A requirement engineering driven approach to security architecture design for distributed embedded systems

Muhammad Sabir Idrees

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2012-ENST-045

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Muhammad Sabir IDREES

21/09/2012

Ingénierie des Exigences pour la Conception d'architectures de Sécurité de Systèmes Embarqués Distribués

Directeur de thèse : **Yves ROUDIER**
Co-encadrement de la thèse : **Ludovic APVRILLE**

Jury :

M. Pierre de SAQUI-SANNES, ISAE, France	Président
M. Camille SALINESI, Université Paris 1, France	Rapporteur
M. Frédéric MALLET, Université Nice Sophia Antipolis, France	Rapporteur
Mme. Nora CUPPENS-BOULAHIA, Télécom Bretagne, France	Rapporteur
M. Denis CAROMEL, Université Nice Sophia Antipolis, France	Examinateur
M. Refik MOLVA, EURECOM, France	Examinateur
M. Yves ROUDIER, EURECOM, France	Directeur
M. Ludovic APVRILLE, TELECOM ParisTech, France	Co-encadrement

TELECOM ParisTech
École de l'Institut Télécom - membre de ParisTech

THÈSE

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2012-ENST-045

EDITE - ED 130

Doctorat ParisTech

A doctoral dissertation submitted to :

TELECOM ParisTech

in partial fulfilment of the requirement for the Degree of :

Doctor of Philosphy

Specialty « Computer Science and Networks »

Presented by

Muhammad Sabir IDREES

21/09/2012

A Requirement Engineering Driven Approach to Security Architecture Design for Distributed Embedded Systems

Thesis Advisor : **Yves ROUDIER**
Thesis Co-advisor : **Ludovic APVRILLE**

Jury :

Mr. Pierre de SAQUI-SANNES, ISAE, France	President
Mr. Camille SALINESI, Université Paris 1, France	Reporter
Mr. Frédéric MALLET, Université Nice Sophia Antipolis, France	Reporter
Ms. Nora CUPPENS-BOULAHIA, Télécom Bretagne, France	Reporter
Mr. Denis CAROMEL, Université Nice Sophia Antipolis, France	Reviewer
Mr. Refik MOLVA, EURECOM, France	Reviewer
Mr. Yves ROUDIER, EURECOM, France	Thesis Advisor
Mr. Ludovic APVRILLE, TELECOM ParisTech, France	Thesis Co-advisor

THÈSE

TELECOM ParisTech
École de l'Institut Télécom - membre de ParisTech

«We have instructed man to honor his parents»
[Qur'an 29:8]

To my parents

Acknowledgements

Working on the Ph.D. has been a wonderful and often overwhelming experience. It is hard to say whether is has been grappling with the topic itself which has been the real learning experience, or grappling with how to write papers and proposals, give talks, work in a group, stay up until the birds start singing, and stay focus ...

In any case, I am indebted to many people for making the time working on my Ph.D. an unforgettable experience. I had the pleasure to meet inspiring people that encouraged me, gave me advice, shared their opinion with me, guided me or were simply there when I needed someone to talk to. Without their valuable assistance, I wouldn't have succeeded this thesis. As all these people helped me in their own special way, it would be unfair to put their names in an order. That is why I chose the following way to express my gratitude:

H	O	R	R	J	I	P	E	J	K	C	L	H	Z	F
O	R	L	G	A	D	M	C	L	U	B	N	A	T	H
A	I	S	A	M	I	H	A	E	U	D	Y	N	R	A
N	U	A	S	A	J	I	D	E	A	C	M	D	A	J
F	D	H	E	U	R	E	C	O	M	D	H	R	U	E
G	S	E	B	A	S	T	I	E	N	X	A	I	D	R
A	F	S	A	M	A	N	E	T	O	M	J	K	E	I
B	X	B	Q	R	Y	V	E	S	N	R	E	E	L	N
R	L	F	V	T	O	C	H	A	F	I	R	T	R	E
I	D	E	H	T	D	A	M	G	E	R	O	D	E	Q
E	X	R	A	L	E	S	A	N	D	R	E	Q	N	T
L	R	I	Q	L	U	D	O	V	I	C	H	X	A	C
Z	D	E	B	B	E	L	U	D	O	V	I	C	U	B
S	A	L	I	A	T	I	M	A	C	C	T	E	D	W
R	P	I	N	T	A	Y	Y	B	A	B	V	M	P	H

"The emotional reaction is all that matters.
As long as there is some feeling of communication,
it isn't necessary that it be understood."
– John Coltrane, 1964

Abstract

During the last ten years, the impact of security concerns on the development and exploration of distributed embedded systems never ceased to grow. This is mainly related to the fact that these systems are increasingly interconnected and thus vulnerable to attacks, and that the economic interest in attacking them has simultaneously increased.

In such a context, requirement engineering methodologies and tools have become necessary to take appropriate decisions regarding security early on. Security requirements engineering should thus strongly support the elicitation and specification of software security issues and solutions well before designers and developers are committed to a particular implementation. However, and that is especially true in embedded systems, security requirements should not be considered only as the abstract expression of a set of properties independently from the system architecture or from the threats and attacks that may occur. We believe this consideration is of utmost importance for security requirements engineering to be the driving force behind the design and implementation of a secure system.

We thus describe in this thesis a security engineering requirement methodology depending upon a constant dialog between the design of system functions, the requirements that are attached to them, the design and development of the system architecture, and the assessment of the threats to system assets. Our approach in particular relies on a knowledge-centric approach to security requirement engineering, applicable from the early phases of system conceptualization to the enforcement of security requirements. Our methodology can be seen of as an iterative and complementary co-design process between security requirements and the system architecture. Its main goals are to identify, refine, and trace security requirements with enough expressivity and precision to become a central element throughout the lifecycle of a security architecture. We illustrate our approach with examples from the automotive on-board system domain.

Résumé

Au cours des dix dernières années, l'impact des questions de sécurité sur le développement et la mise en oeuvre des systèmes embarqués distribués n'a jamais cessé de croître. Ceci est principalement lié à l'interconnexion toujours plus importante de ces systèmes qui les rend vulnérables aux attaques, ainsi qu'à l'intérêt économique d'attaquer ces systèmes qui s'est simultanément accru.

