource. To grant an access request, security level of subjects should dominate that of objects. Some extensions of Unix like AT&T's System V/MLS [18] and Sun's CMW [70] can also support MLS.

DTE: *Domain Type Enforcement* [24] has been implemented in *UNIX*. The *DTE/UNIX* prototype implicitly maintains type associations in the kernel to reduce modifications to legacy *UNIX* systems. Another implementation was done in *Linux* [79], where for instance the *DTE/Linux* proof-of-concept attaches types to each *inode* in the file system. Yet, many more enhancements would be necessary to reach a fully DTE-compliant *Linux*.

RBAC: *Role-Based Access Control* policies [127] call for a more complex architecture comparing to previous MLS and DTE policies. Hence, it is mostly applied to information systems, Web environments, Java middleware, and federated database systems (FDBS).

UCON: *Usage CONtrol* [116] extends authorization to cover the whole software lifecycle. In an OS, software update or dynamic reconfiguration requires decision continuity and attribute mutability. In other words, access attributes are no more constant and may be modified during execution. In [151], Xu et al. propose a UCON framework based on VMM architecture for OS kernel integrity protection.

6.2.3.2 Policy-neutral Architectures

We present a few *Policy-Neutral Access Control Architectures (PNACA)* which specify the organization of a system where authorization mechanisms are independent from a specific authorization model.

Extensible AC/SPIN [78] implements *PNACA* in the *SPIN* OS. It separates not only authorization policies from enforcement, but also security mechanisms from functionalities of system. These schemes both improve authorization flexibility and reduce access control overhead. When a *SPIN* module needs to be dynamically linked to another extension, link-time access control is launched. An enforcement manager determines the type and operations exported by that extension, passes this information to a security policy manager. The policy manager decides which types and operations require access control operations and instructs the enforcement manager to inject access control operation into the extension.

The objective of *Linux Security Module (LSM)* is to develop a lightweight, general purpose, access control framework for *Linux*. It enables different authorization policies to be implemented as loadable kernel modules [150]. Its main principle is to insert hooks in the kernel that check permissions before resource access by delegation to particular *LSM* modules.

Although *sHype* [124] was designed for hypervisor architectures, it follows the same principles as *LSM*. It adds hooks which control all requests before accessing to kernel objects.

The aim of *SELinux* [104] is to introduce multiple security policies like DTE, MLS and RBAC in one OS. *SELinux* notably inspired the development of *LSM* to introduce flexible access control in *Linux* by a set of authorization hooks along with tools to verify the correctness of implementation.

CRACKER [95] is a component-based policy-neutral architecture for kernel-level access control. It supports multiple authorization models by enabling a clear separation between policy specification and enforcement. A large range of policies are supported since the chosen specification language, *ASL* [88], is quite expressive. Run-time change of policies is also allowed using reconfiguration mechanisms.

Other systems such as *Asbestos* [68] or *HiStar* [153] enforce information flow control policies which are out of the scope of this thesis. However, all these architectures show the possibility to integrate different access control policies into one system in a flexible manner.

6.2.4 Summary

The current trend in OS architecture is to minimize the kernel as much as possible, from the simple monolithic kernel to the non-kernel *Configuration Manager*. The *Configuration Manager* approach tends to solve problems about performance overheads, extensibility, flexibility, etc. Disappearance of kernel completely resolves performance problem and shows a huge advantage regarding architecture flexibility. We present the OS architecture levels from 1 to 5 respectively for *monolithic kernel*, *micro-kernel*, *hybrid kernel*, *exokernel*, and *Configuration Manager*.

Different types of reconfiguration mechanisms are divided in five levels. More reconfiguration is fined-grained, more system is flexible. BTES (level 1) achieves configuration during the design step. LES (level 2) is able to bind different modules at execution time. DCILS (level 3) allows to insert new components in systems. DFCLS (level 4) can replace functional code part without touching component state. Finally, DSVLS (level 5) is able to tune or change component state variable. From the viewpoint of embedded systems, DCILS seems to be the most adequate.

For a mobile terminal used in different, dynamic and open environments, it should be possible to choose between different authorization policies. The policy-neutral architecture is a good solution to reach this goal, multiple authorization policies may be selected according to the execution context.

Table 6.1 examines existing OSes through these three aspects. Some platforms (*eCos*, *OSKit*, *Think/CRACKER*, *DYMOS*, *MMLite*, *Contiki*) appear as an OS construction tool set rather than specific OSes. With these tool sets, different OS architectures can be realized. This is the reason why multiple architectures are categorized for such platforms.

6.3 VSK Overview

The proposed VSK OS architecture brings together some existing solutions in order to meet all the design requirements. Main features of such an architecture are:

Applying the *Configuration Manager* paradigm to *Exokernel*: VSK is an improvement combining both the *configuration manager* paradigm and the *exokernel* architecture. The *configuration manager* [48] reduces OS management overhead by visualizing the kernel. *exokernel* [69] demonstrated feasibility to export out of the kernel most OS services as applications, only thread management and access control remain inside the kernel. VSK goes one step beyond, the kernel applies the *exokernel* architecture in the sense that kernel entities will be hiden in the background during most of execution for efficiency ($\mathcal{R}$ 6.2). It is of the *exokernel* architecture since it only handles with thread management and access control. This design, while still guaranteeing effective protection, yields significant performance improvement ($\mathcal{R}$ 6.2), as shown by the performance evaluation (see Chapter 8). Furthermore, the *exokernel* architecture enables flexibility of OS dynamical reconfiguration ($\mathcal{R}$ 6.3).

	OS Architecture	Run-time Control	Authorization
Multics	1	2	BLP
Unix	1	2	DTE,MLS
Linux	1	2	DTE,MLS,RBAC
Chorus	2	?	Capability-based
K42	2	4	ACL
L4/CAmkES	2	3	Capability-based
EROS	2	?	Capability-based
SPIN	3	3	PNACA
VINO	3	3	?
Exokernel	4	2	Capability-based
Xen/ VMM	4	3	UCON
DEIMOS	5	3	ACL
eCos	1-4	1	?
OSKit	1-5	3	?
Think/CRACKER	1-5	3	PNACA
DYMOS	1-4	3	?
MMLite	1-5	4	Capability-based
Contiki	1-5	4	no mechanism

Table 6.1: The Operating Systems Comparison.

Applying the Component-based Approach: Modularity has become one main trend of OS design: different services are encapsulated into components, standard interfaces are defined, and communication between components are carried out through interfaces. *Component-Based Software Engineering (CBSE)* [40] supports exactly this by abstracting hardware, applications and OS services as components, viewed as units of design, management, deployment and reconfiguration. The use of components enables code re-use and reduces development life-cycle. For designs like the *configuration manager* paradigm, *CBSE* is necessary since its modularity enables dynamic reconfiguration during execution. This approach provides a homogeneous view of different resources and shows its advantages on modularity, reusability and flexibility. For some dynamically reconfigurable OSes 6.3, it is an inevitable paradigm ($\mathcal{R}$ 6.3).

Enforcing G-ABAC: The policy management framework ASPF has been proposed which can administer authorization policies through pervasive systems. VSK enforces access control policies in a policy-neutral manner. Policy-neutrality is achieved by a clear separation of access attributes from rules in using G-ABAC ($\mathcal{R}$ 6.1). The authorization of VSK is achieved by computing permissions based on attribute values. A detailed example of the G-ABAC authorization in VSK will be given in later sections.

Achieving One-time Security Check: VSK includes non-invasive protection mechanisms thanks to an optimized access control validation called *one-time check*. Within *one-time check*, the first time a subject attempts to access an object, it asks the VSK kernel for authorization. If VSK grants the request, it creates the corresponding binding between the subject and object. For later invocations, the subject can directly access the object without any access control which improves the efficiency ($\mathcal{R}$ 6.2). Only one check is thus achieved for each subject-object pair.

Other Contributions VSK also proposes a separation of a dynamic but lightweight *control plane* (realized by VSK) from execution resources which prevents affects from executing applications to the OS kernel. The realization of kernel reconfiguration allows adaptation of kernel functionalities to the context evolutions. Finally, a revocation mechanism is provided that guarantees consistency of authorization permissions.

6.4 VSK Architecture

A system built on VSK consists in two parts (see Figure 6.1): a run-time environment for components (*execution space*) and a *control plane* (VSK) to supervise their execution in an almost implicit manner. This separation not only provides customization for executing applications (components in the *execution space*) but also protects the kernel from attacks from the *execution space*. Furthermore, a simple but effective access control mechanism in VSK guarantees the security of sensitive resources in the *execution space*.

The *control plane* is realized by VSK which consists of *Virtual Kernel* (VK) and *Access Control Monitor* (ACM). VK plays the role of a kernel when needed (e.g., during

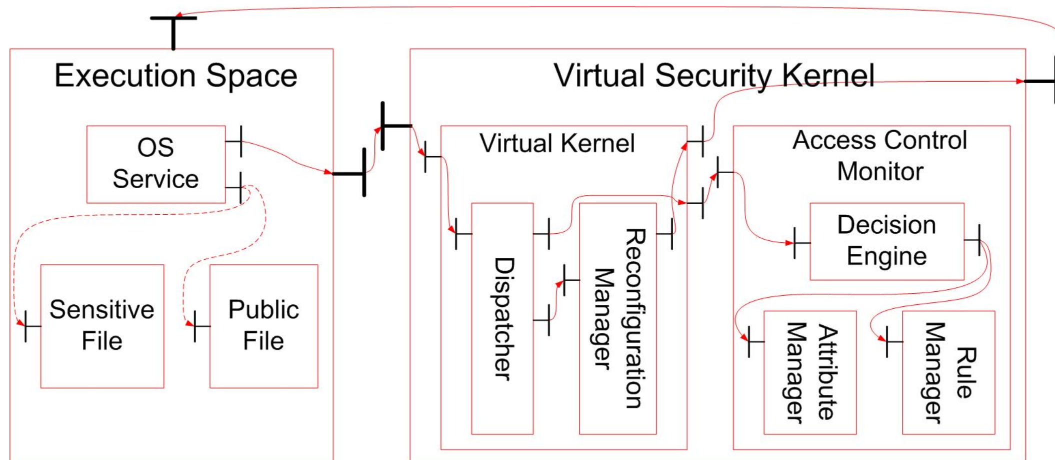

Figure 6.1: VSK Architecture

the initialization and reconfiguration of the execution space) and remains hidden in the background when no more changes occur. It catches events from the *execution space*, decides whether a reconfiguration of that space is required, and reacts accordingly. VK may also dynamically reconfigure the kernel itself according to high-level government strategies, reacting to decisions taken in the *autonomic plane*. ACM enforces access control on resources in the *execution space*, based on access attributes and rules of G-ABAC which may be dynamically loaded, modified, or replaced.

6.4.1 Virtual Kernel (VK)

In the *execution space*, functional code is encapsulated as a component, inter-component communication being realized via bindings. VK supervises and controls the *execution space* with two main functions:

1. a lightweight control *one-time check* over components in the *execution space*;

2. run-time control mechanisms to reconfigure the *execution space*.

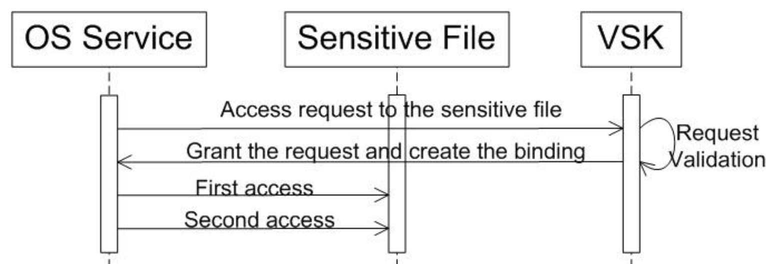

Figure 6.2: VSK One-time Check Sequence Diagram

Within *one-time check*, as shown in Figure 6.2, the first time when an OS service tries to access a sensitive file, it asks VSK kernel for authorization. If VSK grants the request, it creates a corresponding binding between the OS service and the sensitive file. For later invocations, the service can directly access the file without any access control.

This design allows to meet performance requirements, since the kernel overhead is minimized during the execution of applications by limiting interactions between the *execution space* and VSK. From the customization viewpoint, applications can dynamically choose their needed services which increases implementation freedom. Finally, this architecture guarantees strong protection by strict control over creation or modification of bindings. VK consists of a *dispatcher* and a *reconfiguration manager*.

Dispatcher: The *dispatcher* collects execution information from the *execution space*. It provides an effective and extensible event-based communication mechanism between threads. For instance, the *execution space*, VSK and a communication module for interacting with external environment may work in parallel. The *dispatcher* also manages concurrent access to resources by enforcing mutual exclusion. Each time when a subject tends to access an object, an access request is captured by the *dispatcher*. The *dispatcher* coordinates the authorization validation and access control enforcement with the help of the *reconfiguration manager* described in the next paragraph. Finally, this mechanism enables to easily add new security modules in ACM or extend kernel services simply by defining new event types.

Reconfiguration Manager: The *reconfiguration manager* provides run-time component and binding management capabilities to modify the *execution space* based on decisions made in the kernel. For instance, it creates a binding when an access request is granted by ACM. It may also load and install new components in the *execution space*. In the case of permission revocation, it removes all corresponding bindings. Above all, the *reconfiguration manager* is a key building block for efficient access control enforcement. In DEIMOS [48] or sHype [124], access controls were optimized by a *one-time only* enforcement until the next change of authorization policy. In VSK, a similar mechanism is proposed. Access control only takes place during the creation of bindings. Unlike previous systems, all bindings are managed at run-time, and are created only when needed for use. The first time an access to a resource is granted, the corresponding binding is created by the *reconfiguration manager*, subsequent access requests to the resource will not be checked any more.

6.4.2 Access Control Monitor (ACM)

ACM enforces G-ABAC authorization policies. It supports multiple authorization policies and enables dynamic reconfiguration for policy management. It is basically a flexible reference monitor where different authorization policies may be activated dynamically.

In our architecture, active components (subjects) try to access passive components (objects). *Flask/SELinux* [136, 104] has shown its flexibility in separating policy from enforcement mechanisms, with a clear distinction between abstract security identifiers

(SIDs) and dedicated security modules which can process those SIDs. Our ACM goes one step forward in flexibility by separating *access attributes* from *rules* within the G-ABAC model. One SID may be assigned with different security attributes (e.g., role, type, security level, domain, etc.), some of which can be used to compute permissions. When a subject requests to access an object, ACM first gets subject and object attributes, computes permissions via authorization rules, and makes an access decision.

The separation of attributes from rules improves the access control flexibility. It also helps capturing the diversity of security requirements in a mobile context: a device moving to another environment may only require updating access attributes. Thanks to the VSK component-based design, the change of security attributes and rules can be simply realized by replacing components.

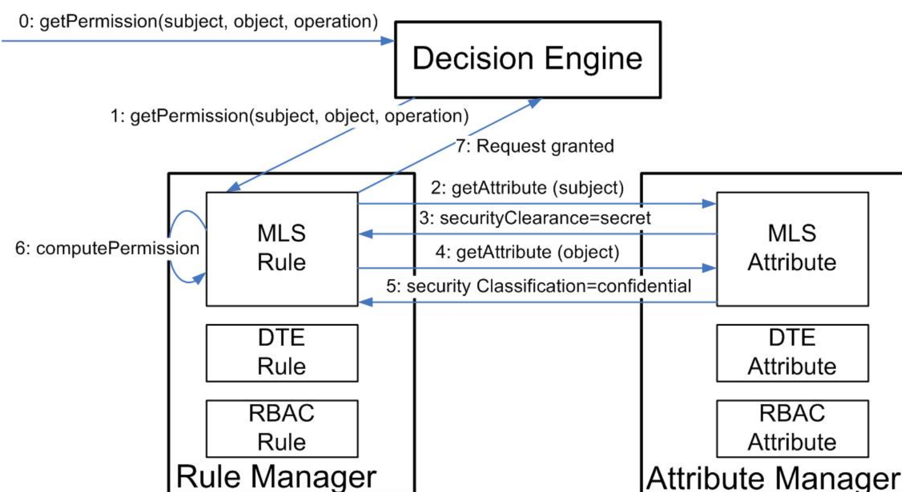

Figure 6.3: The ACM Request Process with MLS.

ACM contains three sub-components: *Decision Engine* (DE) for decision-making, *Attribute Manager* (AM) and *Rule Manager* (RM) for manipulations on attributes and rules respectively. As shown in Figure 6.3, to request access to an object, a subject first issues a request *getPermission(subject,object,operation)* to RM via DE. RM then asks AM for access attributes corresponding to the concerned subject and object. For example, in the case of MLS, the needed attribute for the subject would be its security clearance – a subject-securityClearance table being maintained in the MLS AM sub-component. For the object, a similar object-securityClassification table also resisting in the MLS AM. Once the subject, object attributes are retrieved, RM can then compute permission by examining the dominance between the security clearance of the subject and the security classification of the object for a requested operation – a rule table being maintained in the RM sub-component.

6.5 Execution Support

Upon the VSK design, supplementary functionalities should also be supported.

6.5.1 Kernel Reconfiguration

Mobile terminal systems built on VSK join a pervasive system in a dynamic and open manner. Different execution environments call for different security functions of kernel. The dynamic reconfiguration of OS kernel allows run-time modification of kernel modules to adapt the context evolution.

VSK enables dynamic reconfiguration of its ACM including access attribute and rule replacement together with access attribute modification. New authorization policies may be loaded and installed into ACM which constructs a global self-protection control loop against network level attacks as described in Chapter 5. Attribute value modification realizes a local self-protection control loop as an effective reaction against local attacks. With a revocation mechanism described in the next section, access decision continuity and attribute mutability can be guaranteed.

6.5.2 Revocation

While modifying dynamically access control policies, to avoid privilege abuse, OS should revoke outdated authorization policies. This task is generally out of hands for conventional OS since made decisions may be decentralized throughout the whole system. With VSK, a revocation management is whatever efficient: as authorization is enforced by the *reconfiguration manager*, once there is an update of policy or modification of attribute values, this manager removes all related bindings it created in the *execution space* to guarantee authorization policy consistency. When new access requests are issued for which some authorization is revoked, VSK will rebuild corresponding bindings based on the updated authorization policy.

To illustrate how the revocation is performed in practice with VSK, we consider a mobile terminal connected either to an outdoor, public, insecure network or to a domestic, private, secure network. When the terminal is part of the public network, drivers are installed on the terminal for its communication capabilities of radio access technology. In case of a data connection, these drivers may request access both to sensitive and public files. The driver represents a potential risk for system security which should be mitigated dynamically. Therefore, an OS service such as the wireless driver may only access public files (e.g., user documents) but not sensitive files (e.g., system security settings). When the terminal joins the domestic network, a new authorization policy is installed to adapt this secure environment, the access to sensitive files are granted. The update of authorization policies first removes all existing bindings of the outdated policy. Afterwards, it redirects new access requests to the updated policy.

6.5.3 Multiple Policy Approach

Multiple policy approach activates several policies for authorization validation at the same time. Particularly in highly heterogeneous systems like pervasive systems, this is an promising issue. Since nodes in different environments should respect their own security requirements, the multiple policy approach combines all requirements in validating various authorization policies.

Conventional authorization policies implemented in OS kernels like BLP [27] or Biba [32] lack expressivity. This is the reason why some combined policies like Lipner [101] was proposed which adopts with both the BLP and Biba labels. Furthermore, the combination of existing access control policies can also provide more functionalities. For example, within a context-aware authorization policy, location is taken into account for validation. Within UCON [116], life-cycle of authorization policy is covered. In using these two policies at the same time, context-awareness and life-cycle are embodied together that enrich authorization. Finally, dynamism of pervasive systems allows a node to be shared between multiple clusters which may use different authorization policies. The multiple policy approach enables collaboration of clusters in assuring consistency. In our implementation, for an access of a node shared between two clusters which adopts with their own policies, the request can be granted if and only if both of these two policies grant it.

Inspiring from the XACML [77] framework, VSK achieves the multiple policy approach by simultaneously implementing two access control policies in ACM along with a policy combining algorithm. These two policies need to be checked in parallel for a request, and the decisions of each policy are combined into a final decision by a policy combining algorithm.

6.5.4 Functionality Extensions

VSK is a kernel architecture which applies the exokernel architecture to export most of OS services. These services are achieved in the user mode which is called extensions of the VSK kernel. Different from components in the kernel, extensions cannot be dynamically reconfigured, it resists in the system.

Context-aware Extension: Beyond the VSK kernel, a context-aware extension is developed that provides context information to the kernel. It may collaborate with some monitoring systems like detection intrusion systems in order to supervise execution circumstances. A proof-of-concept prototype of the context-aware extension is implemented and will be elaborated in Chapter 8.

History-based Extension: History-based authorization captures history information to ameliorate access control decision making. For instance, by monitoring a menacing communication, it can block access instead of granting to mitigate threat. However, VSK applies the *one time check* mechanism for authorization which does not provide any monitoring support, history is no more supervised once an access is granted. One solution is to insert interceptors for each binding. The interceptors mediate each communication

and capture history information. However, this technology is not implemented in our framework. More details about interceptors can be found in [117].

6.6 Summary

Within the 3-level policy-based architecture proposed in Chapter 3, the *execution space* is separated from the *control plane*. This chapter focuses on the OS architecture enabling the realization of the *control plane* at the terminal side. A dynamic but lightweight management plane VSK is introduced to enable applications to fully control their execution environment at run-time.

This chapter presented the VSK component-based OS authorization architecture which provides strong and yet flexible security while still achieving good performance, making it applicable to make pervasive devices self-protected. The definition of a dynamic but lightweight *management plane* separate from execution resources allows applications to control and customize their execution environment at run-time, yielding a highly adaptable OS architecture. Including protection mechanisms in this plane also reduces authorization overhead without compromising overall security, thanks to *one-time check* only during creation of bindings until the next change of authorization policy. The use of G-ABAC to specify authorization policies making the OS authorization architecture policy-neutral to support multiple authorization policies. The clear separation of authorization attributes from rules in G-ABAC also improves access control granularity. The component-based structure of the VSK *control plane* allows to reconfigure at run-time kernel access control modules, yielding a flexible and dynamic security architecture.

From the viewpoint of implementing an autonomic framework, an adaptation policy is missing which guides dynamic reconfiguration of the VSK kernel. The specification of such a policy will be given in the next chapter.

Chapter 7

Adaptation Policy Specification

The proposed end-to-end self-protection framework (Chapter 4, Chapter 5, and Chapter 6) so far hardly addressed specification of adaptation strategies which guides risk-aware selection or reconfiguration of authorization policies. Adaptation strategies are usually specified in ad hoc formalisms such as *if-then-else* rules, which are not intuitive for users, and lack error-checking mechanisms. Another aspect also missing is how to easily integrate such policies into the existing security management framework.

Domain-Specific Languages (DSLs) present many benefits to achieve these goals in terms of simplicity, automated strategy verification, and run-time integration. This chapter presents a DSL to describe security adaptation policies. The DSL is based on *Event-Condition-Action (ECA)* approach and on taxonomies of attack and countermeasure. It allows to capture trade-offs between security and other concerns such as energy efficiency during the decision-making phase. A translation mechanism to refine the DSL into a run-time representation, and to integrate adaptation policies within legacy self-protection frameworks is also illustrated.

Section 7.1 outlines design requirements about such an adaptation policy specification language. The DSL should make specification intuitive. It should self-manage various concerns, and make trade-offs between these concerns. Finally, easy integration of adaptation policies should also be taken into account.

Section 7.2 overviews existing works on self-protection frameworks, ontologies of security, and DSLs.

Section 7.3 describes main features of our proposition. The self-protection DSL separates roles of different actors. It applies the *Event-Condition-Action* approach for the specification of policies. The DSL models systems into multiple dimensions of concerns and is able to dynamically generate a run-time representation.

Section 7.4 explains design principles which use the DSL approach for the specification of adaptation policies. It also introduces the whole processus of adaptation policy specification.

Section 7.5 defines a core DSL model, General Adaptation Policies (GAPs) together with associated attack and countermeasure taxonomies to specify adaptation policies for self-protection.

Section 7.6 shows procedure and mechanisms to translate GAPs to a more efficient and robust run-time representation through Autonomic Adaptation Policies (AAPs).

7.1 Design Requirements

A *Domain-Specific Language (DSL)* is a high-level language devoted to explicit different perspectives of a given domain, providing specific language support to describe several types of policies [65]. Compared to administration APIs, the major requirements of using a DSL are:

7.1.1 Intuitive Representations

The specification of adaptation policies should be intuitive. Conventional *if-then-else* rules seems no adequate since dependance between different rules cannot be explicitly illustrated.

Requirement $\mathcal{R}$ 7.1 *The self-protection DSL should be intuitive for specification.*

7.1.2 Self-managed

Adaptation policy specification are manipulated by various actors where each one works on their domain of interest. Thus, adaptation policy becomes complex and unmanageable. Self-management paradigm which makes policy specification autonomous appears as a promising solution.

Requirement $\mathcal{R}$ 7.2 *The self-protection DSL should be self-managed to weave actors of different domains.*

7.1.3 Enabling Trade-off between Multiple Concerns

Diversity of pervasive systems calls for trades-offs between concern dimensions. With some resource limitation, adaptations should take into account other concerns than security (e.g., in wireless sensor networks, energy-efficiency may limit the use of sensors. Adaptation of security functions of a sensor should accord with its physical settings). The self-protection DSL should enable trade-offs between concerns.

Requirement $\mathcal{R}$ 7.3 *The self-protection DSL should take into account different concerns to make trade-offs.*

7.1.4 Easy for Integration

The self-protection framework acts as an execution environment to run applications with protection mechanisms. Adaptation policies which guide execution of running systems need to be easily integrated into the framework at any moment. One solution is to dynamically transform adaptation policies into run-time representations.

Requirement $\mathcal{R}$ 7.4 *The self-protection DSL should be easily translated into run-time representations to be integrated into the legacy self-protection framework.*

7.2 Related Work

Self-protection frameworks apply the autonomic vision to the security domain [47]. Frameworks like Jade [49, 140] or the self-defending platforms of Intel [12] enables reconfiguration of protection functionalities based on evolution of the security context, but generally without addressing the specification of the autonomic security management strategy.

In more generic policy-based management frameworks [13, 142], system execution is governed by applying predefined policies to respond to context changes – adaptation strategy specification was for instance discussed in [91, 144]. Unfortunately, these frameworks hardly considered security, with the notable exception of [142]. The specification and implementation of adaptation policies taking into account different concerns, such as tradeoffs between security and other dimensions, remains an open issue.

Advances in security context modeling also provide useful inputs for security adaptation decisions, abstracting security-related data into a formal model [10]. Ontologies are particularly helpful to structure or characterize security information [92], for instance to classify attacks [133], intrusions [143] or security context [50]. They allow better communication and reuse of the defined concepts, are extensible, and also enable to reason about some security enforcements. Unfortunately, because of the diversity of targeted domains, a unified security ontology, generic but still able to describe in depth and reason about practical aspects of security still remains to be found [34].

DSLs are another modeling approach based on specification languages addressing a particular domain. A DSL offers primitives to describe a given domain and some implementation mechanisms such as annotations for integration into a run-time [65]. Platforms like Tune [45] and Kermeta [111] provide tools to generate various DSLs and their corresponding IDEs. However, a DSL for describing security adaptations seems to be missing, along with language mechanisms for integration into autonomic frameworks.

Like our approach, the ReD (Reaction after Detection) framework enables to choose new authorization policies to counteract network intrusions [64, 59]. Based on OrBAC ontologies, ReD may support multiple types of authorization policies (e.g., integrating information flow requirements [23]) by mapping OrBAC to a security model through the definition of the corresponding ontology [50].

7.3 Main Features

To tackle the definition of security adaptation policies, the approach followed in [45, 111], where adaptation policies are specified using DSLs, is applied to the security domain, resulting in a *DSL for self-protection.* Different from traditional policy specification language, this DSL performs:

Separation of Roles of Different Actors Since various actors collaborate together in sharing this DSL, the separation of roles in the DSL improves self-management. DSL specification by an actor can be autonomically updated, and it is transparent to actors working on other aspects. Therefore, the self-management requirement ($\mathcal{R}$ 7.2) is achieved.

Application of the *Event-Condition-Action* Approach Conventional *if-then-else* approach models systems into conditions and proposes corresponding reactions. The *Event-Condition-Action* approach separates real-time events from relatively stable system states. Hence, time-critical requirements can be fulfilled through real-time events. And the *Event-Condition-Actions* approach is represented by a transition diagram which makes the system modeling intuitive ($\mathcal{R}$ 7.1).

Modeling of Multiple Dimensions for System States An extensible system state model $V = V_1 \times V_2 \times ... \times V_n$) is given in Chapter 3, various state dimensions can be easily instantiated in this model ($\mathcal{R}$ 7.3). Adaptation policies by the DSL is based on it to make trade-offs of these dimensions for final adaptation decision making.

Generation of Run-time Representation Some transformation mechanisms are proposed to the DSL which generate run-time representations relying on adaptation policies. These transformations check compatibility of policies and generate robust rules which can autonomically guide self-protection ($\mathcal{R}$ 7.4).

7.4 DSL Design Principle

7.4.1 The DSL Approach

A *Domain-Specific Language (DSL)* may be defined as "*a language tailored to a specific application domain*" [106]. DSLs present many benefits in terms of expressivity and simplicity, which led [45, 111] to specify adaptation policies using DSLs in autonomous systems. To tackle definition of security adaptation strategy, we follow the same approach, but applying it to the security domain, resulting in a *DSL for self-protection*.

Several elements led us to express self-protection policies with a DSL, instead for instance of a simple management API. A self-protection framework should be able to react to security-relevant events occurring at run-time, either in the system itself or in its environment. Specifying corresponding adaptation policies is a non-intuitive and error-prompt task which may be facilitated with a security-oriented DSL. The DSL may serve to specify adaptation policies. It may also come with some transformation mechanisms for easy integration of policies into running frameworks.

Moreover, in our current self-protection framework, adaptation strategies are purely action-based. However, higher-level strategies using objective or utility function policies are also desirable [91]. By enabling specification of governance strategies with richer types of policies, the DSL approach should allow describing self-managed security at different levels of granularity which can be refined (e.g., with notions of policy continuum [130]), and thus evolve towards greater autonomic maturity in corresponding systems.

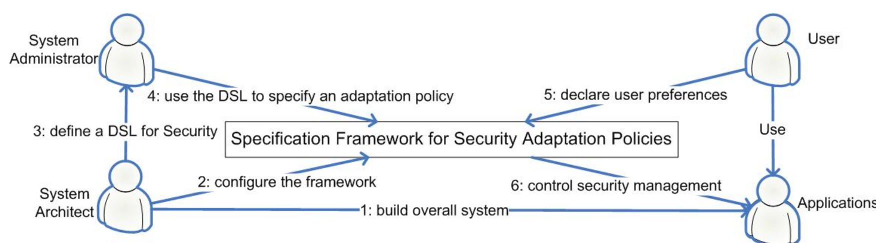

Figure 7.1: Definition and Use of Security Adaptation Policies.

7.4.2 Main Actors of Autonomic Security Management

Security adaptation policies should be viewed as high-level protection strategies that affect and guide execution, usually expressed as rules defining reactions to specific security events. Several stakeholders with clearly-separated roles may cooperate to define and use such policies (see Figure 7.1):

- The *system architect* is a designer of overall system. He implements a set of applications, configures the underlying protection framework, and defines the adaptation policy language with which the system administrator may specify security adaptation policies.
- The *system administrator* is in charge of the execution phase of systems. He has general knowledge about the system architecture and state, and specifies adaptation policies to guide behavior of the protection framework.
- *Applications* are software entities the security of which is regulated by the self-protection framework.
- *Users* specify their preferences about adaptations of system behavior from a usability perspective.

Security adaptation policies thus plays a central role in autonomic security management since they: (1) are described in a language (the DSL) designed by the system architect; (2) are specified by the system administrator; (3) may be customized by users; and (4) guide behavior of the self-protection framework. The DSL design should thus meet requirements from all stakeholders.

7.5 A DSL for Self-protection

We now describe the DSL itself which is based on the notion of *Generic Adaptation Policy (GAP)*. GAPs may be applied to different concerns such as security or energy efficiency, with possible trade-offs between them [14]. In the security case, GAPs are complemented with two taxonomies of attacks and countermeasures to specify security adaptation policies.

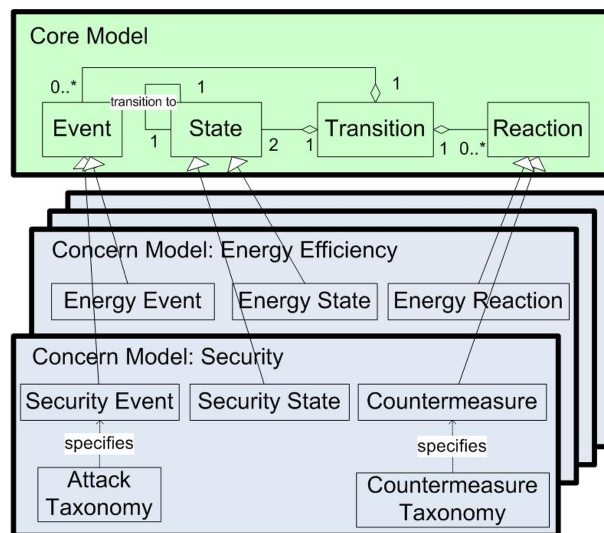

Figure 7.2: DSL Core Model.

As shown in the core model of Figure 7.2, the main concepts of our DSL are the following: the *State* class captures the status of the running system viewed from different concerns (risk level, energy efficiency, QoS, etc.); the *Event* class describes external signals which may trigger adaptations (this class may be specialized for each concern); the *Transition* class indicates the start and target states in an adaptation; and the operations to perform – captured by the *Reaction* class.

For each concern, different *concern models* abstract system features from the concern perspective by refining those classes. As our adaptation policies for self-protection specifically address the security concern, events and reactions are respectively specified by the *attack* and *countermeasure taxonomies* (described next). However, other concerns like energy efficiency may also be integrated into the adaptation policy using specific taxonomies.

7.5.1 Taxonomy of Attacks

This taxonomy classifies potential attacks or security-relevant events calling for a modification of system protection settings. A sample attack taxonomy is shown in Figure 7.3, but others may also be used as well [133], the DSL not being tied to a specific taxonomy or ontology. Attacks may be classified according to confidentiality, integrity, and availability (CIA) security objectives. In turn, each class of attacks contains a set of specific potential attacks that may occur in the system. For instance, the *Packet Flooding* attack that disturbs network by sending a large number of packets may be part of the *DoS* attack class threatening availability. To specify the adaptation policy, the system administrator identifies the most relevant attacks in the taxonomy. The corresponding security events may then be used in the GAP to trigger adequate reactions.

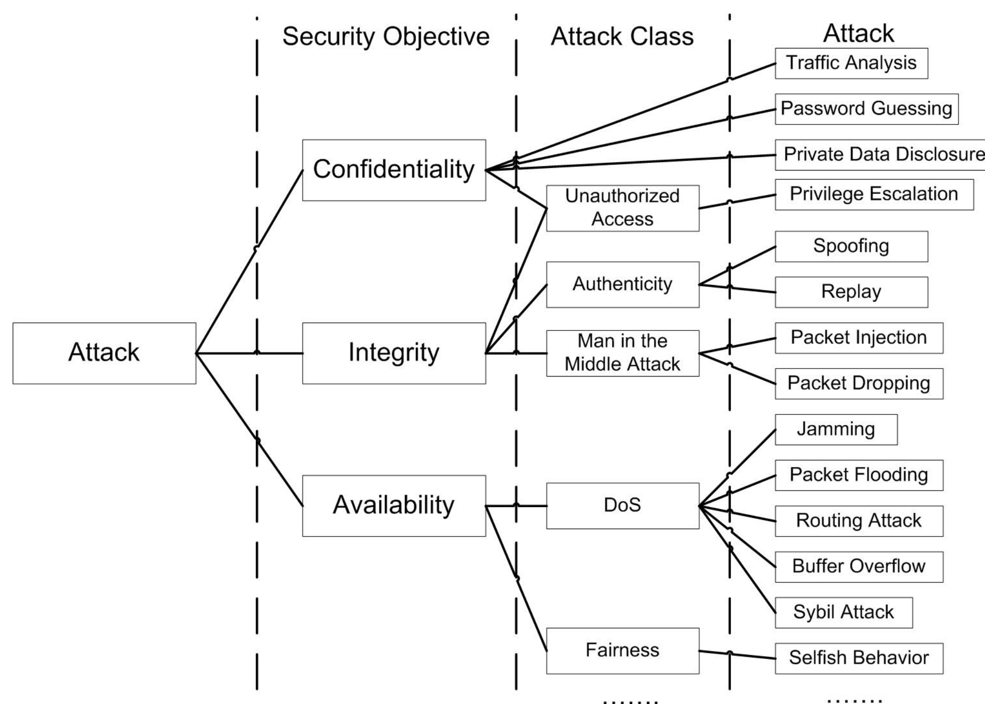

Figure 7.3: Attack Taxonomy.

7.5.2 Taxonomy of Countermeasures

The countermeasure taxonomy classifies reactions applicable to respond to security-sensitive events. A sample taxonomy is shown in Figure 7.4. As for the attack taxonomy, security functions are structured according to the CIA security objectives. For example, the *Strengthen Encryption* countermeasure class is classified as a reaction improving confidentiality. To specify the adaptation policy, based on the identified attack and security objective to guarantee, the system administrator will select from the taxonomy the most appropriate countermeasure to trigger – a more automated approach is for instance described in [57].

7.5.3 Generic Adaptation Policies (GAP)

Security adaptation policies should be intuitive for user-friendly specification, which is not the case of hard-coded *if-then-else* rules. We thus prefer to use *event-condition-action* policies, instantiating the core model presented previously to specify states, events, transitions, and reactions to describe possible adaptations. The result is captured by the notion of GAP, formally defined as a tuple $[V, E, AT, gap]$ where:

- $V \subseteq V_1 ... \times V_i \times ... \times V_m$ is the *state space* of running systems. Each V_i represents a system state dimension which has a finite set of values and is identified by the system architect, e.g., energy efficiency, risk level, etc. The system state is then given by a

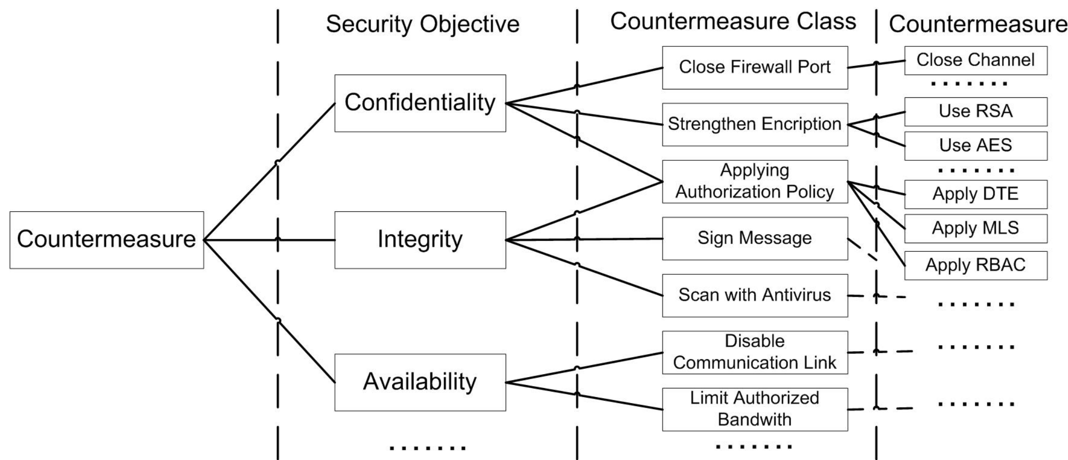

Figure 7.4: Countermeasure Taxonomy.

tuple $[v_1, ..., v_i, ..., v_m]$ of that space, where $v_i \in V_i$ is the value of a state.

- E is a finite set of *events* e that may occur in the system according to system evolution or context change. Events e are mainly related to monitoring infrastructures like intrusion detection systems, and will trigger adaptations. The events are partially derived from the attacks defined in the attack taxonomy.

- AT is a finite set of *adaptation reactions* at that can be applied to the running system. Different types of reactions may be performed, and are described by the system administrator. In the context of self-protection, the reactions are derived from the countermeasure taxonomy.

- The $gap : V \times E \to 2^{V \times AT}$ function maps a current state v and received event e to a set $gap(v, e)$ of proposed reactions and foreseen destination states.

Specifying a GAP then amounts to describing transitions in terms of adaptation reactions and destination states from a set of initial states and events. Such a specification may be compactly represented using transition diagrams, as shown in the following example.

7.5.4 An Example of GAP Specification

We consider a system with only one state dimension, the risk level, which represents the vulnerability of the system, classified in the following 4 levels: $V = V_1 = \{$`very hostile`, `hostile`, `neutral`, `friendly`$\}$.

Two attacks are considered from the attack taxonomy, *Packet Flooding* and *Privilege Escalation*: $E = \{e_0, e_1\}$ (see Table 7.1). The *Packet Flooding* attack belongs to the *DoS* class compromising availability. The *Privilege Escalation* attack is part of the *Unauthorized Access* attack class weakening confidentiality and integrity.

Event ID	Event Name	Event Type
e_0	Packet Flooding	DoS
e_1	Privilege Escalation	Unauthorized Access

Table 7.1: GAP Event Specification (Attack Taxonomy).

Reaction ID	Reaction Name	Reaction Type	Protection Effect
r_0	Close Channel	Disable Communication Link	+++
r_1	Apply DTE Policy	Apply Authorization Policy	++
r_2	Use RSA	Strengthen Encryption	+

Table 7.2: GAP Reaction Specification (Countermeasure Taxonomy).

We consider three reactions from the countermeasure taxonomy, *Close Channel* (r_0) for the availability security objective, and *Apply Domain Type Enforcement (DTE) authorization policy* (r_1) and *Use RSA* (r_2) for the confidentiality and integrity objectives: $AT = \{r_0, r_1, r_2\}$ (see Table 7.2). The table also indicates a subjective assessment of the protection strength of each countermeasure, to give an idea of its effectiveness to decrease the risk level.

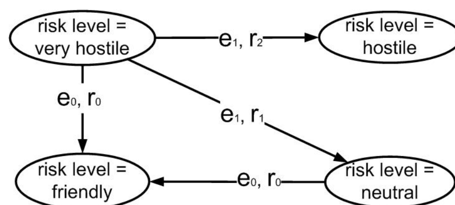

Figure 7.5: A GAP Sample Specification.