Dans un tel contexte, méthodologies et outils d'ingénierie des exigences de sécurité sont devenus indispensables pour prendre des décisions appropriées quant *à* la sécurité, et ce le plus tôt possible. L'ingénierie des exigences devrait donc fournir une aide substantielle à l'explicitation et à la spécification des problèmes et solutions de sécurité des logiciels bien avant que concepteurs et développeurs ne soient engagés dans une implantation en particulier. Toutefois, et c'est particulièrement vrai dans les systèmes embarqués, les exigences de sécurité ne doivent pas être considérées seulement comme l'expression abstraite d'un ensemble de propriétés indépendamment de l'architecture système ou des menaces et des attaques qui pourraient y survenir. Nous estimons que cette considération est d'une importance capitale pour faire de l'ingénierie des exigences un guide et un moteur de la conception et de la mise en œuvre d'un système sécurisé.

Cette thèse décrit une méthodologie d'ingénierie des exigences qui s'appuie sur un dialogue permanent entre la conception des fonctions du système, les exigences qui leur sont attachées, la conception et le développement de l'architecture du système, et l'évaluation des menaces qui pèsent sur ses composants. Notre approche s'appuie en particulier sur une approche centrée sur les connaissances de l'ingénierie des exigences de sécurité, applicable dès les premières phases de conception du système jusqu'à la mise en application des exigences de sécurité dans l'implantation. Notre méthodologie peut être considérée comme un processus itératif et complémentaire de co-conception entre les exigences de sécurité et l'architecture du systéme. Ses principaux objectifs sont l'identification, le raffinement et la traçabilité des exigences de sécurité avec une expressivité et une précision suffisantes pour en faire un élément central tout au long du cycle de vie d'une architecture de sécurité. Nous illustrons notre approche par des exemples tirés de systèmes embarqués d'informatique de bord pour l'automobile.

Contents

List of Figures

List of Tables

List of Publications

- **Journal Papers**
  - M. Dell'Amico, G. Serme, **M. Sabir Idrees**, A. S. de Olivera, and Y. Roudier. *HiPoLDS: A hierarchical Security Policy Language for Distributed Systems*, Information Security Technical Report, ISSN: 1363-4127.
- **International Conferences & Workshops**
  - Y. Roudier, **M. Sabir Idrees**, L. Apvrille. *Towards the Model-Driven Engineering of Security Requirements for Embedded Systems*, 3rd International Model-Driven Requirements Engineering (MoDRE) Workshop, July 2013.
  - M. Dell'Amico, G. Serme, **M. Sabir Idrees**, A. S. de Olivera, and Y. Roudier. *HiPoLDS: A Security Policy Language for Distributed Systems*, 6th Workshop in Information Security Theory and Practice, /Also published also in LCNS, Springer, Volume 7322/2012, London, United Kingdom, 2012.
  - **M. Sabir Idrees** and Y. Roudier. *Effective and Efficient Security Policy Engines for Automotive On-Board Networks*, 4th International Workshop on Communication Technologies for Vehicles - NETS4CARS, /Also published in LNCS, Springer, Volume 6596/2012, Vilnius, Lithuania, 2012.
  - **M. Sabir Idrees**, G. Serme, Y. Roudier, A. S. de Oliveira, H. Grall, and M. Sudholt. *Evolving Security Requirements in Multi-Layered Service-Oriented-Architectures*, 4th International Workshop on Autonomous and Spontaneous Security - SETOP, /Also published in Lecture Notes in Computer Science, Springer, Volume 7122/2012, 2011.
  - G. Pedroza, **M. Sabir Idrees**, L. Apvrille, and Y. Roudier. *A Formal Methodology Applied to Secure Over-the-Air Automotive Applications*, IEEE 74th Vehicular Technology Conference -VTC Fall 2011, San Francisco, USA, 2011.
  - H. Schweppe, B.Weyl, **M. Sabir Idrees**, T. Gendrullis, Y. Roudier, and M. Wolf. *Securing Car2X Applications with effective Hardware-Software Co-Design for Vehicular On-Board Networks*, VDI Automotive Security ["27. VDI/VW-Gemeinschaftstagung Automotive Security"], VDI-Bericht 2131, Berlin, Germany, 2011.

- **M. Sabir Idrees**, H. Schweppe, Y. Roudier, M. Wolf , D. Scheuermann, and O. Henniger. *Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates*, 3rd International Workshop on Communication Technologies for Vehicles - NETS4CARS, /Also published in LNCS, Springer-Verlag, Volume 6596/2011, Oberpfaffenhofen, Germany, 2011.

- G. Serme and **M. Sabir Idrees**. *Adaptive security on service-based SCM control system*, 5th International Conference on Sensor Technologies and Applications - SENSORCOMM, Nice/Saint Laurent du Var, France, 2011.

• **National Conferences**

- **M. Sabir Idrees**, Y. Roudier, and L. Apvrille. *A Framework Towards the Efficient Identification and Modeling of Security Requirements*, 5th Conference on Network Architectures and Information Systems Security - SAR-SSI, Menton, France, 2010.

• **Technical Reports**

- A. Ruddle, D. Ward, B. Weyl, **M. Sabir Idrees**, Y. Roudier, M. Friedewald, T. Leimbach, A. Fuchs, S. Gürgens, O. Henniger, R. Rieke, M. Ritscher, H. Broberg, L. Apvrille, R. Pacalet and G. Pedroza. *Security Requirements for Automotive On-Board Networks based on Dark-side Scenarios*, Deliverable D2.3, EVITA Project, 2010.

- B. Weyl, M. Wolf, F. Zweers, T. Gendrullis, **M. Sabir Idrees**, Y. Roudier, H. Schweppe, H. Platzdasch, R. E. Khayari, O. Henniger, D. Scheuermann, A. Fuchsa, L. Apvrille, G. Pedroza, H. Seudie, J. Shokrollahi, and A. Keil. *Secure On-board Architecture Specification*, Deliverable D3.2, EVITA Project, 2010.

- **M. Sabir Idrees**, Y. Roudier, H. Schweppe, B. Weyl, R. E. Khayari, O. Henniger, D. Scheuermann, G. Pedroza, L. Apvrille, H. Seudie, H. Platzdasch, and M. Sall. *Secure On-Board Protocols Specification*, Deliverable D3.3, EVITA Project, 2010.

- H. Seudie, E. Akcabelen, I. Ipli, H. Schweppe, Y. Roudier, and **M. Sabir Idrees**. *Security Architecture Implementation*, Deliverable D4.3, EVITA Project, 2011.

- **M. Sabir Idrees**, H. Schweppe, Y. Roudier, L. Apvrille, and G. Pedroza. *Test Results*, Deliverable D4.4.2, EVITA Project, 2011.

- R. Douence, H. Grall, I. Mejia, J.-C. Royer, M. Sudholt, **M. Sabir**

Idrees, Y. Roudier, G. Serme, J. Leroux, F. Rivard, and J.-C. Pzzaglia. *Survey and requirements analysis*, Deliverable D1.1, CESSA project, 2010.

– **M. Sabir Idrees**, Y. Roudier, D. Balzerotti, G. Serme, J.-C.Pazzaglia, H. Grall, J.-C. Royer, R.Douence, and M. Sudholt. *State of the art and Requirement Analysis of Security Functionalities for SOAs*, Deliverable D2.1, CESSA project, 2010.

– M. Dell'Amico, **M. Sabir Idrees**, Y. Roudier, G. Serme, A. Santana de Oliveira, and G. Harel. *Language Definition for Security Specifications*, Deliverable D2.2, CESSA project, 2011.

– G. Serme, A. S. de Oliveira, **M. Sabir Idrees**, Y. Roudier, and G.Harel. *Compositional Evolution of Secure Services Using Aspects*, Deliverable D3.1, CESSA project, 2011.

– **M. Sabir Idrees**, Y. Roudier. *Computer Aided Design of a Firmware Flashing Protocol for Vehicular On-Board Networks*, RR-09-235, 2009.

Chapter 1

Introduction

1.1 Context

Designing a secure system has always been a complex exercise. In practice, much of the focus for designers and developers being on delivering a working system in the first place; on the other hand, security concerns have long been considered only in retrospect, especially after serious flaws are discovered. Security experts are thus generally confronted with an existing system, whose architecture might actually hamper the deployment of security mechanisms that would prevent the occurrence of the attacks they envision. An approach that avoids these problems is the development of a security architecture, which is security requirement-driven and which describes a structured collaboration and interrelationship between the architecture design and Security Requirements (SR) to support the long-term needs of the system [138]. The purpose of security architecture traditionally is to bring into focus the key areas of concern, highlighting the decision criteria and security context for each system aspect that has direct or indirect value for a stakeholder. The concept of security architecture encompasses various technical notions wherein security is introduced at different levels of abstraction and based on different mechanisms. Thorn et al. [145] described security architecture as "a cohesive security design, which addresses the security requirements (e.g. authentication, authorization, etc.), and in particular the risks of a particular environment/scenario, and specifies what security controls are to be applied where". To this end, one of the key aspects of security architecture as a tool for secure design is to provide a Security Requirement Engineering (SRE) framework by which more realistic and concrete SRs can be identified and enforced.

From the embedded system viewpoint, this activity, SRE, becomes even more challenging and more critical. These challenges stem from the tight relationship between architecture design and its functional, and non-functional requirements as well as their impact on one another. For instance, if the system architecture design changes or evolve, the SRs should meet the new architecture design objectives. It is especially true when these systems are an integral part of safety critical systems such as automotive systems [129, 9]. This is related to Koscher et al. [80] statement, "automotive systems need not only to be extremely reliable and defect free, but also extremely resistant to the threats and exploitation of vulnerabilities". Specifically, safety applications need to be secured against malicious attacks. Several research ac-

tivities have described potential vulnerabilities and countermeasures in automotive systems, e.g., [51, 14], which we are going to refer to in the rest of this thesis. With some exceptions, most of these efforts consider SRs abstractly, only the requirement identification step, and are not aimed particularly at requirement refinement, and requirement traceability properties. However, there are well-recognized SRE approaches like KAOS [152] or UMLsec [69] that have already shown interesting results in the SRE domain to handle security concerns. Still, before considering these approaches, we first have to make a clear distinction between what we mean by embedded system and what are their functional, as well as non-functional security concerns.

In general, embedded systems are defined as a combination of hardware and software that form a part of some larger system and are generally designed to perform some specific task. More precisely, what makes distributed embedded systems different from general-purpose system are specific features: these systems are resource-constrained in their capacities (and consequently in their defenses). They have reliability and performance concerns, as well as real-time computing constraints. Such systems are often portable or mobile, and they are easily accessible to adversaries at the physical layer. This accessibility has led to several new security attacks in recent years [80, 163, 164]. For example, Koscher et al. [80] demonstrated the ability to adversely control a wide range of automotive functions and completely ignore driver input. These attacks were made by: simply accessing the On Board Diagnostics (OBD-II) port and embedding malicious code into a car's telematics unit. This allows an adversary to virtually control various on-board functionalities – including disabling the brakes, selectively braking individual wheels on demand, stopping the engine, and so on. In addition, the most important aspect of embedded system as defined by Noergaard [100]:

"*. . . none of the elements within an embedded system works in a vacuum. Every element within a device interacts with some other element in some fashion. Furthermore, externally visible characteristics of elements may differ given a different set of other elements to work with. Without understanding the 'whys' behind an element's provided functionality, performance, and so on, it would be difficult to determine how the system would behave under a variety of circumstances in the real world*".

From the security viewpoint, this definition implies that, for an embedded system to be secure, every element as well as its interrelationships with other elements at different abstraction levels (i.e., application, protocol level, middleware level, infrastructure level, storage level, and so on) must be secure. For example, Electric Control Unit (ECU) can rely on a hardware security module to process cryptographic operations; however, if the upper layers (i.e., middleware layer) handle this authentication attribute differently and allow an adversary to fake these attributes (i.e., authentication tickets), the overall security is broken. We will explain this attack in more detail in Chapter 5. In particular, we can identify such security

weaknesses and concerns by examining the subtle interactions and collaborations between layers. Similarly, several approaches [79, 55, 164, 161] have shown that we cannot solve embedded system security at a single level of abstraction. Therefore, it is natural to develop a security requirements specification by focusing on distinct characteristics of embedded systems and particularly considering a layered (modular) representation of an embedded system architecture, which will essentially help us to develop modular security architecture. In this context, the current state of the art approaches to SRE, like KAOS and UMLsec, are falling short in capturing the core essence of the embedded system architectures. For instance, the KAOS framework concentrates mainly on goal satisfaction and on the synthesis of behavior models [151] and does not consider the system architecture of the system. For instance, in KAOS framework it is difficult to capture and model an architecture, and even less so multiple architectural layers. In contrast, UMLsec, which is a model driven engineering approach, considers both the structural and behavioral aspects of SRs. In essence, this approach considers that well formed requirements have already been elicited and refined down to the design level through the definition of the normal behaviors of the system components. UMLsec more specifically focuses on the refinement of those security requirements into security mechanisms. A detailed analysis of these approaches as well as other scientific contributions in this domain are presented in Section 2.2.