The system administrator then may define the *gap* function using the transition diagram shown in Figure 7.5. For instance, transition (e_1, r_2) from state ($risk =$ `very hostile`) to state ($risk =$ `hostile`) means that if a *Privilege Escalation* attack is detected in an already very hostile environment, encrypting communications may help decrease somewhat system vulnerability.

7.6 From DSL to Run-Time Representation

In this section, we show how the adaptation policies may be converted into high-level autonomic guidelines executable within a self-protection framework.

GAPs describe adaptation policies using events, states, etc. But from an implementation perspective, GAP policies are neither dependable nor efficient for run-time control since: (1) for each situation, more than one adaptation path may be proposed, making

selection difficult and time-consuming; and (2) specified policies could be incomplete by construction, automated policy checking being complex and CPU-intensive due to the dimension of the state space. A more compact representation of adaptation policies called *Autonomic Adaptation Policies (AAP)* is thus defined to improve both dependability and efficiency. Some mechanisms are also proposed to translate GAPs to AAPs.

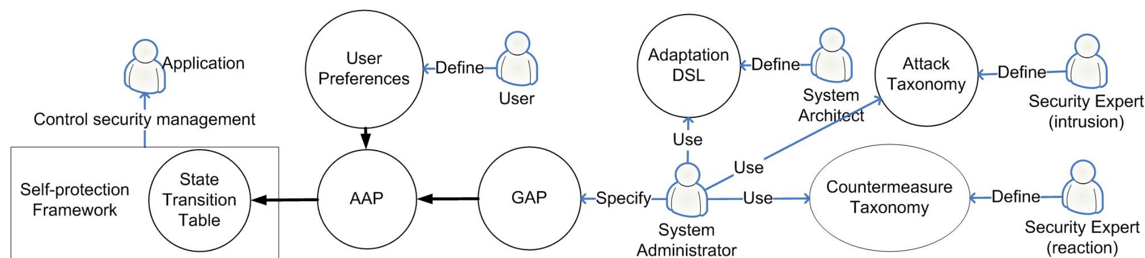

Figure 7.6: Adaptation Policy Transformation.

As shown in Figure 7.6, the main steps of the DSL life-cycle are the following:

1. A security expert of intrusions defines the attack taxonomy;
2. A security expert of reactions defines the countermeasure taxonomy;
3. The system architect defines the syntax and semantics of the DSL in specifying the set of states V, the set of events E and the set of reactions AT;
4. The system administrator specifies the set of transitions of the GAP based on the previous sets of states, events, and reactions;
5. Users define their preferences for security adaptations, e.g., prefer security over energy efficiency;
6. The translation mechanisms integrate the user preferences into the GAP, generate an AAP, and transform the AAP to a *state transition table* as run-time representation of the security adaptation policy.

In this process, each actor works in his own domain, translation mechanisms weaving together their different inputs into a run-time representation. Roles are clearly separated, policy specification and checking becoming independent tasks.

7.6.1 Autonomic Adaptation Policies (AAP)

For proper autonomic decision-making, a run-time adaptation policy should be both *complete* and *deterministic*. For any execution situation, at least one adaptation should be proposed to maintain execution continuity. A GAP should thus satisfy:

Definition 7.1 (Completeness) *A GAP is complete if it leaves no situation unspecified, so that at least one adaptation reaction is associated to each event-state pair, i.e.,* $\forall (v,e) \in V \times E, |gap(v,e)| \geq 1$.

Besides, to avoid the ambiguity of adaptation decisions, different adaptations for the same conditions should not be possible. This property makes the security adaptation strategy highly sensitive information: knowledge how the system will react upon attacks could be used by intruders to defeat counter-measures more easily. We thus assume security adaptation strategies to be well protected. A GAP should thus satisfy:

Definition 7.2 (Determinism) *A GAP is deterministic if no more than one adaptation may be realized in a given situation, so that at most one reaction is proposed for each event-state pair, i.e.,* $\forall (v, e) \in V \times E, |gap(v, e)| \leq 1$.

We define an AAP as a GAP fulfilling the two previous properties. The separation of roles of the different stakeholders is thus effectively achieved since the system administrator only has to define the GAP specification, while property verification and GAP-AAP translation will be performed by DSL compile-time and run-time mechanisms.

7.6.2 A Sample Translation

The following example illustrates how the translation from GAP to AAP may be performed.

7.6.2.1 GAP Definition

We consider a pervasive system where security adaptation policies $GAP = (V, E, AT, gap)$ are defined as follows:

- $V \subseteq V_1 \times V_2$ represents the state of the system with 2 dimensions: the risk level (V_1) and the energy consumption (V_2). The *risk level* captures the vulnerability of the whole system, classified into 4 levels. The *energy consumption* is related to the device energy efficiency, with also 4 levels. Thus V_1 = {`very hostile`, `hostile`, `neutral`, `friendly`}, and V_2={`critical`, `high`, `normal`, `low`}. We consider only 5 states as detailed in Figure 7.7.

- E is the set of alarms raised by an intrusion detection system. To simplify, we only consider two events: $e_{\texttt{attackDetected}}$ when an attack is detected, and $e_{\texttt{returnSafe}}$ indicating the return to a safe state after application of a countermeasure. Then $E = \{e_{\texttt{attackDetected}}, e_{\texttt{returnSafe}}\}$.

- The reactions AT are defined as applying authorization policies into the system. Four policies p_1, p_2, p_3, p_4 are used as countermeasures. In using $\texttt{Permission}(p_i)$ to represent the set of all permissions of the policy p_i for a fixed set of subjects and objects, we define $p_i \succ p_j$ in terms of security strength which means that $\texttt{Permission}(p_i) \subset \texttt{Permission}(p_j)$. We then make the assumption that the policies are ordered as $p_4 \succ p_3 \succ p_2 \succ p_1$ in terms of the security strength which means that $\texttt{Permission}(p_4) \subset \texttt{Permission}(p_3) \subset \texttt{Permission}(p_2) \subset \texttt{Permission}(p_1)$. We also assume that the policies are ordered as $p_2 \succ p_4 \succ p_3 \succ p_1$ in terms of the energy consumption.

- *gap* specifies the allowed adaptations, captured as transitions between states based on received events. Adaptation rules include the initial state, upcoming events, actions to perform, and destination state for each adaptation. Possible evolutions of the system may be represented by a transition diagram as shown in Figure 7.7.

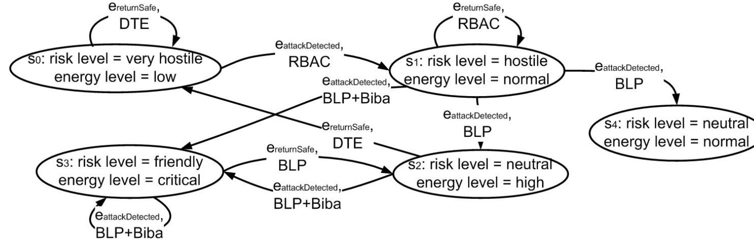

Figure 7.7: A Typical Generic Adaptation Policy.

For instance, the transition from state s_2 ($risk = \texttt{neutral}$, $energy = \texttt{high}$) to state s_2 ($risk = \texttt{friendly}$, $energy = \texttt{critical}$) on event $e_{attackDetected}$, with reaction p_4 means that if the risk level is `moderate`, the energy consumption already `high`, and that an attack is detected, the authorization policy p_4 will be applied, driving the system in a state where the risk is decreased to `friendly`, but the energy consumption being increased to `critical`.

7.6.2.2 GAP → AAP Translation

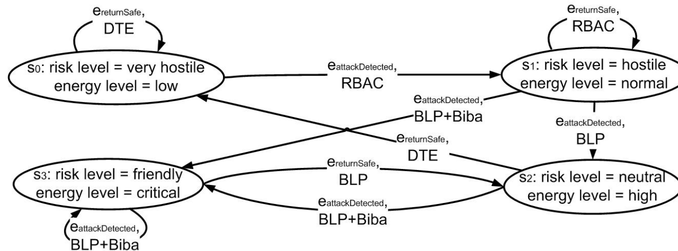

Figure 7.8: GAP fulfilling Property 1.

Note that there are no adaptations associated to the state s_4 ($\texttt{risk} = \texttt{neutral}, \texttt{energy} = \texttt{normal}$) as shown in Figure 7.7, i.e., system execution will be blocked upon entering this state. Such states and the corresponding transitions should be eliminated from the GAP in order for Property 1 to hold. The system is then thus described only by the 4 states shown in Figure 7.8, at least one adaptation being associated to each state-event pair.

The specified policy may be initially non-deterministic, as for same state-event pair several adaptations may be proposed bringing the system to different destination states. For instance, in state s_1, on event $e_{attackDetected}$, the system may apply either p_3 or p_4 policies, respectively driving the system to states s_2 and s_3. This makes adaptation decisions difficult.

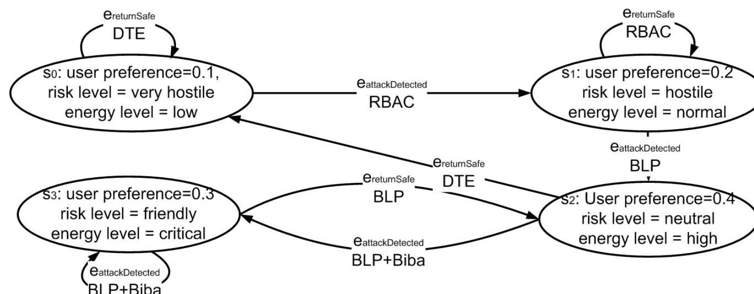

Figure 7.9: GAP fulfilling Properties 1 and 2.

To retrieve a deterministic policy, we use a utility function $uFun$ to assess the utility of each destination state, and trim the states with low utilities. A utility value is thus attached to each state based on the user preferences. In the example, sample utilities of the 4 considered states are $uFun(s_0) = 0.1$, $uFun(s_1) = 0.2$, $uFun(s_2) = 0.4$, $uFun(s_3) = 0.3$. Only transitions with the highest utility values are kept, yielding the policy shown in Figure 7.9, which now also satisfies the determinism property. In the case where several transitions lead to the same highest utility value, one transition will be randomly selected.

The resulting GAP satisfies the two properties defined in the previous section, thus is as an AAP (see Figure 7.9). A *state transition table*, directly executable in a self-protection framework, can be derived from this AAP. This table has been integrated in our self-protection framework in order to propose run-time unique adaptation reactions.

7.7 Summary

This chapter illustrated a DSL to describe security adaptation policies for the self-protection framework. The DSL is based on the ECA approach, and on two taxonomies respectively of attack and countermeasure. Different actors are separated in the policy specification process: the system administrator uses GAPs to specify policies intuitively. GAPs are then checked to guarantee run-time dependability, and translated into AAPs, suitable for integration at run-time into legacy self-protection frameworks (trade-offs between security and other concerns may notably guide the refinement process).

Currently, a DSL framework called yTune is under development, including an editor and a parser for specification and checking of different DSLs. yTune is a meta-language for DSL definition and implementation. Ongoing work is focused around implementing the described refinement mechanisms for the self-protection DSL in the yTune parser, and coupling the DSL toolchain with the ASPF self-protection framework. In the future, we also plan to enhance the DSL with more complete and realistic taxonomies for security attacks and countermeasures, for instance through dedicated security ontologies which may be coupled with the corresponding security components to detect intrusions and perform reactions.

Part III

Validation

Chapter 8

VSK Validation

The implementation of the VSK design is described in this chapter. VSK is implemented as an emulator under *Linux* by means of the *Think/Fractal* framework. Evaluation of the implementation illustrates performance improvement of VSK comparing to other OS architectures.

8.1 VSK Implementation

As described in Chapter 6, a terminal-side system of the end-to-end self-protection framework is built on the VSK OS. In order to apply the component-based approach and accomplish the described dynamical reconfiguration functionalities of VSK, the terminal-side system is specified in using the *Fractal* component model [41] and implemented in using the *Think* framework [72, 20].

8.1.1 Overview of Implementation Framework

Fractal: A hierarchical and reflective component model, *Fractal* [41], is used to design, implement, deploy, and manage software system of the VSK design. *Fractal* components are both design- and run-time entities: acting as unit of encapsulation, composition, and reconfiguration. They implement server interfaces as access points to services, while functional requirements are expressed by client interfaces. Components interact through bindings between client and server interfaces. *Fractal* also defines standard interfaces to control internal structure of a component at run-time, e.g., adding or removing sub-components or bindings. The software architecture of VSK is then given by its hierarchy of bound components.

Think: A framework of *Fractal* on the *C* language is provided which is called *Think* [72, 20]. Using this framework, an OS architect can build a system from components without being forced into a predefined kernel design. This flexibility makes it particularly easy to implement different components of the VSK architecture. The *Think* reconfiguration framework [117] is particularly helpful to implement dynamic reconfiguration mechanisms.

nupse [103], the compiler of the *Think* framework, is a cross-compiler by which we can compile, test or debug an OS kernel on one hardware platform and install it into another platform.

Dynamic Reconfiguration: The *Think/Fractal* framework also defines standard control interfaces (called controllers) to observe and manipulate internal structure of a component at run-time. In particular, a component may implement: a *Component Identity (CI)* to give access to its server interfaces; a *Binding Controller (BC)* to bind its client interfaces; a *Attribute Controller (AC)* to query and change attribute values; and a *Content Controller (CC)* controller to list, add, and remove sub-components.

Creating a binding from a client interface `i` of a component `M` to a server interface `j` of a component N can be achieved through the following steps:

1. finding the `CI` controller of `M` through the `CC` controller of the parent component of `M`;
2. getting the interface `i` of `M` through its `CI` controller;
3. getting `BC` of `M` through its `CI` controller;
4. unbinding existing binding to the interface `M.i` (the interface `i` of the component `M`);
5. getting the `CI` controller of `N` through the `CC` controller of its parent component;
6. getting the interface `N.j` through its `CI` controller;
7. creating the binding from `M.i` to `N.j` through `M.BC`.

Global Extension: An *Architecture Description Language (ADL)* of the *Fractal* component model defines the architecture of the component-based system to build. With the global extension mechanism, *Think* allows defining a component type by extending an existing definition. A global extension is simply an ADL definition where the original definition is automatically extended, the definition matching some expressions is replaced by another. This mechanism makes it possible to transform an architecture in a global way, by abstracting some aspects of concern (into extensions) from system architecture definition.

```
component <main.executionSpace.*> component * {
  provides Fractal.API.ComponentIdentity ci
}
```

Figure 8.1: Global Extension.

For example, for a given ADL description, it is easy to generate a system where all the components implement a *Component Identity (CI)* controller. In Figure 8.1, the global exension inserts this controller to all the sub-components in `main.executionSpace`.

Implementation and Evaluation Context: In the purpose of simplifying the implementation and evaluation, our VSK prototype is realized as an emulator by the *Fractal/Think* framework based on a *Linux Ubuntu 9.04*. This implementation is a proof-of-concept to illustrate its feasibility, it can be easily exported to other platforms with the help of the *nuptse* cross-compiler. Several existing functionalities (communication or multi-thread) in *Linux* can be profited for implementation and evaluation rather than developing platform-specific functionalities. Alternatively, since performance benchmark for embedded mobile terminals is not easy to realize, we uses existing tools of *Linux* for the performance evaluation of VSK.

In the case where we want to implement VSK on other hardware platforms such as *ARM* or *AVR*, we only need to develop a thread management component and a file system component for the specific platform and replace *pthread.h*, *stdio.h* function invocations by interface invocations to these components. Platform-specific communication mechanisms should also be developed if communication functions are needed. Several experiments, such as *WiFLY* [146] on 8bits μC *ATM128L* and Sense&Sensitivity on *MSP430* for wireless sensors, *CRACKER* [95] on *Nokia 800* Internet Tablet of *ARM9* in the context of Smart Home, have already shown the possibility and efficiency of code exportation from an emulator to a specific platform.

8.1.2 VSK Architecture Overview

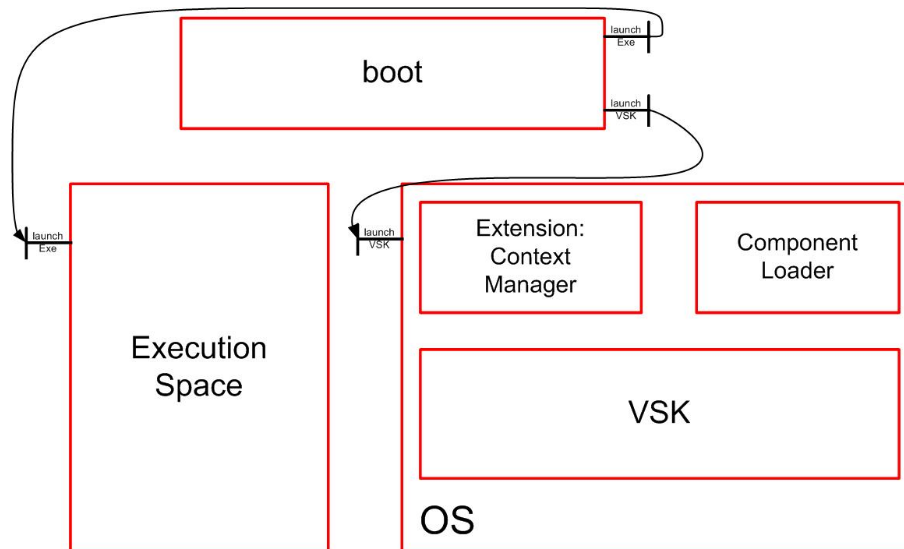

Figure 8.2: VSK-based System Overview

Figure 8.2 shows the overview implementation architecture of VSK based on the *Fractal/Think* framework. The main components are:

- a *boot* component that installs VSK and launches the *execution space*;
- the *execution space* in which applicative components are installed;
- the VSK component which is the kernel to manage the *execution space*;
- a *component loader* which enables to dynamically load new components;
- some *extension modules* that extend VSK with additional functionalities such as context-awareness and session management.

Therefore, an OS consists of VSK together with a *component loader* and several extension modules. Components in the *execution space* are applications which run above this OS.

8.1.3 Execution Space

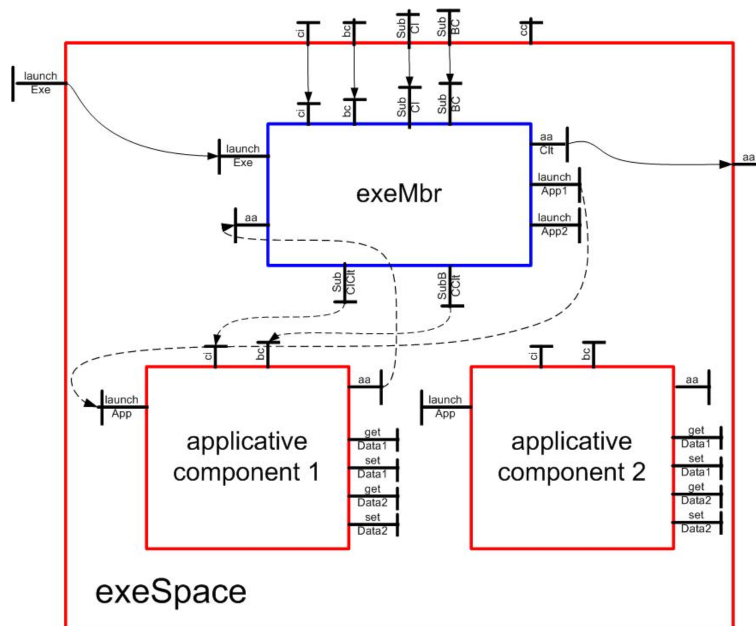

Figure 8.3: Execution Space

The *execution space* (see Figure 8.3) can be considered as a running environment for applicative components. Behind the execution of these components, access control is performed through an authorization check interface called *access authorization (aa).* Moreover, an *execution membrane* is automatically implemented in the *execution space* to administer run-time behaviors of these applicative components.

Applicative Components: All applications are encapsulated as components in the *Think* framework, these components are thus called applicative components. Inter-components communication is realized via interfaces and bindings between components. With the help of the global extension, some specific interfaces or controllers can be inserted automatically to a part of applicative components during the compilation. A *launch execution (LaunchExe)* server interface is used to start the execution of an applicative component. An *authorization access (aa)* client interface serves for access request validation with VSK. A *binding controller (bc)* and a *component identity (ci)* are used to achieve dynamic reconfiguration. Since an applicative component in the *execution space* needs to realize all these functions, a *LaunchExe*, an *aa*, a *bc* and a *ci* interfaces are automatically added to each applicative component of the *execution space* during the compilation by the global extension mechanism.

Execution Membrane: In the purpose of dynamic reconfiguration of the *execution space* which is a composite containing several sub-components, a specific component, *execution membrane (exeMbr)*, is used to manipulate the internal structure of the composite *execution space* at run-time. The membrane mediates external invocations to its sub-components through interfaces and controllers. Furthermore, the membrane can start or stop its sub-components and create or remove bindings between them. For the membrane of the *execution space*, it not only mediates access requests from the *execution space* to VSK for access validation but also transmits and realizes kernel commands like *creating a binding* or *removing a component*. The same membrane concept is also applied to other composites such as *Attribute Manager* and *Rule Manager* in the kernel to cope with the dynamic reconfiguration of these two composites.