1.2 Thesis Contributions and Outline

In this dissertation we propose a Security Requirement Engineering (SRE) approach that enables the design of security architecture for embedded systems. We in particular focus on security-related knowledge acquisition and management through the definition of a SRE process that makes it possible to design a system that would intrinsically be secure from its design on. Our approach is composed of three successive parts.

1. **A Knowledge-Centric Approach to Security Requirements Engineering:** In the first phase of this dissertation, we present the main building blocks of our proposed SRE methodology and discuss its integration with a system engineering modeling language.

    - In Chapter 2, we systematically analyze various sources such as security standards, a set of methodologies representative of the current state of the art approaches, in order to build a unified SRE methodology. The proposed methodology demonstrates how the capabilities of different SRE models and approaches can be integrated through a knowledge-centric SRE process. In addition, we take the key concepts defined in these approaches and build security ontologies for each concept in order to

guide our SRE process with a knowledge base. This will make it possible to analyze different security concepts and enables a particular way of structuring, reusing, and sharing security related knowledge base within the SRE process. Although our proposed methodology is dedicated to embedded systems, it is still flexible enough to be adapted to any general-purpose system architecture like Service Oriented Architectures (SOA) [115], and capable of producing accurate security requirements. The results are presented in [60].

- In Chapter 3, we first explore the capabilities of SysML, the System Modeling Language [107] for supporting our knowledge-centric SRE methodology. The SysML is an OMG [106] standard for modeling system engineering applications and has sufficient expressiveness to describe detailed system design. However, a major obstacle toward the usage of SysML is the lack of support from the security viewpoint. To take benefit from the capabilities of SysML, we proposed several extensions to the SysML semantics to integrate our security concepts. In particular, we have integrated the security requirement and the security attack diagrams. Moreover, we enriched these diagrams with our proposed ontological concepts such as a controlled vocabulary. The use of ontologies within modeling languages offers a concrete opportunity to reason about the correctness of these models. Furthermore, we implemented these capabilities into the TTool [82] engine that also supports our extended SysML model for defining security requirements and attack tree modeling. This tool readily supports the iterative methodology that we advocate.

2. **By Design Security Requirements Engineering:** In the second part of this dissertation, we present each activity of the Security Requirement Engineering Process (SREP) in more detail and explain how a security related knowledge base is generated and shared among all activities.

   - In order to illustrate the different parts of this thesis, we introduce in Chapter 4 a running example used all along the thesis to explain our proposals. The example is originating from the secure design of a vehicular embedded system developed in the European project EVITA [117]. The case study has been developed for illustrating the secure firmware update process.

   - In Chapter 5, we address the problem of identifying security attacks and vulnerabilities in the context of a multilayered system architecture, where the security related information is generated, processed and stored at different layers. The idea is to extract the knowledge about different system activities spread across and that corresponds to various system develop-

ment activities, and to use this knowledge for security analysis purposes. In particular, we use the knowledge bases relying upon different ontologies such as the system architecture ontology, the goal ontology, etc. to analyze the security of the system and to specify how an adversary can attack the system. Furthermore, the concept of a knowledge based attack tree is brought in as the foundational graphical representation for attack modeling.

- In Chapter 6, we illustrate the approach in the context of security requirements identification and refinement, and present a way to trace security requirements. We describe first the security requirement identification process, which makes use of the different knowledge bases produced in different phases of the SREP. It enables us to discover security requirements from early system development stages and in relation to different available knowledge bases. Then, we propose the concept of dependent refinement model to address some shortcomings and limitations of existing approaches to SR refinement. Finally, we propose an approach for tracing requirements in order to determine the source of requirements and the reason about requirements existence. We in particular use our extended SysML security requirement diagrams to model and share SR related knowledge.

3. **Security Requirements Enforcement:** In the third and final part of this dissertation, we handle SR enforcement issues and propose solutions for designing and deploying cryptographic protocols and for enforcing access control related security requirements.

   - In Chapter 7, we propose an approach based on the use of cryptographic key material protected with inexpensive hardware to build the firmware flashing cryptographic protocol specification. We show how a root of trust in hardware can sensibly be combined with software modules. These modules and primitives have been applied to show how firmware updates can be done securely and over-the-air, while respecting existing standards and infrastructures. Despite the fact that a trusted platform model entails certain constraints, such as the obligation to bind cryptographic keys to a given boot configuration, we show how the protocols we presented deal with the update of the platform reference registers during the boot phase of an Electronic Control Unit – ECU.

   - The final contribution of this thesis, in Chapter 8, is dedicated to the enforcement of access control related security requirements. We have proposed and developed a policy decision module that is used to enforce various access control rules by deploying multiple policy enforcement points at the different levels of system abstraction. We discuss how to design

policy engines that implement an effective enforcement in such architectures despite the complexity of the protocol stacks of on-board electronic control units. It also evaluates how policies expressed in XACML can be adapted to the efficiency requirements of the automotive environments despite the limited computational power of those units and their network bandwidth limitations.

Part I

A Knowledge-Centric Approach to Security Requirements Engineering

Chapter 2

Overview of Security Requirements Engineering Methodology

2.1 Introduction

A very important part in the security architecture design for the achievement of secure systems is that known as security requirements engineering which provides techniques, methods and standards for tackling this task in the system development cycle. Security requirements engineering frameworks derive security requirements (SR) using various security-specific concepts, borrowed from security engineering paradigm [4]. For instance, security requirements stem from potential adversaries that attempt to compromise the system [62]. Security goals [151] are another concept in SR literature defined as a prescriptive statement of intent, which expresses some security objective to be achieved by the system. In addition, security requirements are derived from analysis of interactions and dependencies of system models and the subjects of the attacks [69]. These different views capture a certain types of information and results in different types of SRs and security design solutions. For example, some approaches [153, 93, 86, 141] evaluate system from the behavioral perspective by building obstacle and threat models and exploring resolutions, security requirements, to enrich and update the system behavior. In approaches such as [69, 88, 45], security requirements are derived to focus on the different aspects of the system from structural viewpoint. However, as previously mentioned, the tight relationship between different architecture layers of an embedded system requires the security engineers take the collaborative SRE approach into consideration to extract and enforce security requirements. This requires that SRE framework expand the analysis from the problem space to the solution space as well. Nevertheless, security requirements are not just related to identification, and prioritization or refinement. Security requirement traceability is yet another important issue: providing a rationale for the definition of finer-grain requirements is necessary to understand whether a given requirement is still necessary if associated assumptions about the environment, an attacker, or even the system architecture change, for instance. We need to fill a gap from requirement identification to requirement enforcement, verification and to testing. Establishing relationships between require-

ments and such later phases of engineering should thus receive appropriate support: for instance, it should be possible to document the fact that some security mechanism is introduced in order to satisfy one security requirement, or to point at some test over the implementation in order to verify that it is compliant with the same requirement. That is to say, security requirements probably constitute one of the most abstract documentation of the expected system behavior. These requirements should provide a specification that has to be satisfied at every subsequent stage of the system: analysis, design, implementation, and validation/testing. This leads to another important challenge for SRE to cope with inconsistent and incomplete security requirements specifications.

In this chapter, we first start from reviewing and analyzing different approaches to SRE and their strengths and weaknesses with respect to aforementioned design objectives. In Section 2.3, we look at ontologies in the requirement engineering, how and what benefits we can achieve by using the ontological concepts (i.e., knowledge acquisition and management) in requirement engineering process. In Section 2.4, we proceed to defining a unified SRE methodology for the use of ontologies in security requirement engineering process. Section 2.4.2, presents security ontologies for each security class identified in the SRE methodology, including design objectives, domain and scope, and detail descriptions. In Section 2.5, general conclusions concerning the functionality of security requirements engineering process are drawn.

2.2 The State of the Art SRE Approaches

This section reviews the existing approaches for eliciting, modeling, and analyzing security requirements. The goal of this section is to investigate the capabilities as well as shortcomings of the state of the art SRE approaches and to extract the core artifacts defined in these approaches for driving security requirements. We study how different approaches for deriving and expressing security requirements result in different expressions of requirements.

2.2.1 Goal Oriented Approaches

Goal oriented approaches focus on the concept of a goal or objective for eliciting, elaborating, structuring, specifying, and modifying security requirements. In this category, we review two frameworks:

- **KAOS [153]:** was the first to feature a goal-oriented approach for modeling, specifying, and analyzing requirements. KAOS is a requirement engineering method concerned with the elaboration of the objectives to be achieved by the system-to-be. In particular, KAOS takes into consideration that there are

multiple stakeholders in and multiple views towards a system-to-be. These views here do not refer to the differing views of the stakeholders, but to the goal, object, agent, system operation, obstacle, and agent behavior models – each model stands for a different view of the system as shown in Figure 2.1. The main purpose of KAOS is to ensure that high-level goals are identified and progressively refined into precise operational statements. Along this, various alternative goals and responsibility assignments are considered until the most satisfactory solution is chosen. KAOS extended to security has been introduced in [151]. This approach extends an earlier framework on eliciting goals and identifying potential obstacles to satisfying goals to security engineering. The security obstacles are called anti-goals and are similar to the idea misuse cases [141], which are the attackers goals and malicious obstacles to security goals, set up by the attackers to threaten security goals. Anti-goals are refined to form a threat tree, in which the leaf nodes are either software vulnerabilities or anti-requirements.

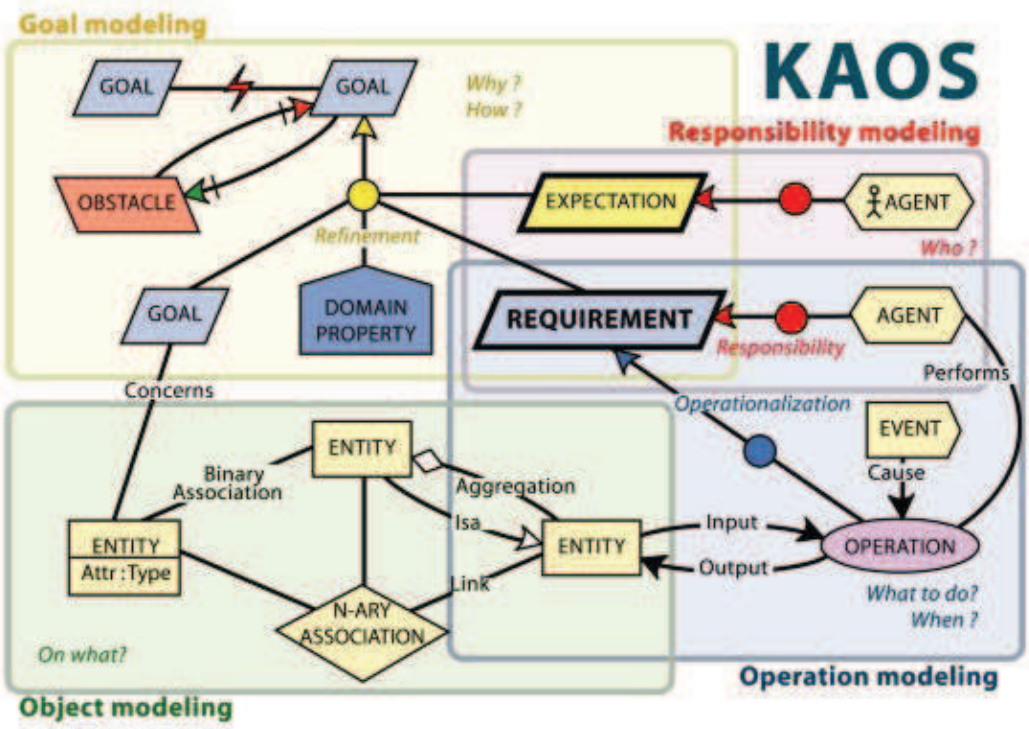


Figure 2.1: KAOS security requirements metamodel (taken from [103])

All requirements in KAOS are written by default using semi-formal graphical notations and, if needed, using formal notation. In [26], further features a formalization of KAOS requirements definitions using linear time temporal logic. This representation makes use of generic refinement patterns to decompose goals into a set of sub-goals. However, a major limitation of the approach (and the framework) results from the fact that at the highest level of abstrac-

tion, the system behavior is only characterized by focusing on a particular functionality of subject/object of analysis. Thus, goals may be insufficient for analyzing all the security concerns, especially when lower-level security requirements have to deal with concrete details of the system architecture. As pointed out in [45], it is required to incorporate approaches for reasoning about the behavior of contexts and how that behavior contributes to satisfying or violating security requirements. More precisely, since the output of SRE is a set of required protection mechanisms and constraints on the system-to-be, the need for building tight relationship between architecture design and security requirements and impact of security mechanisms on other requirements is absolutely imperative.