8.1.4 Control Plane Implementation

As described in Chapter 3, the *execution space* is controlled by a *control plane*. VSK achieves this *control plane* which mainly performs two functionalities as: the management of tasks and validation of access requests (see Figure 8.4). VSK includes a *Virtual Kernel (VK)* and an *Access Control Monitor (ACM)*. VK contains a *dispatcher* that supervises the *execution space* and transmits requests to different modules in VSK. VK also has a *reconfiguration manager* which reconfigures the *execution space* based on decision taken in ACM. ACM achieves attribute-based authorization validation (see Chapter 6)

Dispatcher: For the task management, the *dispatcher* changes the kernel from user mode to kernel mode, organizes all received requests, and transmits requests one by one to ACM. When the *dispatcher* treats a request, it runs to completion for the request. Therefore, no concurrence may be induces by the *dispatcher*. If a transmitted request is validated by ACM, the *dispatcher* invokes the *reconfiguration manager* to create a binding for requested access in the *execution space*. Otherwise, it returns the negative response and starts another request.

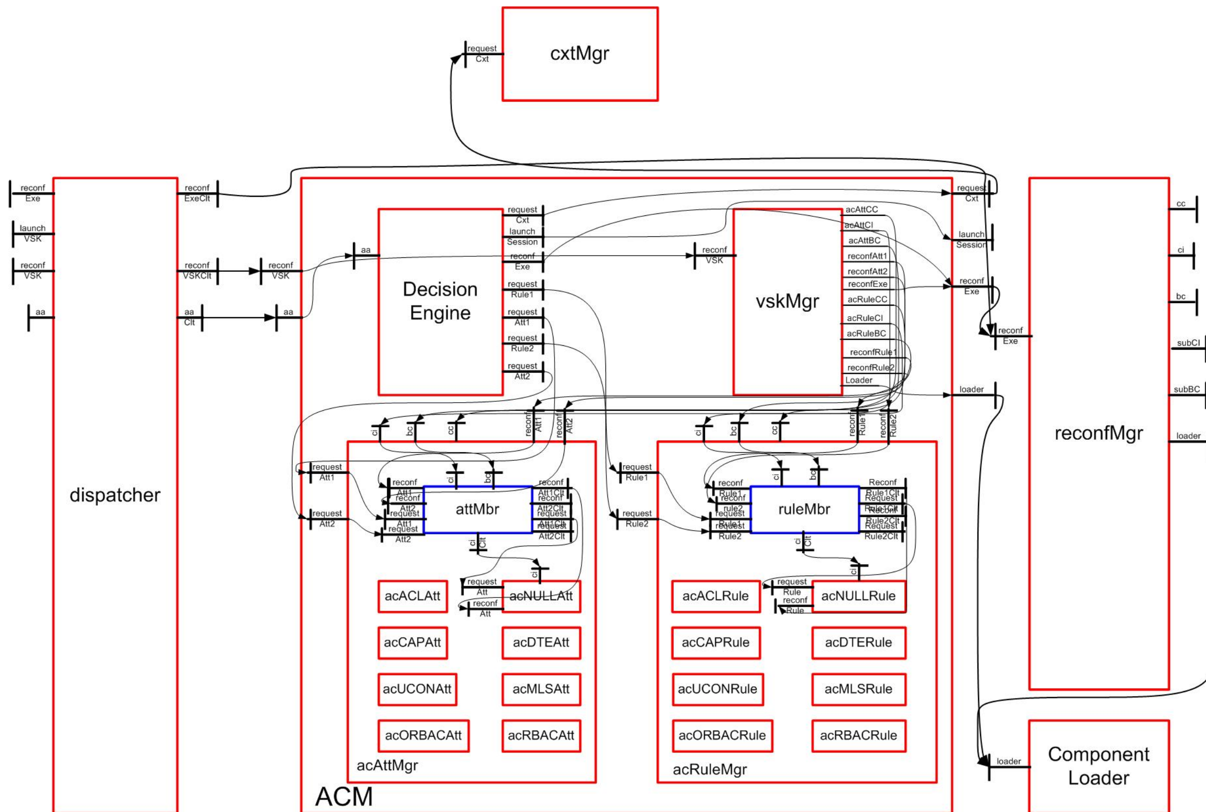

Figure 8.4: VSK Kernel

ACM: The ACM component plays the role of the security kernel [17] mainly containing four modules: an *Access Control Attribute Manager (acAttMgr)* from which access attributes can be queried; an *Access Control Rule Manager (acRuleMgr)* to get access rules depending on attributes; a *Decision Engine* that makes access decision based on attributes and rules; and a *VSK Kernel Manager (vskMgr)* that makes ACM dynamically reconfigurable.

- the *Access Control Attribute Manager (acAttMgr)* returns attribute values in the response to access requests. As different authorization policies can be implemented in the kernel, a policy-neutral interface is proposed for acAttMgr;
- the *Access Control Rule Manager (acRuleMgr)* contains authorization rules. An access control request with associated attribute values is transmitted to this component in order to find out whether a corresponding rule exits;
- the *Decision Engine* makes a final decision based on attributes and rules. Since our prototype simultaneously supports two authorization policies, this component is adopted with two access control channels. One policy is said to be "installed" if it is bound to one channel. One channel is considered as suspended if a *null* policy is

attached to it. If two policies are used in the kernel, a policy-combining algorithm is applied for the combination of two access decisions. We implement a *deny-override* algorithm where an access permission should be granted by both two policies;

- the *VSK Kernel Manager (vskMgr)* dynamically administers ACM. Various authorization policies can be loaded (by the *component loader*) and installed in ACM. vskMgr can enable or disable one access channel or replace one authorization policy by another on one access channel.

Six basic authorization policies (ACL, capability-based, DTE, MLS-BLP, RBAC, OrBAC) and a null policy which represents the absence of authorization policy are implemented. Corresponding attribute component and rule component of each policy are maintained in acAttMgr and acRuleMgr. New policies including attributes and rules can be loaded and installed into ACM by the *component loader*. In the prototype, we implement two channels to VSK which means that two access control policies can be simultaneously installed. The final decision combines the decisions of these two policies in using and policy-combining algorithm (in fact the *deny-override* algorithm) which is embedded in the *decision engine* component.

Reconfiguration Manager: When an access request is validated by ACM, the *reconfiguration manager* creates the corresponding binding in the *execution space*. Moreover, it registers created bindings. For the access control model revocation, all associated bindings are removed by the *reconfiguration manager* component.

8.1.5 Authorization Policy Implementation

In terms of authorization policies, we assume that a simple embedded OS has at maximum 10 threads as subjects and 60 system calls as objects. In order to introduce the same evaluation conditions for different authorization models, we choose as authorization policy, a policy where all access are authorized for the 10 subjects to the 60 objects with *read* and *write* access operations. The six basic authorization models (ACL, Capability-based, DTE, MLS, RBAC, ORBAC) and a combined models (Lipner - BellLapadula with Biba) are used to specify all these permissions (that is, using different access control models to specify the same policy) with G-ABAC and implemented as *Think* components for VSK.

However, the chosen "all granted" policy is a specific case. A complete evaluation should take into account all possible authorization policies for each access control model. Since the evaluations are realized on the *Think* platform which compiles each policy in forme of component and loads it into the VSK OS, such a complete evaluation seems complicated.

8.2 VSK Evaluation

8.2.1 Authorization Overheads of ACM

We evaluate the terminal-side system built on VSK in this chapter and provide an end-to-end benchmark of the self-protection framework which integrates VSK with ASPF in the next chapter. All measurements were performed on a 2.7GHz *DELL OptiPlex 740 desktop PC* with *Linux/Ubuntu* 9.04 and 1GB of RAM. We start the evaluation on the ACM module in order to find out the authorization overhead with different access control policies installed in VSK. An authorization permission is computed on four steps:

1. getting subject and object attributes;
2. getting context information if the authorization policy is context-aware;
3. finding access rules and computing authorization permissions;

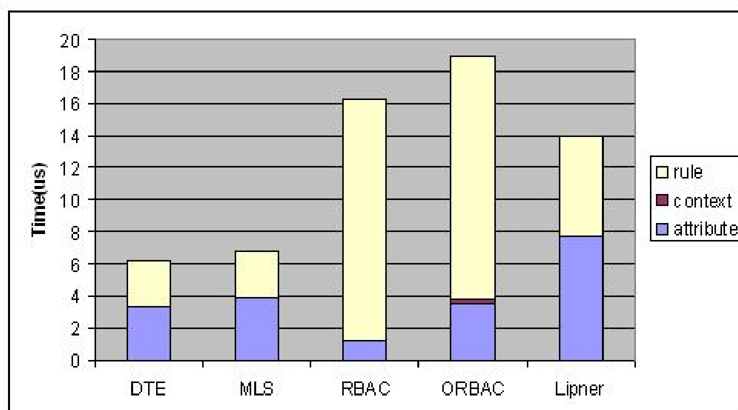

Figure 8.5: Authorizations Overhead of ACM

In sending all the 1200 different access requests (we have 10 subjects, 60 objects with 2 operations) to the ACM module, we measure average permission computing duration as the overhead of authorization for the nine access control policies. Since ACL and capability-based policies directly use their identities for authorization, a huge subject-object-operation table is maintained in ACM which should contain permissions of all subjects to all objects with *read* and *write* operations (we build all the permissions to create an equivalent evaluation environment for different authorization policies as described in Section 8.1.5). Their authorization overheads are respectively `1273us` for ACL and `1271us` for the capability-based policy which are significantly higher than others and do not appear in Figure 8.5.

Among the other authorization policies, as shown in Figure 8.5, the authorization overheads are in the order of `10us` since DTE regroups subjects and objects to domains

and types and it requires to go through the rule table to find the permission for a given domain-type pair; MLS needs security levels of both subjects and objects to perform a domination comparison; RBAC uses roles to represent users' privileges of access; OrBAC integrates context information into authorization rules with an overhead of context information acquirement. Furthermore, in the purpose of expressivity, Lipner [101], a BLP-Biba combined policy, is proposed to apply two authorization policies at the same time (thanks to the two channels of VSK) in order to express policies which can not be described by one policy.

Results show that the authorization overheads of the combined policies are bigger than a basic policy since two validations should be achieved through two access channels. But all these overheads are much smaller than the conventional ACL or capability-based policies thanks to the separation of access attributes from basic access elements (subject, object, action, and context).

8.2.2 Authorization Validation and Enforcement Overheads of VSK

VSK contains not only ACM but also VK which manages the switch of modes (user mode and kernel mode) and the reconfiguration of the *execution space* to enforcement an access decision. Once an access request is issued from the *execution space*, the *dispatcher* firstly copes with concurrency and then passes control to ACM for authorization validation. If ACM grants the request, the *reconfiguration manager* is launched to create the binding for the requested access in the *execution space*. In order to construct the same evaluation condition for different access control policies, we build granted permissions for all the subjects to all the objects.

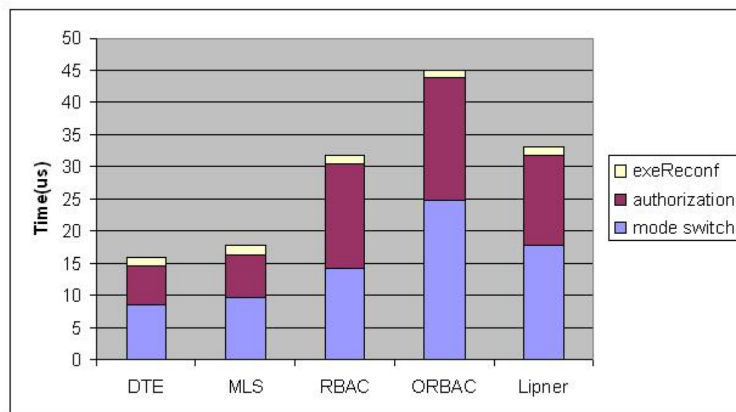

Figure 8.6: Authorization Validation and Enforcement Overheads of VSK

In sending 1200 different access requests to VSK which ensures binding creation for each request, the average overheads are shown in Figure 8.6 which are in the order of `10us`. We figure out that the mode switch overheads depend on the complexity of authorization policies, e.g., OrBAC has supplementary context module for the acquirement of context

information. In terms of the combined policies, the *mode switch* overheads are the sum of individual overheads. The authorization overheads are explained in the previous section. At the end, the *execution space reconfiguration (exeReconf)* overheads may be ignored comparing to the *mode switch or authorization* overheads since the reconfiguration mechanism by *Think* is significantly efficient. The overheads by the different policies illustrate the fact that the structure of authorization policies may affects its access validation and enforcement. But all these overheads remain acceptable for the kernel level authorization.

8.2.3 Comparison with Microkernel

Since VSK is designed as an OS kernel, it is compared to the microkernel architecture in this section. The philosophy underlying the microkernel architecture is to maintain only absolutely essential core OS functions in the kernel and remove less essential services and applications to the user mode. An extreme example is exokernel which contains only the access control and concurrency management services in the kernel. Within our case, since the access control function is implemented in the kernel, microkernel or exokernel does not make any difference.

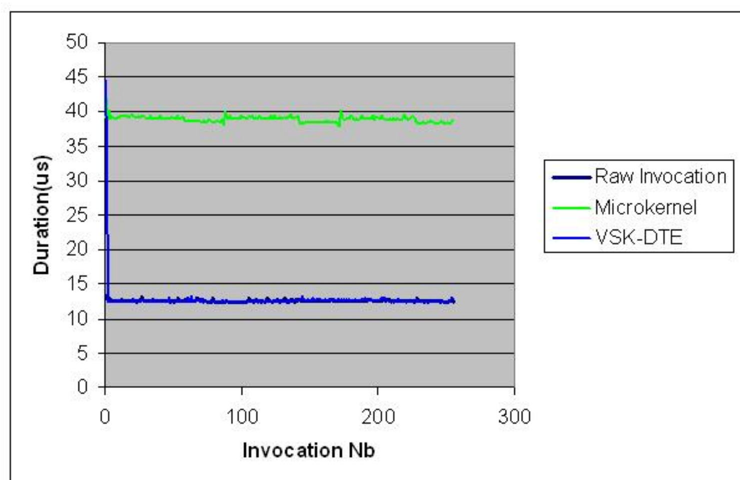

Figure 8.7: VSK vs Microkernel

Three prototypes respectively: raw invocation without kernel control, microkernel equipped with a DTE policy, and our VSK with the same DTE policy are implemented by the *Think* framework. We measure average invocation durations in order to find out the kernel overheads in various OS architectures. As shown in Figure 8.7, the duration of a raw invocation without protection is about `12.5us`. With the microkernel, since each access needs to be controlled by the kernel, its duration is risen to `39us`. The main difference between VSK and the microkernel architecture is run-time support for system reconfiguration. In a microkernel, authorization hooks are statically bound into a reference monitor to intercept each access request, inducing high access control overheads. But VSK applies

the *one-time check* mechanism where once an access is checked, no more control will be realized until the next kernel update. In the figure, VSK-DTE has one access overhead for its first invocation and remains as raw invocations after.

This performance gain might be explained by the more lightweight and dynamic VSK architecture which enforces access decisions in a dynamic manner. Additionally, this kernel also brings benefits like dynamic authorization policy reconfiguration or permission revocation management introduced in the next section.

8.2.4 Reconfiguration Overhead of VSK

With this benchmark, we compute the reconfiguration overhead for changing attributes (e.g., assignment of a new role), rules, and authorization policies. A reconfiguration in VSK is composed on three steps:

1. reconfiguring authorization components in the kernel like replacing one access attribute component by another;

2. initializing the reconfigured components;

3. revoking outdated access decisions by removing some existing bindings in the *execution space*.

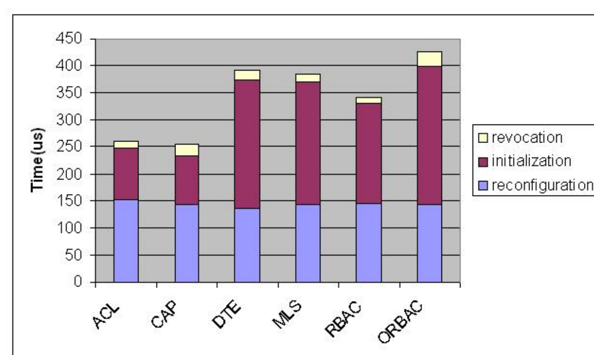

Figure 8.8: Access Control Attribute Reconfiguration

For the six basic authorization policies, their average overheads of 1000 reconfigurations are calculated. The overhead of a combined policy is the sum of two individual policies. As shown in Figure 8.8, for the reconfiguration of attributes, the reconfiguration overheads of replacing an attribute component by another seem to be identic. But the initialization overheads depend on the complexity of each policy, and the revocation overhead is much lower. The total overheads of attribute reconfiguration remain similar among different policies which are approximately `300us`.

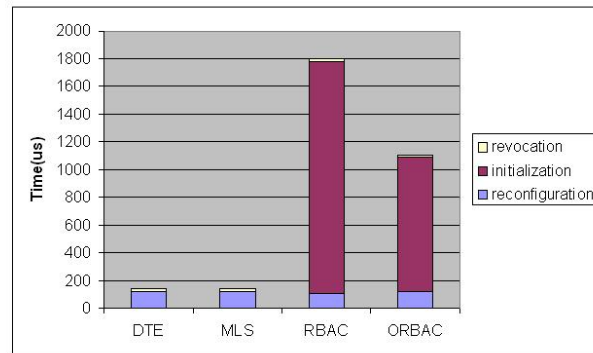

Figure 8.9: Access Control Rule Reconfiguration

In terms of the rule reconfiguration, since the ACL and capability-based policies contain a huge subject-object-operation rule table, their rule initialization durations are respectively `22439us` and `22521us` which are much higher than the other policies and are not presented in Figure 8.9. Within the other policies, RBAC has a complexity of role-permission relation and OrBAC has a possibility of multiple organization/context. Therefore, their initialization overhead are significantly higher than others which are respectively `1800us` and `1120us`. Other policies (DTE and MLS) have a simple structure which thus result in lower overheads about `160us`.

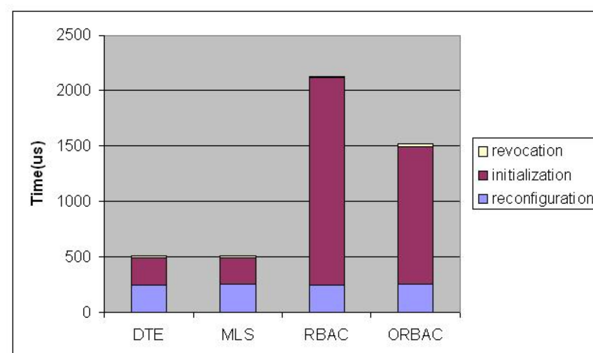

Figure 8.10: Access Control Policy Reconfiguration

The policy reconfiguration takes into account the reconfiguration for both attributes and rules. Its overhead can be viewed as the sum of the two overheads.

8.2.5 Kernel Occupation Rate

Our proposed architecture is named as a *virtual* security kernel in the sense that it remains hidden during most of the execution time with the help of the *one-time check* mechanism.

In order to show the efficiency of the *virtual* architecture, we define a matric - *Kernel Occupation Rate (KOR)* - which is the rate of kernel executing time over total running duration.

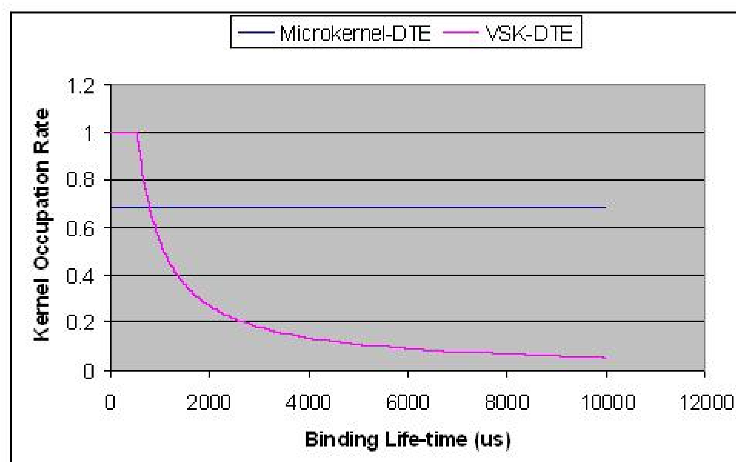

Figure 8.11: Kernel Occupation Rate between Microkernel and VSK

In repeating an invocation of `12.5us`, the microkernel occupation rate is fixed to `68.4%` as en average there is a kernel access control overhead of `39.5us` by invocation. For VSK with the *one-time check* mechanism, once the binding is created, no more controls are needed. Its kernel occupation rate depends on the binding life-time. A binding will be removed when the access decision is revoked due to the authorization policy update. As shown in Figure 8.11, with the same invocation, VSK KOR remains `100%` until `800us` of binding life-time and begins to fall. For binding life-time greater than `1000us`, its KOR is always less than `10%` and its KOR is under `2%` for all binding life-time bigger than `10000us`. As in our working scenario, the authorization policy update frequency is estimated in the order of minutes, this rate can always be considered as less than `1%` comparing to `68.4%` of microkernel.