- **Secure Tropos [93]:** introduce extensions to Tropos [13] for incorporating security concerns into the goal-oriented development process. Tropos defines four requirement development phases in which each successive phase refines the high level description from the previous phase to a lower level towards implementation as shown in Figure 2.2. In this enhancement of Tropos, security constraints, secure dependencies, threats, and security goals, tasks, and resources are introduced and added to the Tropos modeling notation. In this approach, secure entities are tagged with an "S" (see Legend in Figure 2.2) to indicate those tasks, goals, and softgoals are security related. In particular, security requirements are described as constraints on the functionalities. In [90] security concerns are integrated into all phases of Tropos agent-oriented methodology: from early and late requirements, and architecture and detailed design. At the early requirements phase, Security Diagram is constructed and security constraints are imposed to the stakeholders. During the late requirements stage, security constraints are imposed to the system-to-be in the Security Diagram. The system is presented as one or more new actors, who have a number of dependencies with the other actors of the organization. In the architectural design stage, security constraints, secure entities that the new actors introduce, and secure capabilities are identified and assigned to each agent of the system. This approach follows a step-by-step refinement construct, where goals are formulated at different levels of abstraction, ranging from high-level, strategic concerns to low-level, technical concerns. However one general concern about this framework is that it does not provide means for propagating changes between the different levels of abstraction. For example, if there is a change in a organizational model, which includes relevant actors and their respective dependencies, there is no systematic way of relaying such changes. A clear interaction relationship between the models would provide a systematic way of propagating changes between the different models and hence support maintaining security properties as requirements evolve.

Summary: In general, the goal-oriented approach is a natural way of expressing security requirements that refine other more abstract security requirements. While this is an important strength of that model, those approaches generally

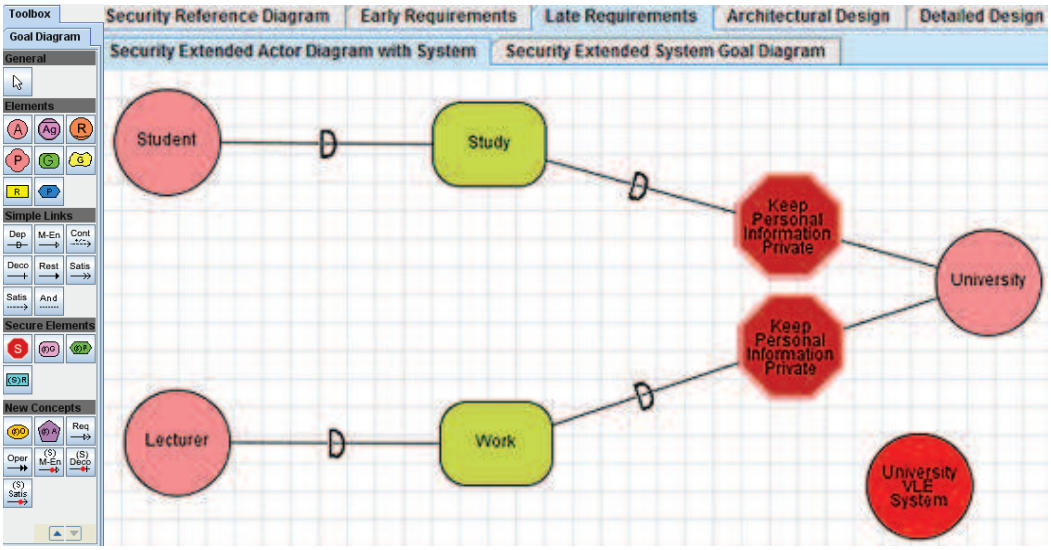


Figure 2.2: SecureTropos security requirement diagram view (taken from [113])

assume a rather static model of the system architecture. Another strength of goal oriented approaches lies in their ability to capture dependencies between security requirements; however, how those dependencies may evolve when security requirements are refined is generally ignored by those approaches, especially if the refinement is dictated by a refinement of the system architecture.

2.2.2 Model Oriented Approaches

In contrast to goal-oriented frameworks, the general concept of model-based approaches is underlined by the definition of architecture. Security requirements are expressed through the architectural concepts described, at different levels of abstraction. In particular, those requirements arise in that approach from the identification of security concerns about system components or the way they interact. We review two model based approaches in the following.

- **UMLsec [69]:** is an extension to UML that allows expressing security relevant information within UML diagrams. The main uses of such approach are first, to encapsulate knowledge and make it available to developers in form of a widely used design notation, and secondly, to provide formal evaluation to check if the constraints associated with the UMLsec stereotypes are fulfilled in a given specification. More precisely, UMLsec goal is to define a universal set of stereotypes and tags that encapsulate security design knowledge to be used as part of UML diagrams. In [71] combines the use of UMLsec modeling, Use Case driven process, and goal trees to design the system along with modeling functional and non-functional requirements respectfully. In this method, the

goal tree is developed to record the result or reasons of design actions, which are expressed in UMLsec diagrams. The security goals are refined in parallel by giving more system details, such as UMLsec stereotypes or tag-values, in design phases. However, UML is not a requirements engineering notation, and the only diagram that focuses on the expected functionalities from the users point of view is the use case diagram. The resulting models do not express attackers' behavior, and threat description is limited to using the specific stereotypes (i.e., Delete, Read, Insert) to changes a state of the subsystem. Therefore, the usefulness of the modeling constructs is based on the expressiveness and comprehensiveness of the stereotypes. Moreover, UMLsec also assumes that the system architecture is defined to a large extent. In essence, the methodology considers that well formed requirements have already been elicited and refined down to the design level as normal behaviors of the system components as shown in Figure 2.3, and there exists some system design to satisfy them. More precisely, UMLsec more specifically focuses on the refinement of those security requirements into security mechanisms and their verification.