8.2.6 VSK Qualitative Evaluation

Security Analysis: For a software system based on VSK, three main threats exit:

1. an applicative component can violently gain access right through an existing binding. For the created binding, the current *Think* framework cannot distinguish the *read* and *write* actions, i.e., a binding for a *read* access action can be used to perform a *write* invocation. Thus, a malicious component can violently gain supplementary access permissions through existing binding, there does not exist any checking mechanism in *Think* or VSK to prevent from it;

2. an applicative component can illegally access to another applicative component by bypassing VSK protection. VSK serves as a kernel which checks and validates access

requests. Unfortunately, a malicious applicative component may access to another component by bypassing the kernel. In the *Think* framework, an invocation can directly access to a physical address without any control;

3. an applicative component can illegally access to VSK. Since VSK is also built by *Think*, the same threat of the previous threat may menace the kernel. A malicious applicative component in the *execution space* can illegally access to the kernel. That is, the kernel itself is not secure.

The first threat can be coped by extending the definition of interfaces of the *Fractal/-Think* framework. Rather that defining an interface, we should determine its associated access action type. During the compilation, checking should be achieved which verifies invocation methods with its access action type.

The second and third threats refer to the *bypass* attack. A *Memory Management Unit (MMU)* hardware mechanism is usually used to avoid circumventing the *reference monitor*. This mechanism prevents bypass of VSK authorization checks. One MMU solution for component-based OS was implemented in *CRACKER* [95]. The MMU patent [71] applied in *CRACKER* organizes components of different security levels into different memory pages, and provides supplementary checking enforcement for inter-page invocation. For some hardware platforms like *AVR* or *ARM* which do not support MMU, Rippert [119] proposed a tool for code checking which replaces memory access by a pointer to a manager for security policy validation. We believe that isolation between applicative components and between the *execution space* and VSK can be achieved through these two categories of solutions.

Consequently, for a system build on VSK, traditional integrity and confidentiality attacks are taken into account: all access should be granted by VSK. But availability is not dealt, e.g., a kind of *Denial of Service* attack which continuously sends fault access requests will induce VSK to reconfigure itself all the time and block it from commodity requests. The availability has not been handled by VSK.

VSK as a Security Kernel: We argue that the VSK architecture has all the distinguishing features of a security kernel – or minimal implementation in an OS of the security-relevant features that mediates all accesses, is protected from modification, and is verifiable as correct. Indeed, our VSK intercepts all access requests (*completeness*), and cannot be modified from the *execution space* (*isolation*). Moreover, its simple architecture should facilitate proof of correctness (*verifiability*). VSK also provides additional features like: support of multiple authorization policies (*flexibility*), dynamic choice of the most adequate security configuration (*manageability*), and easy introduction of new authorization policies in the kernel (*extensibility*). Hence, VSK enables strong and yet flexible protection for applications running in the *execution space* during their whole life-cycle – from design, deployment, execution, maintenance, to un-installation.

8.3 Summary

The implementation (see Section 8.1) showed the realization of such a OS. Our prototype applies the component-based approach to organize all its software modules, it achieves dynamic reconfiguration to enable run-time controls over executing applications and its kernel. The evaluations (see Section 8.2) illustrated simplification of the authorization sub-system and performance improvement comparing to other OS architectures. Finally, a qualitative evaluation analyzed hardware support for such an OS.

Chapter 9

Validation of the End-to-end Framework

We have implemented and evaluated a terminal-side system built on VSK in the previous chapter. In this chapter, we describe two implementations of ASPF. We also present an end-to-end evaluation of the self-protection framework which involves VSK and ASPF implementations. Performance and security evaluations show that our approach for self-protection of pervasive systems achieves strong security with reasonable overhead.

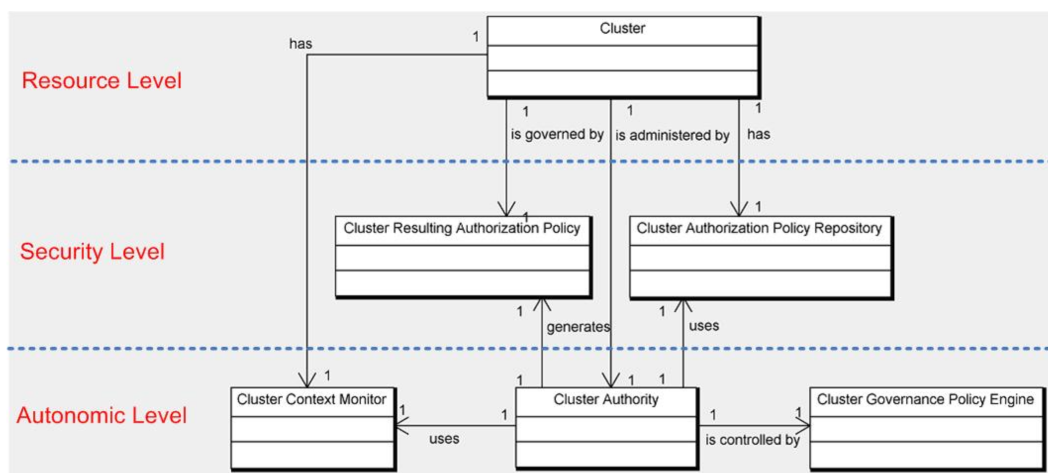

Figure 9.1: The Cluster Implementation Model.

In Chapter 5, two implementation models (the *cluster implementation model* and the *node implementation model*) are defined which indicate building blocks. As shown in Figure 9.1, the *cluster implementation model* consists of a *Cluster Authority* component for self-protection coordination, a *Cluster Context Monitor* for global context supervision; a *Cluster Authorization Policy Repository* for potential authorization policy storage; a *Cluster Governance Policy Engine* to generate security adaptation strategies; and a *Cluster Resulting Authorization Policy* which is output of the cluster-level security adaptation

process.

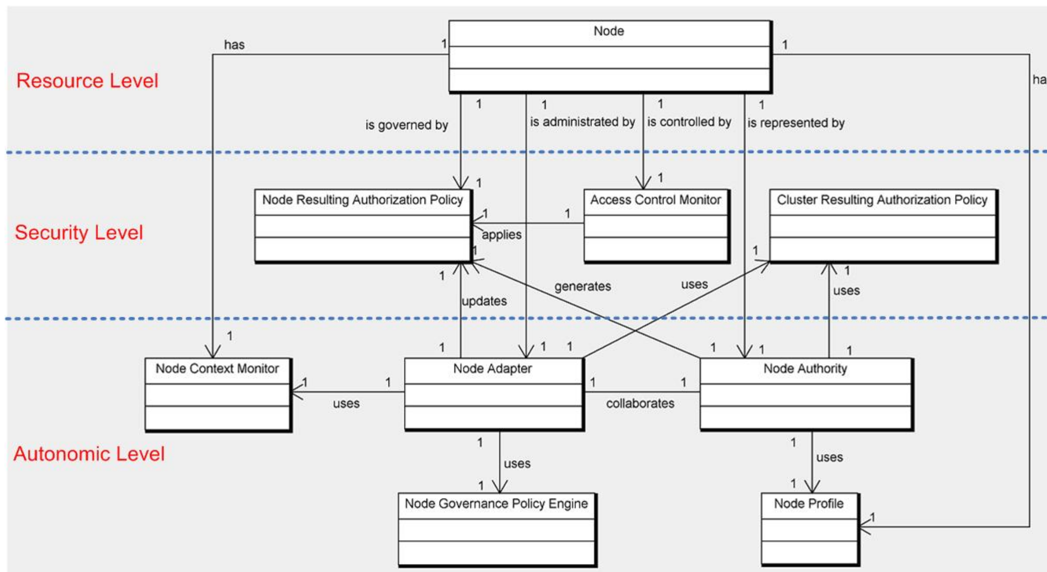

Figure 9.2: The Node Implementation Model.

The main elements of the *node implementation model* (see Figure 9.2) are a *Node Authority* and a *Node Adapter* for self-configuration and self-protection coordination; a *Node Profile* for policy customization; a *Node Context Monitor* for local context supervision; a *Node Governance Policy Engine* for local security adaptation strategies; and a *Node Resulting Authorization Policy* which is final output of the node-level security adaptation process.

9.1 A Basic ASPF Implementation

A first ASPF implementation consists of two parts: cluster-side ASPF components which play the role of a centralized monitor managing distributed terminals and node-side ASPF components which serve as a local controller to be installed in each node for synchronization with the cluster-side and for local resource administration. This implementation was realized as a standard JAVA application for the cluster-side. The node-side controller was developed using the *Think* framework [20] (see Figure 9.3).

9.1.1 Cluster-side ASPF

To simplify evaluation, we take into account a simple pervasive system which contains one cluster with several nodes. The cluster-side ASPF has only two software components: a *cluster implementation* which manages and updates settings of the cluster, and a *node implementation* which is a generic controller of all nodes in the cluster. In a general case

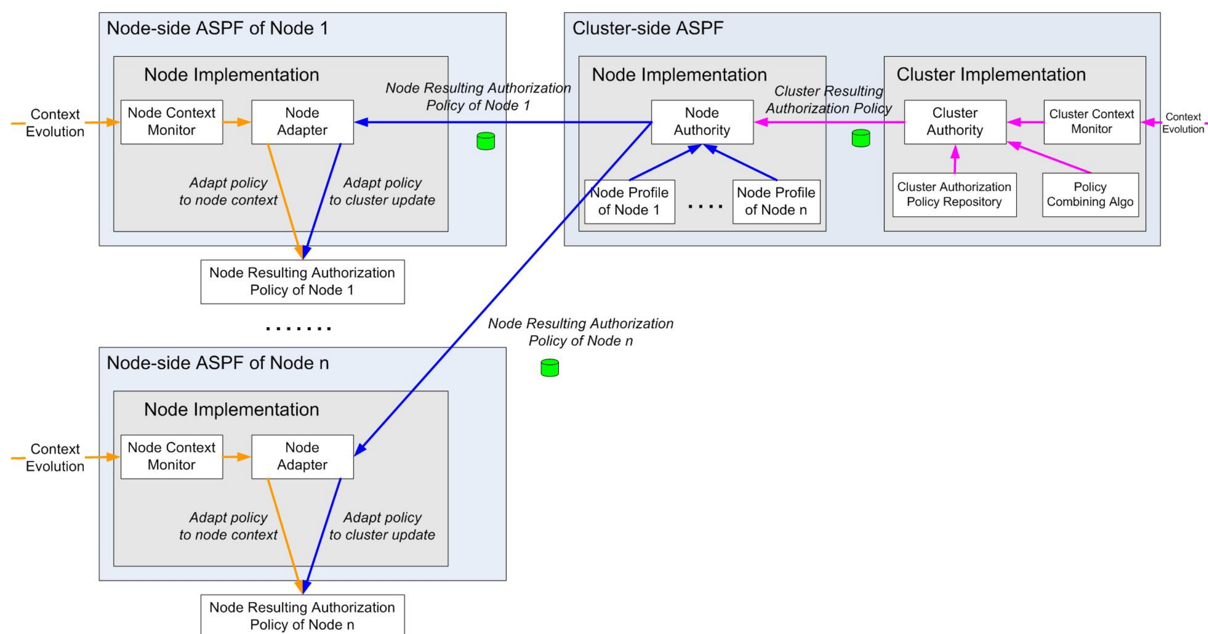

Figure 9.3: A Basic ASPF Implementation.

with several clusters, a set of *cluster implementation* and *node implementation* components will be defined in the cluster-side.

Cluster Implementation: As presented in Chapter 5, an implementation of ASPF cluster should contain four main components: a *cluster authority*, a *cluster context monitor*, a *cluster authorization policy repository*, and a *cluster governance policy engine*.

These four components yield four implementation components respectively: a *cluster authority*; a *cluster context monitor*; a set of *cluster-level authorization policies*; and a *policy-combining algorithm*.

When the *cluster context monitor* recognizes some security-related events in the environment, it abstracts them into a high-level representation such as an attack report. The reported attack triggers a cluster-level self-protection loop. Among a set of authorization policies (in the implementation, a DTE policy, a MLS policy and a RBAC policy are provided), the cluster authority selects one relevant policy using the *policy-combining algorithm*.

In our implementation, we give a simple example where the cluster context is modeled as three security levels (low, middle and high). Since we have not implemented a real intrusion detection system, we simulate attacks by manually changing the security levels which launches self-protection control loops. If the security level is tuned from the middle level to the low level, the DTE policy is applied. In the case of transition from the high security level to the middle level, the MLS policy is enforced. If the system is recovered

to the high level from an attack, it calls for the RBAC policy. The selected policy is then sent to the *node implementation* for further customization for all nodes of the cluster.

Node Implementation: The *node implementation* involves a *node authority* and a set of *node profiles* corresponding to each node of the cluster. The *node authority* realizes customization of authorization policies for each node. The *node profile* expresses node characteristics. To simplify the implementation, functionality of the *node governance policy engine* is achieved by filtering the *cluster resulting authorization policy* based on the *node profile*: when a *node authority* receives an updated *cluster resulting authorization policy*, it customizes it by removing irrelevant rules based on its *profile*. This customization generates a *node resulting authorization policy* which is specific for this node.

9.1.2 Node-side ASPF

The *node-side ASPF* implementation in the *Think* framework [20] contains two components: a *node adapter* and a *node context monitor*. The *node adapter* is a terminal-side entity that manages self-protection of nodes. The *node context monitor* collects local context information. When a *node resulting authorization policy* is customized by a *node authority*, it is then forwarded to a local *node adapter* on the terminal side. The *node adaptor* installs the policy into the VSK OS to realize a self-configuration loop. It also collaborates with the *node context monitor* to realized a node-level self-protection control loop.

9.1.3 Summary

This implementation realizes a middleware to administer authorization policies. It customizes the *cluster resulting authorization policy* into *node resulting authorization policies*, broadcasts them to each terminal and installs these policies.

Unfortunately, dynamical reconfiguration can be hardly achieved within the JAVA platform. This implementation is not fully adequate for dynamic pervasive systems since only a fixed number of nodes are permitted for each cluster; and additional *cluster implementation* and *node implementation* modules cannot be dynamically added. Therefore, a second ASPF implementation based on the *OSGi/iPOJO* platform is realized to include dynamic reconfiguration features.

9.2 Second ASPF Implementation

9.2.1 Platform Overview

OSGi: OSGi is a specification produced by the *Open Services Gateway initiative (OSGi) alliance* [5]. Its main objective is to enable modular assembly of software built with JAVA technology. It takes into account whole life-cycle of an application, from development, packaging, to uninstallation. Its ease of deployment is a principal reason why we choose it as the framework to develop server-side ASPF components. Applications can be packaged

into a deployment unit called *bundle* to be dynamically loaded or unloaded. Several implementations of the OSGi specification exist such as *Spring Dynamic Modules* [8], *Equinox* [3], and *Apache Felix* [1].

iPOJO - Service-oriented Component Model: *Apache Felix* was selected to implement server-side ASPF components. We apply iPOJO [37] which is a service-oriented component model based on *Felix*. *Service-oriented architecture* [107] allows to coordinate heterogeneous resources or applications, and it mitigates dependency between different modules. *Component-based design* [40] improves manageability (at both design and deployment phases) by encapsulating resources and applications into components, and enabling dynamic reconfiguration of such components. A *service-oriented component model* [44] combines these two approaches: (1) components implement service specifications; (2) interactions between components are achieved through service publication and usage, the underlying platform managing dependencies.

iPOJO is a service-oriented component model implemented on *Felix*. In iPOJO [37], component instances implement interfaces which describe provided services. A set of component instances can be packaged into a *bundle* as a deployment unit. Underlying iPOJO platform manages service publication and usage. Once a service becomes available, it automatically creates dependencies between this component instance and all others that call for the service. Therefore, users only address functionality development and leave iPOJO to manage non-functional aspects.

9.2.2 ASPF Implementation

As shown in Figure 9.4, unlike the first ASPF implementation, a *node implementation* is associated to one node instead of one *node implementation* component for all nodes of one cluster. With this design choice, when a new node joins a cluster, a corresponding *node implementation* component instance is dynamically created in the *cluster-side ASPF*. The *node authority* sub-component customizes a *cluster resulting authorization policy* into a *node resulting authorization policy* based on its own *node profile*.

With the iPOJO platform, we apply the *service-oriented* approach where each component provides a service to be used by another components (see Figure 9.5). The *policy set controller* provides a set of potential authorization policies to be selected. The *cluster context monitor* provides real-time context information. The *adaptation policy engine* uses these information and chooses a most adequate policy fitting the context. This policy is then broadcasted by the *cluster authority* to all the *node implementation* component instances whose corresponding node is in the cluster. Coordination of these instances are performed by a *node coordinator*. All the described modules are implemented in one OSGi bundle called *core bundle*. Once the *core bundle* is installed in execution platform, all these services become available.

To simulate the execution of this framework, we add two supplementary bundles: a *node life-cycle manager bundle* and a *cluster-level self-protection manager*. The *node life-cycle manager bundle* monitors availability of nodes in one cluster. To simplify the implementation, we use an interface to manually add nodes which simulate node joining in

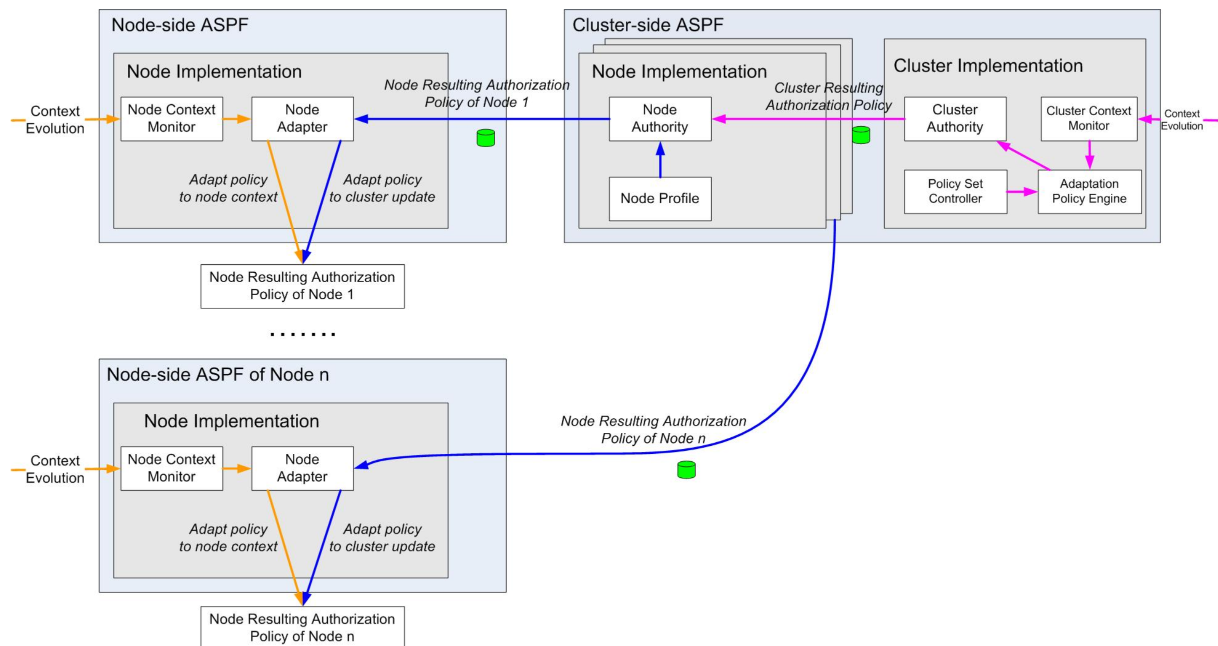

Figure 9.4: an iPOJO-based ASPF Implementation

the pervasive system. With this simulation, ASPF is able to create new *node implementation* component instances. The *cluster-level self-protection manager* serves to launcher cluster-level self-protection control loop.

9.3 Evaluation of the End-to-End Framework

The self-protection capabilities of the framework (including both ASPF and VSK) were evaluated in terms of overall response time and resiliency to attacks. All measurements were performed on a 2.7GHz *DELL OptiPlex* 740 desktop PC with *Linux/Ubuntu* 9.04 and 1GB of RAM, on which are run the first version of the ASPF implementation and the VSK emulator described in the previous chapter.

Three authorization policies, DTE, MLS and RBAC, are currently supported, with 10 subjects (threads) and 60 objects (system calls) to model a typical real-time OS environment, and 3 security levels for the cluster security context. The *node authority* filters the *cluster resulting authorization policy* according to active subjects or objects described in its node profile. Corresponding attribute mappings and rules are loaded inside VSK via a dedicated reconfiguration interface `ReconfVSK` being able to dynamically change security attributes and rules in the ACM component of VSK.

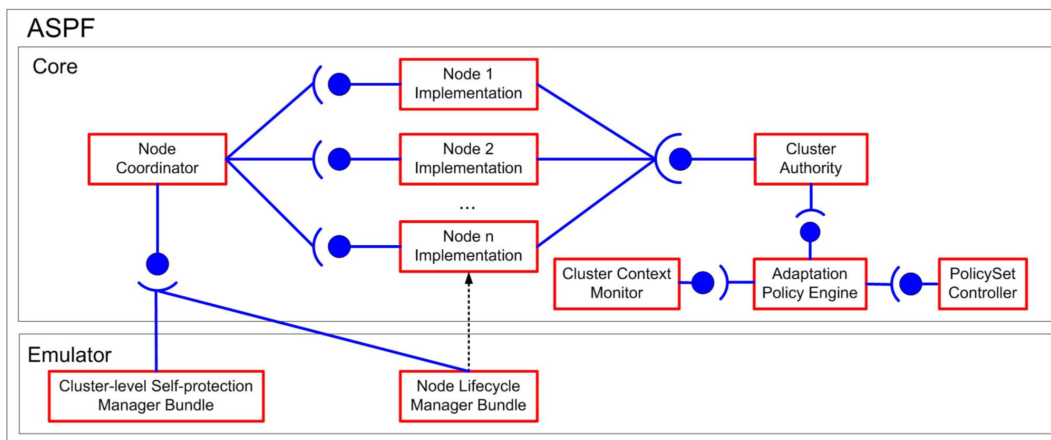

Figure 9.5: ASPF Service-oriented Implementation.

9.3.1 End-to-End Response Time

We measure overall latency to complete a full self-protection loop at the cluster and node levels. Evaluation results for each step of the loop are shown in Figures 9.6 and 9.7 for different types of authorization policies.

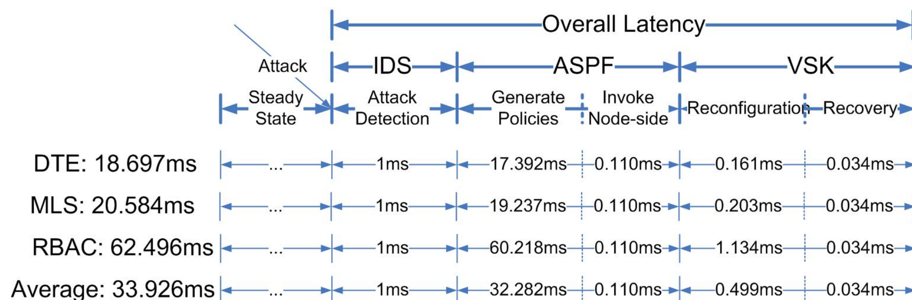

Figure 9.6: Cluster-Level Self-protection Latencies

In the first benchmark, detection of an attack on a cluster of 100 nodes in a steady state is simulated by a direct update of the cluster security context. In practice, this step would be performed by an Intrusion Detection System (IDS) such as Snort [7], with 1ms as typical order of magnitude for attack detection and countermeasure initiation. The next steps are generation of a node-specific policy (given times are averaged on the number of nodes), invoking the node VSK to load a policy, kernel reconfiguration with the new policy, and return to the steady state. Overall latency averaged over different authorization policies is `33.92ms`.

In the second benchmark, attacks are detected by a *node context monitor*. The next steps include invoking the VSK, tuning security attributes to adapt to the new security

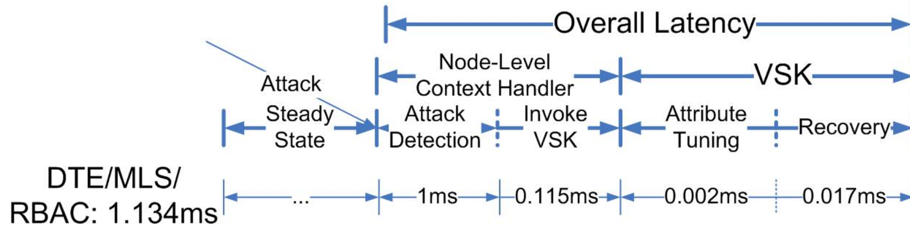

Figure 9.7: Node-Level Self-protection Latencies

context, and returning to a steady state. Measured overall latency for this adaptation loop is `1.134ms`.

Overall, the adaptation response times seem reasonable, since time between two policy reconfigurations is typically from a few seconds to one minute, for instance when switching between wireless networks in different locations. As expected, node-level adaptations are much lighter than cluster-level reconfigurations. This is in part due to the attribute-based approach: same authorization rules may be applied, only attributes values being tuned. For highly dynamic environments, this design makes self-protection more flexible, allowing to follow small variations of context, without regenerating a full authorization policy.

9.3.2 Resilience

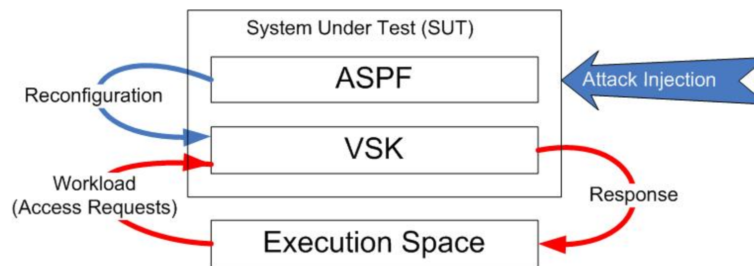

Figure 9.8: Benchmarking Self-Protection Capabilities: Principles

To measure effectiveness of self-protection, we use a methodology for benchmarking self-* capabilities of autonomic systems proposed in [39] based on injection of disturbances (see Figure 9.8). The idea, coming from dependability benchmarks, is to introduce in a *System Under Test (SUT)* disturbances in the form of attacks or faults, and to measure impact on performance workload. This type of benchmark, already used to assess self-healing abilities, measures how well the *SUT* adapts to the injected changes in terms of speed of recovery, impact on performance, etc.

In our case, the *SUT* is the end-to-end self-protection framework including VSK and ASPF on which is applied a workload to validate access requests from the *execution space*. We measure impact on throughput (number of requests per second validated by VSK,

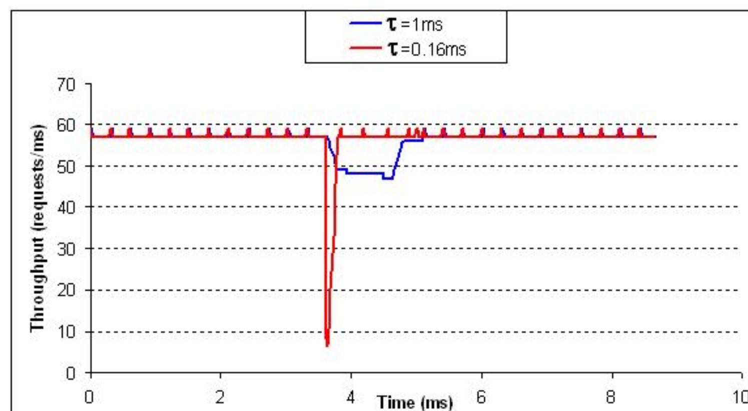

Figure 9.9: Benchmarking Self-Protection Capabilities: Results

averaged over a sliding sampling time window τ) of updating authorization policies to respond to injected attacks. An attack from a malicious node is simulated by directly changing the cluster security context at the beginning of an injection slot, and waiting from the SUT to come back to a steady state. The results are shown in Figure 9.9 for $\tau = 1ms$ and $\tau = 0.16ms$, which are about latency value for an end-to-end reconfiguration. The decrease in throughput due to security adaptations depends on the sampling slot value: 89% for $\tau = 0.16ms$ (worst case), but only 15% for $\tau = 1ms$ (standard situation). These results show that the system is able to protect itself effectively with a reasonable performance cost. The recovery time is almost immediate for $\tau = 0.16ms$, and about `2ms` for $\tau = 1ms$. Thus, the system is able to complete successfully its reconfiguration in time which are largely acceptable. These metrics tend to show that ASPF provides self-protection with minimal impact on system resources.

9.3.3 Security Evaluation

A assessment of security of the framework is also given. Evaluating the quality of the autonomic response is harder: does a system remain secure after a security reconfiguration? To avoid rogue third parties to directly update node authorization policies inside VSK, a single reconfiguration interface (`ReconfVSK`) is introduced as unique entry point to control VSK. This interface remains internal to a node, to avoid policy update requests coming from network aiming to lower node security settings.

ASPF behaves as a distributed authorization management plane which guarantees complete mediation over this interface: all authorization policy modifications may only be issued by *Node Authority*, *Node Coordinator*, and *Cluster Authority* components along a trusted path. Links between node-side and cluster-side ASPF components are also assumed to be secure, authenticated channels to avoid *man-in-the-middle* attacks or rogue cluster authorities.

Finally, a MMU hardware mechanism in the node (see Chapter 8) prevents circumvent-

ing the *Node Adapter* component. These features qualify ASPF as a strongly protected management plane over VSK authorization mechanisms.

9.4 Summary

Two implementations of the Autonomic Security Policy Framework (ASPF) are firstly elaborated in this chapter. Section 9.1 presents an implementation based on JAVA, and Section 9.2 describes an ASPF implementation realized by the *OSGi/iPOJO* platform which enables a more flexible and dynamic control. Section 9.3 illustrates efficiency and resilience of the first implementation.

Chapter 10

Cloud Computing Validation

10.1 Cloud computing Environments

In cloud computing, computing services is migrating away from local machines. Computation and storage are concentrated in remote data centers. Hardware and software of infrastructure are self-organized to achieve cloud services which offer both end-users advantages in terms of mobility and collaboration, and improve scalability and availability of asked services. Principle behind is computing capability allocation for IT systems.

Since such a system consists of a collection of interconnected *Virtual Machines (VMs)* which are distributed and dynamically provisioned, protection of cloud infrastructure for confidentiality, privacy or availability becomes an open issue. As various cloud services may be built on the same infrastructure and each one is adopted with its own protection mechanisms, a customized and reconfigurable security framework is demanded for diversity of security requirements. During construction of a service, an end-user should first declare his security requirements such as data confidentiality, user privacy and system availability. Furthermore, each service has its own security policy depending on their settings, some data sensitive cloud services need additional protection supports. Based on all these requirements, a security model is needed to realize self-protection through all the levels from hardware, OS, middleware to application. Once a service is constructed, it needs to be monitored at run-time and can be self-adapted in following its context evolution.

In this chapter, we validate the framework design by showing through a short case study that ASPF is generic enough to be applicable to the cloud computing infrastructure, other types of large-scale systems than simply pervasive networks. In the sequel, we focus on cloud computing infrastructures. We first recall some main security issues of those environments (Section 10.2), highlighting need for self-protection mechanisms. We then present the targeted self-protection scenarios (Section 10.3). We finally show how the ASPF core model (Section 10.4), extended model (Section 10.5), and authorization architecture (Section 10.6) may be refined to realize and coordinate several self-protection loops in the cloud setting.

10.2 Towards Self-Protecting Clouds

Cloud computing raises many security challenges [16], notably due to vulnerabilities introduced by virtualization of computing resources, and unclear effectiveness of traditional security architectures in fully virtualized networks. One of the main issues is how to guarantee strong resource isolation, both on the computing and networking sides in a multi-tenant environment.

Few solutions are available, usually addressing only one of the two aspects [28, 122]. The extremely short response times required to activate system defenses efficiently, and the impossibility of manual security maintenance call for a flexible, dynamic, and automated security management of cloud infrastructures, which is clearly lacking today. A framework enabling self-protection of a cloud infrastructure could provide answers to some of those challenges, making ASPF an interesting candidate to reach this objective.

In the cloud, virtualization has two facets:

- *Computing resources* are abstracted away from the hardware in the form of VMs isolated by a hypervisor on each server of a data center. Threats come at two levels of granularity: at the host level, through weaknesses either in the VM (guest OS) or the hypervisor; and at the cloud-level, mainly in the form of network-level attacks found in traditional security environments (e.g., DoS). An autonomous security management framework for the cloud should thus put in place self-protection loops at each of those two levels.

- *Network resources* (routers, firewalls,...) themselves become virtualized, e.g., as virtual appliances. Network zones where traffic could be separated physically or logically using VLANs or VPNs are replaced by *logical security domains* which may have flexible boundaries. It is thus critical to be able to manage security autonomously in such “islands”. The security management framework should thus also provide self-protection abilities in logical security domains, called *VSBs (Virtual Security Domains)* in the sequel.

10.3 Cloud Self-Protection Scenario

We explore the realization of *adaptable quarantine zones*: a number of VMs considered as compromised are isolated from the data center temporarily. Confinement may be lifted when the risk has decreased, and the VMs not considered hostile any more.

We assume that on each physical machine of the data center is installed a firewall component which allows to control strictly communications between VMs: an authorization policy specifies which interactions are allowed/forbidden. This virtual firewall may for instance be located in the domain 0 of a Xen hypervisor. Additional firewalls may also be placed at the cloud level to control inter-machine communications. The authorization policy is dynamically reconfigurable according to the estimated level of risk. Self-protection of the virtualized infrastructure then consists in adapting this set of policies according to the execution context of the data center, more or less hostile. Depending on alerts generated

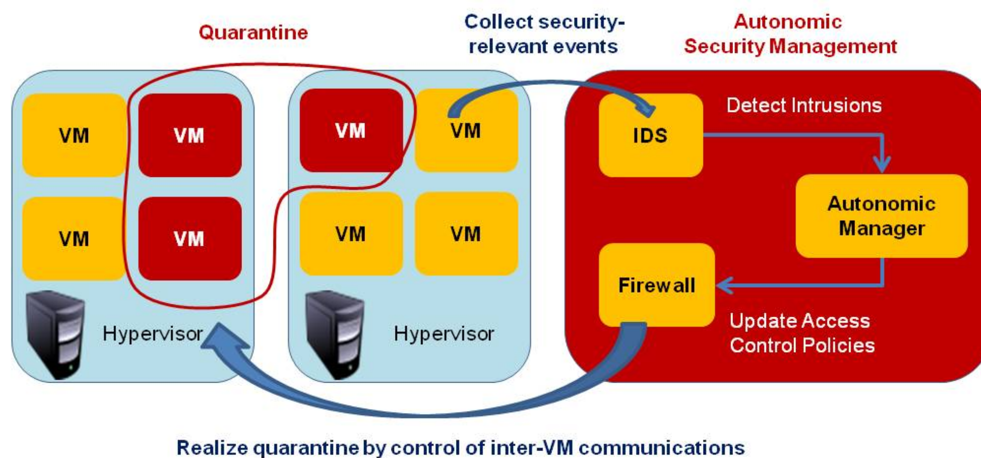

Figure 10.1: An Adaptable Quarantine Zone.

from an IDS (local or distributed in the data center), the most adequate authorization policy is autonomously selected, and installed in the different firewalls to realize hardened control over VM communications, and enforce the quarantine zone (see Figure 10.1).

In what follows, the quarantine zone is implemented at three levels of granularity: (1) within in physical server (*machine-level self-protection*); (2) within a VSB (*logical self-protection*); and (3) at the cloud level (*system-level self-protection*). The next sections describe how the ASPF core and extended models may be refined to realize those 3 self-protection loops.

10.4 ASPF Core Model

10.4.1 Resource Model

This model describes the organization of a cloud infrastructure (see Figure 10.2). As for the pervasive case, entities derive from a generic *Resource* class.

- The *System* class represents the overall cloud infrastructure to be protected, physically composed of a set of machines and logically divided into several *VSBs*. Both physical and logical isolation are realized through *Authorization Policies*.

- A *Machine* is a server in the data center. It hosts several VMs, isolated by an hypervisor, which may create, destroy, or migrate VMs on demand.

- VM is the first-class architectural component of the cloud. It runs a guest OS on top of the hypervisor, which manages VM resources.

- VSB is a logical unit of VM isolation, e.g. to compartimentalize different services. VMs belonging to a VSB may be distributed on several machines. VSBs may be strictly isolated between each other using network-level mechanisms.

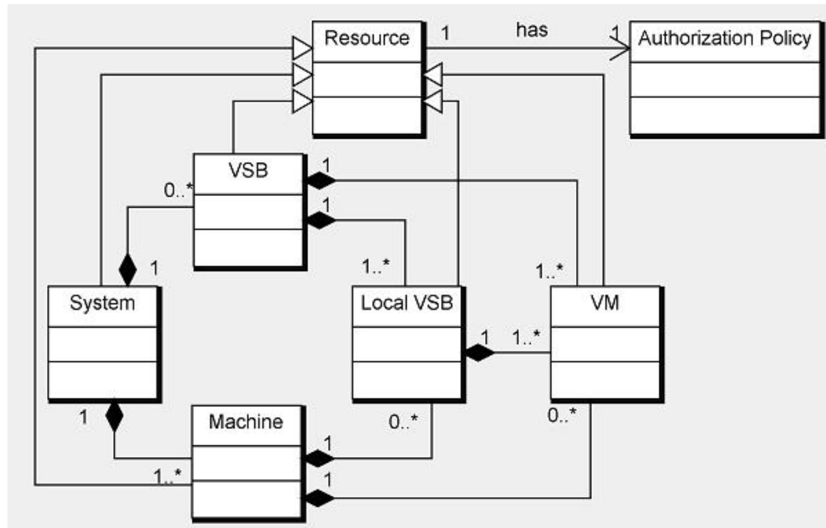

Figure 10.2: Cloud Resource Model.

- *Local VSB* contains all VMs of a VSB which reside on a given machine. It realizes local isolation from VMs of other VSBs in the machine. VM isolation at the VSB level is achieved by collaboration between all the corresponding Local VSBs.

10.4.2 Security Model

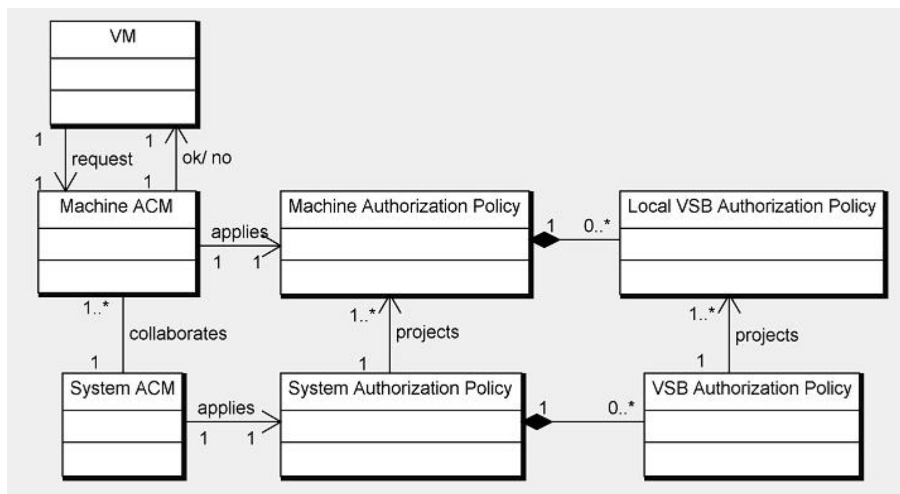

Figure 10.3: Cloud Security Model.

As for the pervasive case, access to resources is controlled by authorization policies. However, in the cloud, the security model features several types of policies since the resource model is richer (see Figure 10.3).

- The *system authorization policy* contains all access permissions to cloud resources. It will be enforced by the *system ACM* component at the cloud level.

- The *VSB authorization policy* contains access permissions in the scope of a VSB: it controls VM access at a logical level (the VSB security domain), regardless of the VM physical location. If we assume that access between two VMs belonging to different VSBs is always denied (strict isolation between VSBs), the *system authorization policy* may be viewed as the collection of *VSB authorization policies*. Policies in each VSB may be specified in different authorization models (e.g., DTE, MLS, or RBAC), as each VSB is a security island where policies may be administrated in a specific manner.

- The *local VSB authorization policy* is the projection of the *VSB authorization policy* inside a machine, and thus corresponds to two types of situations: VMs are co-located on the same machine; or VMs reside in different machines. In the former situation, access may be directly validated by at the machine-level. The latter calls for inter-machine collaboration.

- The *machine authorization policy* is the collection of *local VSB authorization policies* for all Local VSBs in the machine. Due to possible heterogeneity of authorization models between VSBs, in the general case, the *machine authorization policy* will be a set of *local VSB authorization policies* specified in different models. This policy will be enforced by the *machine ACM* component residing on each machine.

In our cloud model, to control inter-VM communications, policy enforcement is performed both at the machine level and the system level. We describe next a simple solution, other alternatives being possible.

If the VMs reside on the same machine, the *machine ACM* applies the *machine authorization policy* to validate the request. Since by default the VMs reside in the same VSB, validation is straightforward by enforcing the corresponding *local VSB authorization policy*. However, since *local VSB authorization policies* may be described in different models, a policy-neutral solution is required for access control enforcement at the machine level. Using G-ABAC for policy specification allows to achieve that goal as in the pervasive case.

If the VMs reside in different machines, the *machine ACM* of the requesting VM checks in its *machine authorization policy* whether this VM has permission to access an external machine. Control is then transferred to the *system ACM* which checks in the System Authorization Policy whether inter-machine communication to the target VM is allowed. Finally, the *machine ACM* of the target VM checks that requests to this VM coming from a remote machine are allowed. Such a three-step validation of requests allows authorization to be more efficient and scalable (local policies do not deal with inter-machine communications) and to check consistency of distributed policies at the system level.

10.5 Extended Models

The extended models describe the realization of several self-protection loops at different levels of granularity in the cloud, to address threats targeted at a machine, a logical security domain (i.e., a VSB), or the cloud itself by updating the corresponding authorization policies.