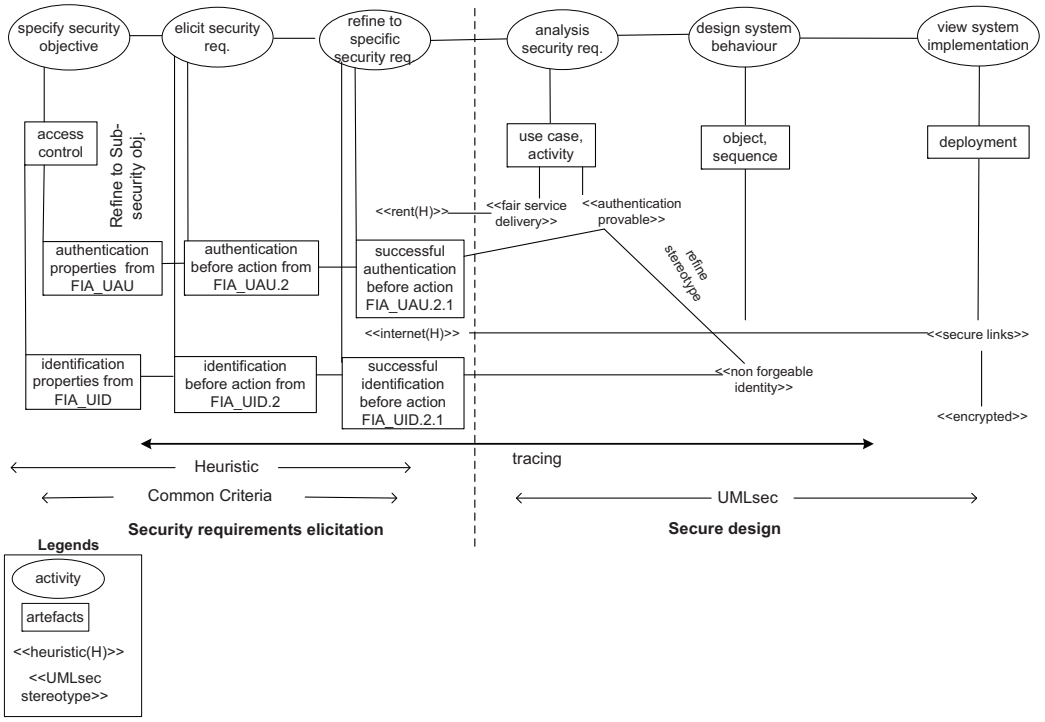


Figure 2.3: UML diagrams and UMLsec stereotypes (taken from [54])

- **SecureUML [88]:** is another UML-based modeling language for the model-driven development of secure, distributed systems based on the Unified Modeling Language (UML). SecureUML takes advantage of Role-Based Access Control (RBAC) for specifying authorization constraints by defining a vocab-

ulary for annotating UML-based models with information relevant to access control. In particular, their approach focuses on embedding role-based access control policies in UML class diagrams (see Figure 2.4) using a UML profile. The UML profile defines a vocabulary for annotating class diagrams with relevant access control information. From a SRE perspective, and quite similarly to UMLsec, SecureUML focuses on a later phase of software development than the goal oriented security requirement approaches. As can be seen from SeureUML metamodel presented in Figure 2.4, the SecureUML methodology does not consider security goals, domain knowledge, potential attacks and vulnerability analysis, and focus on only authorization constraints and access control requirements. SecureUML does not consider SRs (in the sense of SRs in the conceptual framework) elicitation, completeness of the set of requirements, refinement, nor traceability and coflicts of requirements. Furthermore, this approach does not provide a systematic way of building relationship between different security elements (i.e., high level security objectives and security requirements, security attacks and security requirements), which is an important aspect for designing security solution for embedded systems. Thus, SecureUML can be considered as a notation to specify and design secure software systems, rather than a SRE method.

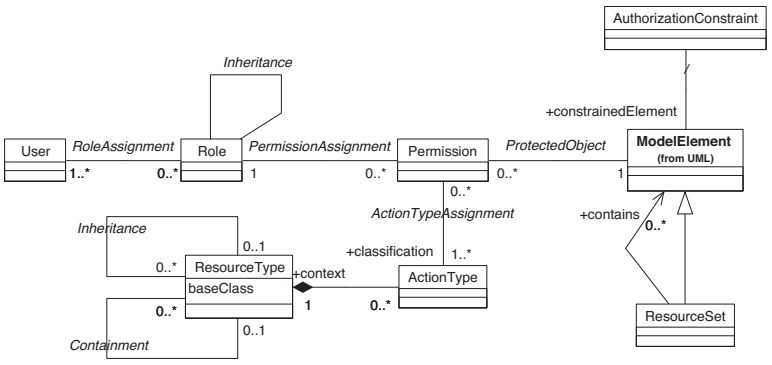


Figure 2.4: SecureUML metamodel (taken from [88])

Summary: This discussion highlights a major limitation of the model-based approach to security requirement engineering in that it mainly intervenes in relationship with the system architecture design and focuses on linking low-level security requirements with security mechanisms that would satisfy them. Conversely, this feature also depicts the main strength of this approach, which is perfectly aligned with the fine grained design of embedded system architecture.

2.2.3 Problem Oriented Approaches

Problem oriented approaches to defining security requirements focus on the definition of threats and how security requirements can be extracted from their identification. We review the following two approaches:

- **Abuse Frames [86]:** is based on the Jackson's problem frames approach [68] and is intended to analyze security problems in order to determine security vulnerabilities and to derive security requirements. This approach introduces the notion of anti-requirement (similar to the concept of an anti-goal [151]) to describe the behavior of a malicious user that can subvert an existing requirement. The basic idea behind the definition of abuse frames is to bind the scope of a security problem with anti-requirements in order to derive security requirements. Such explicit and precise descriptions facilitate the identification and analysis of threats, which in turn drive the elicitation and elaboration of security requirements as shown in Figure 2.5. However, it does not provide any specific techniques or approach to deal with security requirement refinement as well as requirement traceability.