10.5.1 Machine Extended Model

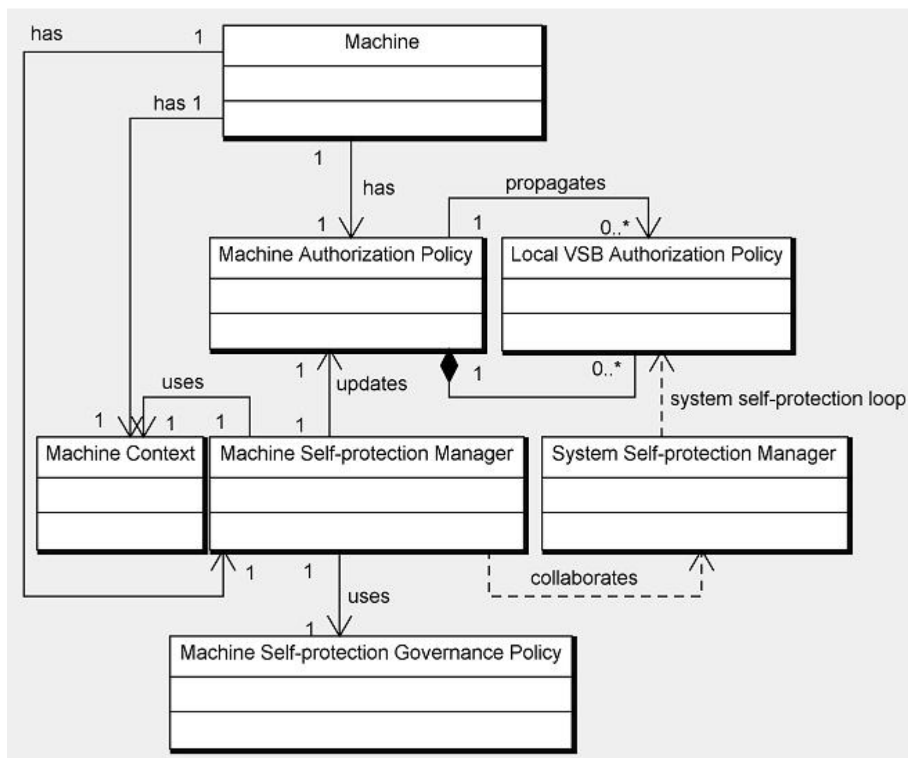

Figure 10.4: Machine Extended Model.

If a malicious VM compromises the hypervisor [123, 120], the threat may spread to all the VMs residing on the machine, which may need to be confined. Defeating such attacks is the objective of this self-protection loop (Figure 10.4).

When an attack is detected by the *machine context monitor*, the *machine self-protection manager* applies a *machine self-protection governance policy* to adapt the *machine authorization policy* to the current situation, policy which will be propagated to the authorization policies of each *local VSB* on the *machine*. At the same time, the manager collaborates with the *system self-protection manager* to determine whether further counter-measures should be triggered at the cloud level.

10.5.2 VSB Extended Model

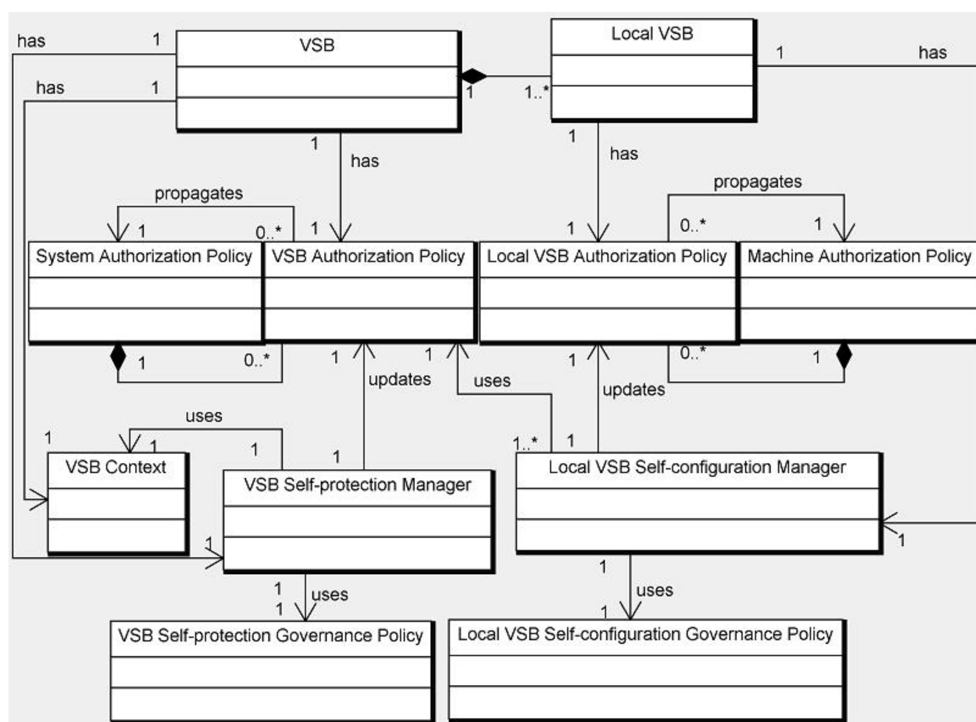

Figure 10.5: VSB Extended Model.

This self-protection loop (Figure 10.5) addresses a wider scope: it aims to defeat attacks which have spread into a logical security domain, e.g., by isolating compromised VMs. The *VSB authorization policy* is updated to fit the evolving *VSB security context* – those modifications are propagated to the *system authorization policy* to maintain policy consistency. A self-configuration loop is then launched to refine this policy into corresponding *local VSB authorization policies* – the modifications being propagated to the *machine authorization policies*.

10.5.3 System Extended Model

Two events may launch the system self-protection loop (Figure 10.6): detection of a cloud-level attack through *system context monitoring*; or a request from a *machine self-protection manager* for increased counter-measures, faced with an anomaly which cannot be handled at the machine level alone. Regarding self-protection, the *system self-protection manager* tunes the *system authorization policy* following the run-time adaptation strategy defined in the *system self-protection governance policy*. This update is propagated towards the relevant *VSB authorization policies*. As in a pervasive case, on each machine, a self-configuration mechanism then translates each *VSB authorization policy* into a *local VSB authorization policy*, finally updating the *machine authorization policy*.

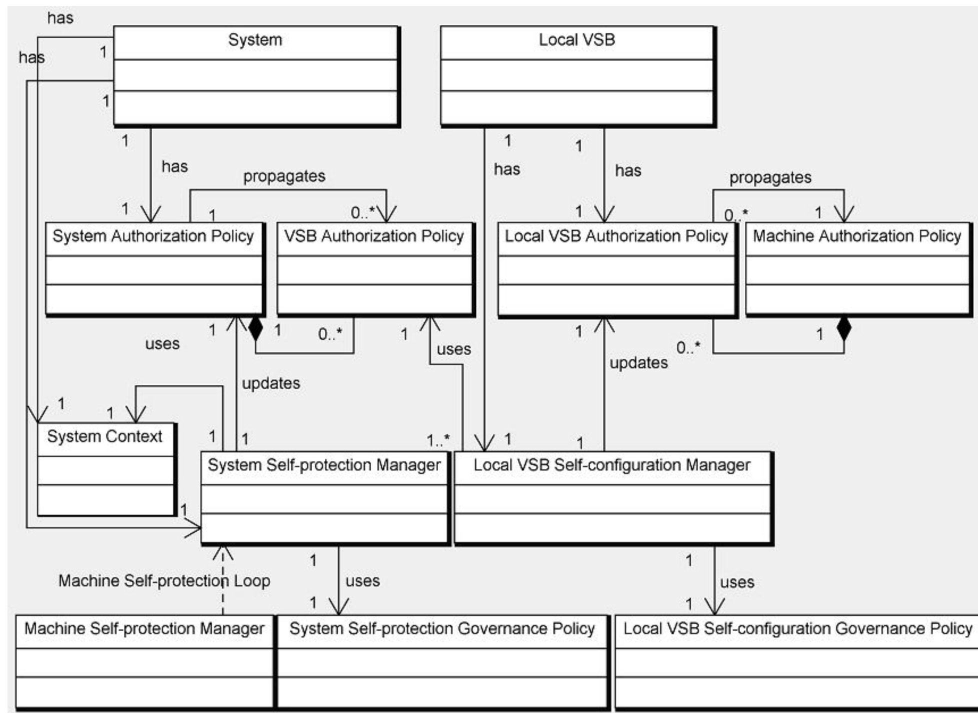

Figure 10.6: System Extended Model.

10.6 Authorization Architecture

An authorization architecture called *SECloud* was defined to implement the previous self-protection models. *SECloud* refines the ASPF authorization architecture. As shown in Figure 10.7, authorization validation is the result of a collaboration between *System* and *Machine ACMs*. *SECloud* consists of a number of server-side components to control *system*, *VSB*, and *local VSB* functionalities, while some machine-side components essentially apply authorization policy adaptation decisions taken at the other end-point, and control access among local VMs. Such an architecture is currently under implementation.

10.7 Summary

This chapter applies the same design approach in the setting of cloud computing. The *Cloud Resource* model featured a cloud infrastructure and the *Cloud Security* model described authorization validation of such an infrastructure. The extended models showed the organization of software components. Finally, the *SECloud* authorization architecture illustrated the collaboration of such components to realize self-protection.

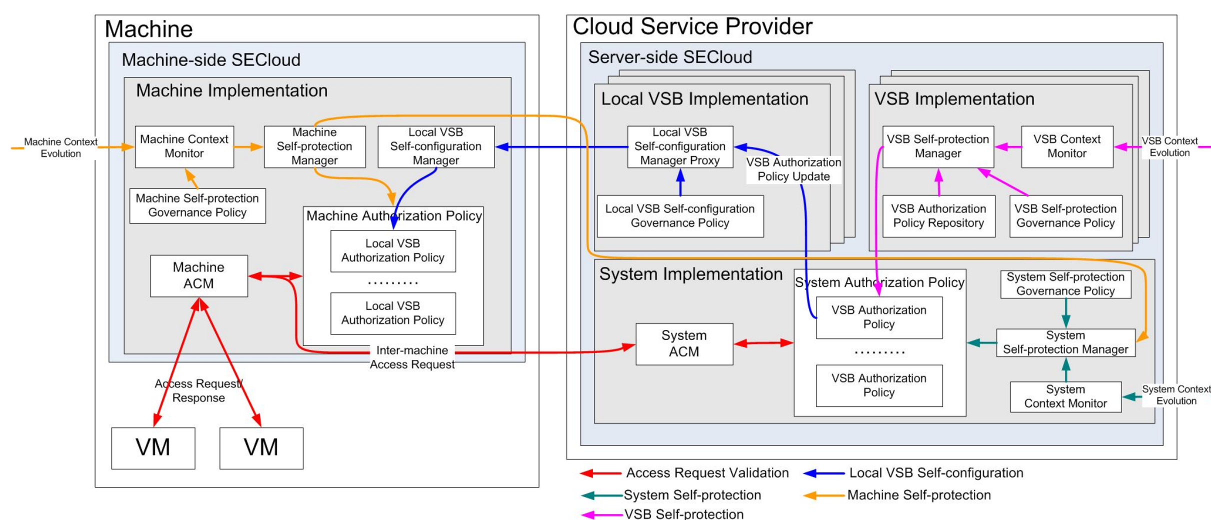

Figure 10.7: The SECloud Authorization Architecture.

Chapter 11

Conclusion

11.1 Summary of Contributions

This thesis applies autonomic computing to conventional authorization infrastructures. We illustrate that autonomic computing is not only useful for managing security infrastructure complexity, but also to mitigate continuous evolution threats. However, its application to pervasive systems is identified by a collection of design building blocks, ranging from an overall security architecture to the design of the OS embedded on the device. In this thesis, we propose:

- A three-layer abstract architecture for self-protection: a three-layer self-protection architecture is defined. A lower *execution space* provides a running environment for applications, a *control plane* supervises the *execution space*, and an *autonomic plane* guides the control behavior of the *control plane* in taking into account system status, risk evolution, administrator strategy and user preferences.

- An attribute-based access control approach: the proposed approach (called Generic Attribute-Based Access Control) applies attribute-based formulization for authorization which improves both policy-neutrality to specify a wide range of access control policies and flexibility to enable fine-grained manipulations on policies.

- A policy-based framework for authorization to realize autonomic security management: the policy-based approach has shown its advantages when handling complex and dynamic systems. An Autonomic Security Policy Framework (ASPF) provides a consistent and decentralized solution to administer authorization policies in large-scale distributed pervasive systems. The integration of autonomic features also enhances user-friendliness and context-awareness.

- A terminal-side security kernel for access control enforcement: the distributed authorization policies defined previously are enforced by an OS-level authorization architecture. This efficient OS kernel called VSK controls resource access in a dynamic manner to reduce authorization overhead. This dynamic mechanism also enables to support different authorization policies.

- A Domain-Specific Language (DSL) for adaptation policy specification: all adaptations of our end-to-end self-protection framework are controlled by high-level strategies called adaptation policies. A DSL to specify such policies is given which takes into account several aspects for adaptation decisions.

Implementations of the terminal-side OS and of the network-side server show the feasibility to realize the proposed design for self-protection and fulfill requirements such as flexibility of run-time control, efficiency of protection mechanism, and integration of autonomic functions. The results of evaluations (both in terms of local micro-benchmarks and end-to-end macro-benchmarks) show that such a framework provides strong and yet flexible security while still achieving good performance, making it applicable to build self-protected pervasive systems.

Such a framework achieves self-protection by adapting authorization policies according to the context evolution, system status, administrator strategies, and user preferences. However, the implementation of the DSL for policy adaptation is still missing. Currently, a DSL framework called yTune is under development, including an editor and a parser for specification and checking of different DSLs. Ongoing work is focused around implementing the described refinement mechanisms for the self-protection DSL in the yTune parser, and coupling the DSL toolchain with the ASPF self-protection framework. In the future, we also plan to enhance the DSL with more complete and realistic taxonomies for security attacks and countermeasures, for instance through dedicated security ontologies which may be coupled with the corresponding security components to detect intrusions and perform reactions.

From the security perspective, the memory isolation is assumed to be achieved by some MMU-like mechanisms. Existing MMU solutions guarantee strict isolation for a closed system where no new applications are dynamically added. For highly open systems such as pervasive systems, a flexible and dynamically reconfigurable MMU solution is still lacking.

11.2 Perspectives

The initial objective of this thesis was to realize an end-to-end self-protection framework for pervasive systems. The evaluation results show the feasibility and efficiency of our proposition. Future works address applications of our framework to other types of IT systems or extend the framework to other concerns than security.

11.2.1 Self-protection for Other Types of IT Systems

A pending question is whether this self-protection framework can be used to protect other types of IT systems than simply pervasive systems. Current IT systems have several tendencies like increasing complexity, continuous evolution and high administration overhead. Existing security solutions usually focus on one specific aspect without a global control over the whole system. Moreover, administrators need to manually configure the protection mechanisms which leads to a high overhead and may induce disfunctions and dramatic losses. We believe self-protection should be an inevitable design approach to manage security in future emerging IT systems.

To extend our framework to such IT systems, the three layer self-protection architecture should be applied. The protection system needs a clear separation between executing applications, control mechanisms, and autonomic functions to mitigate complexity. Furthermore, the policy-based approach improves the enforcement of adaptation decisions. All security updates can be achieved through policies leaving the infrastructure unchanged. Run-time control is another important design challenge since the adaptation decision affects control mechanisms. This modification of control mechanisms need to be propagated to running applications. Unfortunately, existing IT systems hardly provide dynamic reconfiguration solutions. This roadblock will need to be overcome in the future.

For example, the cloud computing validation is a first step towards applying the same framework to cloud infrastructures. Our first experiments show that the server-side policy-based framework can be reused. Authorization policies are customized according to each hypervisor and are delivered through the cloud infrastructure to control local access. However, the terminal-side framework has a dependency on the underlying platforms. The terminal-side prototype of the initial framework was implemented on the *Fractal/Think* framework in which dynamic reconfiguration can be easily achieved. Unfortunately, in a cloud infrastructure, each service is equipped with its own platform. For the tested *Xen* platform, dynamic reconfiguration is hardly addressed, and we cannot dynamically create or remove bindings to manage access permissions. One solution is to use static hooks by which each access request should be checked. This framework is currently under implementation. Another promising solution of dynamic reconfiguration is to re-implement existing IT systems on a component-based platform. Component-based software engineering has proven as a valuable approach to increase manageability and dynamic reconfiguration. The re-implementation of legacy IT systems enables encapsulation of applications into components. Interfaces and controllers treating specific functionalities may be inserted to each component. However, this solution calls for huge workload.

11.2.2 Framework Extensions to other Concerns

Another possible extension of this work is to apply the same approach for other aspects than security, e.g., QoS or energy efficiency management. We believe that a three-level architecture should still be applicable and that the policy-based approach may also be used to guide behaviors. Instead of authorization policies, other types of policies need to be applied.

Our current self-protection prototype uses authorization polices for the *control plane* and self-protection adaptation policies for the *autonomic plane*. In order to extend the framework to other aspects, we need to specify other classes of control policies. For instance, for QoS improvement, some kinds of QoS policies will be integrated in the *control plane*. The adaptation policies of the *autonomic plane* should thus take into account other aspects to produce adaptation guidelines. This asks for a generic DSL combining multiple aspects. On the other hand, run-time control always remains as a roadblock. Some adaptations call for supplementary mechanisms, e.g., changing communication protocols, or replacing coding module at the hardware level. All these require coarse- or fine-grained reconfiguration mechanisms which is missing.

Bibliography

[1] Apache Felix. http://felix.apache.org/site/index.html.

[2] eCos. http://ecos.sourceware.org/.

[3] Equinox. http://www.eclipse.org/equinox/.

[4] Intel Active Management Technology white paper. *http://www.intel.com/go/iamt*.

[5] Open Services Gateway initiative. http://www.osgi.org/.

[6] Privacy and Identity Management for Europe (PRIME). http:://www.prime-project.eu/.

[7] Snort: a Network Intrusion Prevention and Detection System. http://www.snort.org/.

[8] Spring Dynamic Modules for OSGi Service Platforms. http://www.springsource.org/osgi.

[9] RFC1035: Domain Names - Implementation and Specification. 1987. http://tools.ietf.org/html/rfc1035.

[10] Workshop on Logical Foundations of an Adaptive Security Infrastructure (WOLFASI). In *In conjunction with Workshop on Foundations on Computer Security (FCS)*, 2004.

[11] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, and M. Parashar. AutoMate Enabling Autonomic Applications on the Grid. In *Autonomic Computing Workshop*, pages 48–57, 2003.

[12] J-M. Agosta, J. Chandrashekar, D-H. Dash, M. Dave, D. Durham, H. Khosravi, H. Li, S. Purcell, S. Rungta, R. Sahita, U. Savagaonkar, and E-M. Schooler. Towards Autonomic Enterprise Security: Self-defending Platforms, Distributed Detection, and Adaptive Feedback. *Intel Technology Journal*, 10(4), 2006.

[13] D. Agrawal, K-W. Lee, and J. Lobo. Policy-based Management of Networked Computing Systems. *Communication Magazine, IEEE*, 43(10):69–75, 2005.

[14] M. Alia, M. Lacoste, R. He, and F. Eliassen. Putting Together QoS and Security in Autonomic Pervasive Systems. In *International Symposium on QoS and Security in Autonomic Pervasive Systems (Q2SWinet)*, 2010.

[15] M. Aljnidi and J. Leneutre. ASRBAC: A Security Administration Model for Mobile Autonomic Networks (MAutoNets). In *Data Privacy Management and Autonomous Spontaneous Security*, volume 5939/2010, pages 163–177. 2010.

[16] Cloud Security Alliance. Top Threats to Cloud Computing. *http://www.cloudsecurityalliance.org/topthreats.html*.

[17] S-R-J. Ames, M. Gasser.M, and R-R. Schell. Security Kernel Design and Implementation: An Introduction. *Computer*, 16(7):14–22, 1983.

[18] E. Amoroso. Fundamentals of Computer Security Technology. *Prentice-Hall, Englewood Cliffs, New Jersey*, 1994.

[19] A. Anderson. XACML profile for role based access control (RBAC). *OASIS Access Control TC committee*, 2004.

[20] M. Anne, R. He, T. Jarboui, M. Lacoste, O. Lobry, G. Lorant, M. Louvel, J. Navas, V. Olive, J. Polakovic, M. Poulhies, J. Pulou, S. Seyvoz, J. Tous, and T. Watteyne. Think: View-Based Support of Non-Functional Properties in Embedded Systems. In *6the IEEE International Conference on Embedded Software and Systems*, 2009.

[21] C-A. Ardagna, S-D. Capitani di Vimercati, S. Paraboschi, E. Pedrini, and P. Samarati. An XACML-based Privacy-centered Access Control System. In *1st ACM workshop on Information security governance*, pages 49–58, 2009.

[22] M. Auslander, D. Dasilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R-W. Wisniewski, and J. Xenidis. K42 Overview. Technical report, 2002.

[23] S. Ayed and N. Cuppens. An Integrated Model for Access Control and Information Flow Requirements. In *Asian Computing Science Conference Focusing on Secure Software and Related Issues (ASIAN)*, 2007.

[24] L. Badger, D-F. Sterne, D-L. Sherman, K-M. Walker, and S-A. Haghighat. Practical Domain and Type Enforcement for Unix. In *IEEE Symposium on Security and Privacy*, page 66, 1995.

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *9th ACM symposium on Operating Systems Principles*, pages 164–177, 2003.

[26] A. Baumann, J. Kerr, D-D. Silva, O. Krieger, and R-W. Wisniewski. Module Hot Swapping for Dynamic Update and Reconfiguration in K42. In *6th Linux Conference Au*, 2005.

[27] B-E. Bell and L. Lapadula. Secure Computer System Unified Exposition and Multics Interpretation. Technical report, 1976.

[28] S. Berger, R. Caceres, D. Pendarakis, R. Sailer, E. Valdez, R. Perez, W. Schildhauer, and D. Srinivasan. TVDc Managing Security in the Trusted Virtual Datacenter. *ACM SIGOPS Operating Systems Review*, 42(1):40–47, 2008.

[29] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. Mcnamee, S. Savage, and E.G. Sirer. SPIN: An Extensible Microkernel for Application Specific Operating System Services. Technical report, 1994.

[30] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *5th ACM Symporium on Operating Systems Principles*, pages 267–283, 1995.

[31] E. Bertino, B. Catania, M-L. Damiani, and P. Perlasca. GEO-RBAC: A Spatially Aware RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 10, 2007.

[32] K-J. Biba. Integrity Considerations for Secure Computer System. Technical report, 1977.

[33] M. Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[34] C. Blanco, J. Lasheras, R. Valencia-Garcia, E. Fernandew-Medina, A. Toval, and M. Piattini. A Systematic Review and Comparison of Security Ontologies. In *3rd International Conference on Availability, Reliability and Security*, pages 813–820, 2008.

[35] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *1996 IEEE Symposium on Security and Privacy*, page 164, 1996.

[36] G. Boudol and M. Kolundzija. Access Control and Declassification. *COMPUTER NETWORK SECURITY Communications in Computer and Information Science*, 1:85–98, 2007.

[37] J. Bourcier. *Auto-Home: A Framework for Autonomic Pervasive Applications*. PhD thesis, University JOSEPH FOURIER - Ecole Doctorale Mathematiques, Sciences et Technologies de l'Information, Informatique, 2008.

[38] L. Broto, D. Hagimont, P. Stolf, N. Depalma, and S. Temate. Autonomic Management Policy Specification in Tune. In *Symposium on Applied Computing*, pages 1658–1663, 2008.

[39] A-B. Brown and C. Redlin. Measuring the Effectiveness of Self-Healing Autonomic Systems. In *2nd International Conference on Autonomic Computing*, pages 328–329, 2005.

[40] A-W. Brown and K-C-V. Wallnau. Engineering of Component-Based Systems. In *Engineering of Complex Computer System*, pages 414–422, Montreal, Canada, 1996.

[41] E. Bruneton, T. Coupaye, M. Leclercq, and V. Quema. The Fractal Component Model and its Support in Java. In *Software: Practice and Experience*, volume 36, pages 1257–1284. 2006. Special Issue: Experiences with Auto-adaptive and Reconfigurable Systems.

[42] A. C-Myers. JFlow: Practical Mostly-Static Information Flow Control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[43] J. Camenisch and T. Grob. Efficient Attributes for Anonymous Credentials. In *15th ACM conference on Computer and Communications Security*, pages 345–356, 2008.

[44] H. Cervantes and R-S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *26th International Conference on Software Engineering*, pages 614–623, 2004.

[45] O. Chebaro, L. Broto, J-P. Bahsoun, and D. Hagimont. Self-TUNe-ing of a J2EE Clustered Application. In *6th IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 23–31, 2009.

[46] S-W. Cheng, A-C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste. Rainbow Architecture-based Self-adaptation with Reusable Infrastructure. In *1st International Conference on Autonomic Computing*, volume 0, pages 276–277, 2004.

[47] D-M. Chess, C-C. Palmer, and S-R. White. Security in an Autonomic Computing Environment. *IBM Systems Journal*, 42(1):107–118, 2003.

[48] M. Clarke and G. Coulson. An Architecture for Dynamically Extensible Operating Systems. In *4th International Conference on Configurable Distributed Systems*, page 145, 1998.

[49] B. Claudel, N. DePalma, R. Lachaize, and D. Hagimont. Self-protection for Distributed Component-Based Applications. In *Stabilization, Safety, and Security of Distributed Systems*, volume 4280/2006 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[50] C. Coma, N. Cuppens, F. Cuppens, and A.R. Cavalli. Context Ontology for Secure Interoperability. In *3rd International Conference on Availability, Reliability and Security*, 2008.

[51] IBM Autonomic Computing. An Architectural Blueprint for Autonomic Computing. *White Paper*, 2004.

[52] R-P. Cook and I. Lee. DYMOS: A Dynamic Modification System. In *Symposium on High-level debugging*, pages 201–202, Pacific Grove, Calfornia, 1983.

[53] M-J. Covington, M-J. Moyer, and M. Ahamad. Generalized Role-Based Access Control for Securing Future Applications. In *National Information Systems Security Conference*, 2000.

[54] M.J. Covington, V. Long, S. Srinivasan, A.K. Dev, M. Ahamad, and G.D. Abowd. Securing Context-Aware Applications Using Environment Roles. In *6th ACM Symposium on Access Control Models and Technologies*, pages 10–20, 2001.

[55] J. Crampton and G. Loizou. Administrative Scope and Role Hierarchy Operations. In *7th ACM symposium on Access control models and technologies*, pages 145–154, 2002.

[56] J. Crampton and G. Loizou. Administrative Scope A Foundation for Role-based Administrative Models. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):201–231, 2003.

[57] F. Cuppens, S. Gombault, and T. Sans. Selecting Appropriate Counter-Measures in an Intrusion Detection Framework. In *17th IEEE Workshop on Computer Security Foundations*, page 78, 2004.

[58] F. Cuppens and A. Miege. AdOrBAC An Administration Model for Or-BAC. *Computer Systems Science and Engineering (CSSE'04)*, 19, 2004.

[59] N. Cuppens, F. Cuppens, J-E. Lopew de Vergara, E. Vazquez, J. Guerra, and H. Debar. An Ontology-based Approach to React to Network Attacks. *International Journal of Information and Computer Security*, 3(3/4):280–305, 2009.

[60] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Workshop on Policies for Distributed Systems and Networks*, 2001.

[61] S. Davy, K. Barrett, S. Balasubramaniam, S. Meer, and J. Strassner. Policy-based Architecture to Enable Autonomic Communications: A Position Paper. In *International Conference On Emerging Networking Experiments And Technologies*, 2006.

[62] H. Debar, D. Curry, and B. Feinstein. The Intrusion Detection Message Exchange Format. *RFC 4765*, 2006.

[63] H. Debar, Y. Thomas, F. Cuppens, and N. Cuppens. Enabling Automated Threat Response through the Use of a Dynamic Security Policy. *Journal in Computer Virology*, 3(3):195–210, 2007.

[64] H. Debar, Y. Thomas, N. Cuppens, and F. Cuppens. Using Contextual Security Policies for Threat Response. In *3rd International Conference on Detection of Intrusions and Malware Vulnerability Assessment*, 2006.

[65] A-V. Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[66] S. Dobson, S. Denazis, A. Femandez, D. Gaiti, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A Survey of Autonomic Communication. *ACM Transaction on Autonomous and Adaptive Systems*, 1(2):223–259, 2006.

[67] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *4th international Conference on Embedded Networked Sensor Systems*, pages 15–28, 2006.

[68] P. Efstathopoulos, M. Krohn, S. Vanebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Mrris. Labels and Event Processes in the Asbestos Operating System. In *20th ACM Symposium on Operating Systems Principles*, volume 17-30, 2005.

[69] D-R. Engler, M-F. Kaashoek, and J. O'Toole. Exokernel: an Operating System Architecture for Application-level. In *15th ACM Symposium Operating System Principles*, pages 251–268, 1995.

[70] G. Faden. Reconciling CMW Requirements with Those of X11Applications. In *14th Annual National Computer Security Conference*, 1991.

[71] J-P. Fassino, T. Jarboui, and M. Lacoste. An Access Control System and Method, A Component-Based Kernel Including It, and Its Use. *US Patent Application n 11/792,900*, 2008.

[72] J-P. Fassino, J-B. Stefani, J. Lawall, and G. Muller. Think: A Software Framework for Component-Based Operating System Kernels. In *USENIX Annual Technical Conference*, pages 73–86. USENIX Association, 2002.

[73] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[74] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit A Substrate for Kernel and Language Research. In *16th ACM symposium on Operating systems principles*, pages 38–51, 1997.

[75] A-G. Ganek and T-A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 41(1):5–18, 2003.

[76] J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, and S. Preda. MIRAGE: A Management Tool for the Analysis and Deployment of Network Security Policies. In *3rd International Workshop on Autonomous and Spontaneous Security*, 2010.

[77] S. Godik and T. Moses. OASIS eXtensible Access Control Markup Language (XACML) Version 2.0 OASIS Standard. *OASIS Standard*, 2005.

[78] R. Grimm and B.N. Bershad. Separating Access Control Policy, Enforcement, and Functionality in Extensible Systems. *ACM Transactions on Computer Systems*, 19(1):36–70, 2001.

[79] S-E. Hallyn and P. Kearns. Domain and Type Enforcement for Linux. In *4th annual Linux Showcase Conference*, volume 4, pages 15–15, 2000.

[80] Z. Hayat, J. Reeve, and C. Boutle. Ubiquitous Security for Ubiquitous Computing. *Information Security Tech. Report*, 12(3):172–178, 2007.

[81] R. He, M. Lacoste, and J. Leneutre. An OS Architecture for Device Self-protection. In *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Lyon, 2009.

[82] R. He, M. Lacoste, and J. Leneutre. A Policy Management Framework for Self-protection of Pervasive Systems. In *6th International Conference on Autonomic and Autonomous Systems*, pages 104–109, 2010.

[83] R. He, M. Lacoste, and J. Leneutre. Virtual Security Kernel: A Component-Based OS Architecture for Self-Protection. In *3rd IEEE International Symposium on Trust, Security and Privacy for Emerging Applications*, 2010.

[84] J. Helander and A. Forin. MMLite: A Highly Componentized System Architecture. In *8th ACM SIGOPS European workshop on Support for Composing Distributed Applications*, pages 96–103, 1998.

[85] J-J. Hellerstein. Self-Managing Systems: A Control Theory Foundation. In *29th Annual IEEE International Conference on Local Computer Networks*, 2004.

[86] M-C. Huebscher and J-A. McCann. A Survey of Autonomic Computing–Degrees, Models, and Applications. *ACM Computing Surveys (CSUR)*, 40(3), 2008.

[87] P. Inverardi and M. Tivoli. The Future of Software Adaptation and Dependability. *Software Engineering*, 5413/2009:1–31, 2009.

[88] S. Jajodia, P. Samarati, and V-S. Subrahmanian. A Logical Language for Expressing Authorizations. *IEEE Symposium on Security and Privacy*, 0:31–42, 1997.

[89] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2003.

[90] J-O. Kephart and D-M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[91] J-O. Kephart and W-E. Walsh. An Artificial Intelligence Perspective on Autonomic Computing Policies. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–12, 2004.

[92] A. Kim, J. Luo, and M. Kang. Security Ontology for Annotating Resources. In *International Conference on Ontologies, Databases, and Application of Semantics (ODBASE)*, 2005.

[93] O. Krieger, M. Auslander, B. Rosenburg, R-W. Wisniewski, J. Xenidis, D. Da, S-M. Ostrowski, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *EuroSys Conference*, pages 133–145, 2006.

[94] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: A Component Model for Secure Microkernel-Based Embedded Systems. *Journal of Systems and Software*, 80(5):687–699, 2006.

[95] M. Lacoste, T. Jarboui, and R. He. A Component-Based Policy-Neutral Architecture for Kernel-Level Access Control. *Annals of Telecommunications*, 64:121–146, 2009.

[96] M. Lacoste, G. Privat, and F. Ramparany. Evaluating Confidence in Context for Context-Aware Security. In *Ambient Intelligence*, volume 4794/2007 of *Lecture Notes in Computer Science*, pages 211–229. Springer Berlin / Heidelberg, 2007.

[97] B-W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[98] B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnam, and T. Freeman. A Flexible Attribute Based Access Control Method for Grid Computing. *Journal of Grid Computing*, 7:169–180, 2009.

[99] H-M. Levy. Capability- and Object-based System Concepts. In *Capability-based Computer System*, page 250. 1984.

[100] Y-T. Lim, P-C. Cheng, J-A. Clark, and P. Rohatgi. Policy Evolution with Grammatical Evolution . In *Simulated Evolution and Learning*, volume 5361, pages 71–80. 2008.

[101] S-B. Lipner. Non-discretionary Controls for Commercial Applications. In *IEEE Symposium on Security and Privacy*, pages 2–10, 1982.

[102] M. Litoiu, M. Woodside, and T. Zheng. Hierarchical Model-based Autonomic Control of Software Systems. In *Workshop on the Design and Evolution of Autonomic Application Software*, pages 1–7, 2005.

[103] O. Lobry, J. Navas, and J-P. Babau. Optimizing Component-Based Embedded Software. In *33rd Annual IEEE International Computer Software and Applications Conference*, volume 02, pages 491–496, 2009.

[104] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference*, pages 29–42, 2001.

[105] D-A. Menasce and J-O. Kephart. Guest Editors' Introduction: Autonomic Computing. *IEEE Internet Computing*, 11(1):18–21, 2007.

[106] M. Mernik, J. Heering, and A. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[107] N. Milanovic and M. Malek. Service-Oriented Operating System: A Key Element in Improving Service Availability. In *4th International Symposium on Service Availability*, pages 31–42, 2007.

[108] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy Core Information Model Version 1 Specification. *RFC3060*, 2001.

[109] H. Muller, M. Pezze, and M. Shaw. Visibility of Control in Adaptive Systems. In *2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems*, pages 23–26, 2008.

[110] H-A. Muller, H-M. Kienle, and U. Stege. Autonomic Computing Now You See It, Now you Don't. In *Software Engineering/ Design and Evolution of Autonomic Software Science*, volume 5413, pages 32–54. 2009.

[111] P-A. Muller, F. Fleurey, and J-M. Jezequel. Weaving Executability into Object-Oriented Meta-Languages. In *8th International Conference on Medel Driven Engineering Languages and Systems*, pages 264–278, 2005.

[112] NIST. A Survey of Access Control Models. In *NIST Privilege (Access) Management Workshop*, 2009. Available at: http://csrc.nist.gov/news_events/privilege-management-workshop.

[113] S. OH, R-S. Sandhu, and X-W. Zhang. An Effective Role Administration Model Using Organization Structure. *ACM Transactions on Information and System Security (TISSEC)*, 9(2):113–137, 2006.

[114] F. Hnsen V. Oleshchuk. SRBAC: A Spatial Role-Based Access Control Model for Mobile Systems. In *7th Nordic Workshop on Secure IT Systems*, pages 129–141, 2003.

[115] S. Osborn, R-S. Sandhu, and Q. Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(2):85–106, 2000.

[116] J. Park and R-S. Sandhu. Towards Usage Control Models Beyond Traditional Access Control. *7th ACM Symposium on Access Control Models and Technologies*, pages 57–64, 2002.

[117] J. Polakovic and J-B. Stefani. Architecting Reconfigurable Component-Based Operating Systems. *Journal of Systems Architecture: the EUROMICRRO Journal*, 54(6), 2008.

[118] C. Prehofer and C. Bettstetter. Self-organization in Communication Networks Principles and Design Paradigms. *IEEE Communication Magazine*, 43(7):78–85, 2005.

[119] C. Rippert. *Protection dans les architectures de systmes flexibles.* PhD thesis, 2004.

[120] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud Exploring Information Leakage in Third-Party Compute Clouds.pdf. In *16th ACM conference on Computer and Communications Security*, pages 199–212, 2009.

[121] M. Rozier, V. Abrossimov, F. Arm, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems*, 1:39–69, 1988.

[122] P. Ruth, J. Rhee, D-Y. Xu, R. Kennell, and S. Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *IEEE International Conference on Autonomic Computing*, pages 5–14, 2006.

[123] J. Rutkowska and R. Wojtczuk. The Qubes OS Architecture. *Invisible Things Lab Tech Rep*, 2010.

[124] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J-L. Griffin, and L-V. Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *21st Annual Computer Security Applications Conference*, pages 276–285, 2005.

[125] R-S. Sandhu. Lattice-based Access Control Models. *Computer, IEEE*, 26(1):9–19, 1993.

[126] R-S. Sandhu, V. Bhamidipati, and A-Q. Munawer. The ARBAC97 Model for Role-Based Administration of Roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[127] R-S. Sandhu, E-J. Goyne, H-L. Feinstein, and G-E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[128] A. Saxena, M. Lacoste, T. Jarboui, U. Lucking, and B. Steinke. A Software Framework for Autonomic Security in Pervasive Environments. In *Information System Security*, volume 4812, pages 91–109. Springer Berlin/ Heidelberg, 2007.

[129] M. Serrano, S. Meer, J. Strassner, S. Paoli, A. Kerr, and C. Storni. Trust and Reputation Policy-based Mechanisms for Self-protection in Autonomic Communications. In *6th Internation Confernece on Autonomic and Trusted Computing*, volume 5586, pages 249–267, 2009.

[130] M. Serrano, S. Van der Meer, J. Strassner, S. De Paoli, A. Kerr, and C. Storni. Trust and Reputation Poicy-based Mechanisms for Self-protection in Autonomic Communications. In *6th International Conference on Autonomic and Trusted Computing*, volume 5586, pages 249–267, 2009.

[131] J-S. Shapiro, J-M. Smith, and D-J. Farber. EROS: a Fast Capability System. *ACM SIGOPS Operating Systems Review*, 33(5):170–185, 1999.

[132] J-F. Da Silva, L-P. Gaspary, M-P. Barcellos, and A. Detsch. Policy-based Access Control in Peer-to-Peer Grid Systems. In *6th IEEE/ACM International Workshop on Grid Computing*, pages 107–113, 2005.

[133] A. Simmonds, P. Sandilands, and L. van Ekert. An Ontology for Network Security Attacks. In *2nd Asian Applied Computing Conference*, pages 317–323, 2004.

[134] C. Small and M. Seltzer. Structuring the Kernel as a Toolkit of Extensible Reusable Components. In *4th International Workshop on Object-Orientation in Operating Systems*, pages 134–137, 1995.

[135] J-E. Smith and R. Nair. An Overview of Virtual Machine Architectures. Technical report, 2004.

[136] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture System Support for Diverse Security Policies. In *8th USENIX Security Symposium*, pages 123–139, 1999.

[137] R. Sterritt, M-G. Hinchey, J-L. Rash, W. Truszkowski, G-A. Rouff, and D. Gracanin. Towards Formal Specification and Generation of Autonomic Policies. In *Embedded and Ubiquitous Computing*, volume 3823, pages 1245–1254. 2005.

[138] J. Strassner. *Policy-based Network Management: Solutions for the Next Generation.* Morgan Kaufman, 2003.

[139] J. Strassner, S. Samudrala, G. Cox, Y. Liu, M. Jiang, and J. Zhang. The Design of a New Context-Aware Policy Model for Autonomic Networking. In *International Conference on Autonomic Computing (ICAC)*, 2008.

[140] M. Toure, G. Berhe, P. Stolf, L. Broto, N. Depalma, and D. Hagimont. Autonomic Management for Grid Applications. In *16th Euromicro Conference*, pages 79–86, 2008.

[141] D. Truex, R. Baskerville, and H. Klein. Growing Systems in Emergent Organizations. *Communications of the ACM*, 42(8):117–123, 1999.

[142] K. Twidle, N. Dulay, E. Lupu, and M. Sloman. Ponder2: A Policy System for Autonomous Pervasive Environments. In *5th International Conference on Autonomic and Autonomous Systems*, 2009.

[143] J. Undercoffer, A. Joshi, and J. Pinkston. Modeling Computer Attacks: An Ontology for Intrusion Detection. In *6th International Symposium on Recent Advances in Intrusion Detection*, pages 113–135, 2003.

[144] D.C. Verma, S.B. Calo, and G. Cirincione. A State Transition Model for Policy Specification. *IBM Research Repport*, 2009.

[145] J-P. Walters, Z-Q. Liang, W-S. Shi, and V. Chaudhary. Wireless Sensor Network Security: A Survey. In *Security in Distributed, Grid, and Pervasive Computing*. CRC Press, 2006.

[146] T. Watteyne, D. Barthel, M. Dohler, and I. Auge-Blum. WiFly: Experimenting with Wireless Sensor Networks and Virtual Coordinates. Research Report RR-6471, INRIA, 2008.

[147] M. Weiser. The Computer for the 21st Century. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, 1999.

[148] S-R. White, J-E. Hanson, I. Whalley, D-M. Chess, and J-O. Kephart. An Architectural Approach to Autonomic Computing. In *1st International Conference on Autonomic Computing*, 2004.

[149] M. Wooldridge. Agent-Based Software Engineering. In *IEEE Proceedings on Software Engineering*, volume 144, pages 26–37, 1997.

[150] C. Wright, C. Cowan, and J. Morris. Linux Security Modules General Security Support for the Linux Kernel. In *11th USENIX Security Symposium*, pages 17–31, 2002.

[151] M. Xu, X-X. Jiang, R-S. Sandhu, and X-W. Zhang. Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection. In *12th ACM Symposium on Access Control Models and Technologies*, pages 71–80, 2007.

[152] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control. *International RFC*, 2000.

[153] N. Zeldovich, S. Boyd-wichizer, E. Kohler, and D. Mazieres. Making Information Flow Explicit in HiStar. In *7th USENIX Symposium on Operating Systems Design and Implementation*, volume 7, page 19, 2006.

[154] G. Zhang and M. Parashar. Dynamic Context-aware Access Control for Grid Applications. In *4th International Workshop on Grid Computing*, page 101, 2003.