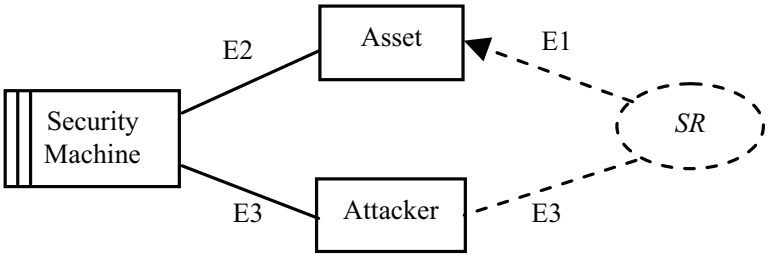


Figure 2.5: Abuse frame diagram (taken from [85])

- **Misuse cases [141]:** extend the traditional use case approach to also consider misuse cases, which represent behavior not wanted in the system to be developed. Misuse cases are initiated by misusers. A use case diagram (see Figure 2.5) contains both, use cases and actors, as well as misuse cases and misusers (notated in black color). Development of misuse cases allows the identification of security attacks and associate security requirements during application development. In [162], authors present a formal representation of misuse cases and provide an intuitive way to executable misuse case model. Although misuses cases are not entirely problem-oriented as they represent aspect of both problems and solutions, they have become popular as a means of representing security concerns in the early stages of software development. However, they are limited by the fact that security attacks and requirements are only analyzed and derive through use case specification. The completeness of the security requirements analyzed through scenarios is not guaranteed as other scenarios by which the security of a system could be exploited may be left

out. Furthermore, the approach does not consider validation, verification, conflicting requirements, or the interaction of security and other non-functional requirements.

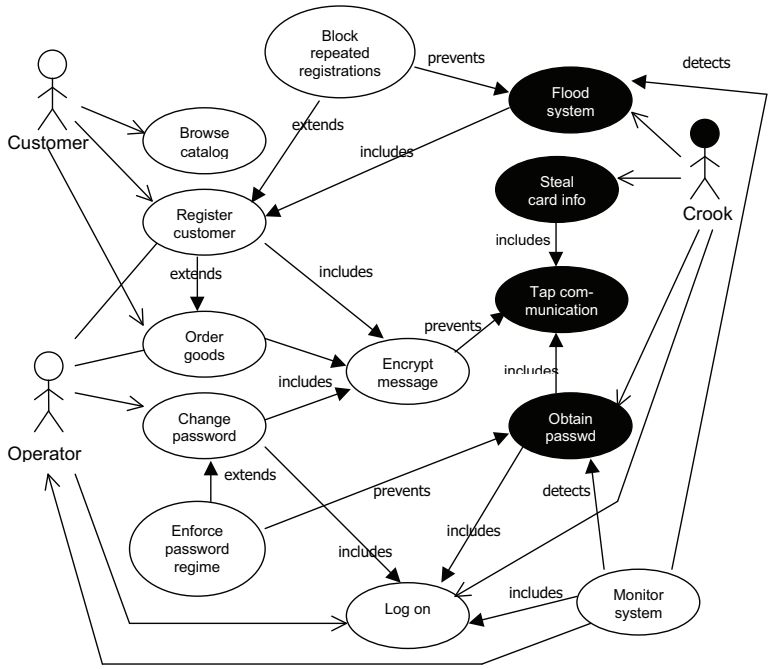


Figure 2.6: Use case diagram containing misusers and misuse cases (taken from [141])

Summary: The main liability of those problem oriented approaches is that they need both a very detailed description of the system architecture as well as a detailed knowledge of known vulnerabilities that may be present at each of the components of this architecture. While they fit well the needs of security certification, in which the security properties expected from a system are assessed, they are much less appropriate to the definition of a brand new secure system.

2.2.4 Process Oriented Approaches

Process oriented approaches focus on the analysis of security requirements throughout the system design. These approaches involve identification of threats and vulnerabilities, identification and exploration of security requirements for addressing identified weaknesses, risk analysis, and the verification of security properties. The SQUARE methodology is the most prominent proposal in this category:

- **SQUARE [91]:** is a comprehensive methodology for SRE. Its aim is to integrate SRE into software development processes [91]. SQUARE stresses applicability in real software development projects and thus provides an organizational framework for carrying out SRE activities. It is assumed that SQUARE is carried out jointly by requirements engineers as well as by the stakeholders. The SQUARE methodology is composed of nine steps to provide a mean for electing, categorizing, and prioritizing security requirements for information technology systems and related applications. However, the definition of SRs in the SQUARE methodology considers requirements at the system or software level. This definition does not consider the properties and behavior of the context in which an application operates. According to [68], a more concise definition of SRs should consider their context of operation as satisfaction of a requirement is expressed in terms of the state changes in the context. Moreover, the steps provide by the SQUARE methodology are "waterfall model" in nature, and this does not make a provision for iterations to revise SRs and support the evolution of a system [95]. Although SQUARE claims to operator with software engineering activities, its main drawback may originate from the lack of integration and consistency between those different models from the point of view of SRs. Furthermore, SRs are described using text-based description (see Table 2.1): this makes it even more difficult to integrate SRs with other system models as well as to organize the set of SRs into a description with different levels of complexity. Another disadvantage is the lack of support for SR traceability making that approach not suitable for complex systems.

Goal (s):	The claimed identities of all users and client applications will be authenticated before they are allowed access. Protect from unauthorised attacks involving addition, modification, deletion, or reply of data in network.
Category:	Authentication
Requirement(s):	AN-1) Authentication control mechanism shall be enforced in production environment. Authentication control will be done on user name and password or other user credentials.
No.	AR-01
Misuse case:	MC-01
Architectural Recommendation:	All shared drives on the network should enforce authentication policies.
Implementation Choices:	In IIS 6/0, the IIS Manager contains a check box that permits the Administrator to omit the user name and password. If no user name and password are specified, IIS uses the requesting user credentials when the Administrator is using an authentication method that can perform delegation to authenticate to the remote share.

Table 2.1: SQUARE security requirement template (taken from [91])

Summary: This approach is interesting in that it comprehensively and consistently combines different phases of requirement engineering like threat model,

requirements elicitation, risk analysis, and requirement prioritization in order to achieve a more precise and multi-facetted description of SRs. However, the hard coding of a particular software development methodology also has a strong impact on the SRs: the example of the SQUARE waterfall model inadequacy illustrates, in particular, the need for an iterative approach in a **process** oriented evaluation of SRs.

2.2.5 Conclusion

We have reviewed the state of the art approaches to SRE. The approaches have been classified into goal-oriented, model-oriented, problem-oriented, and process-oriented. Table 2.2 presents a summary of these approaches and brief summaries of their main characteristics.

Conceptual Classification	**Security Requirement Approach**	**Security Specific Charateristics**
Goal Oriented	KAOS	Elicitation of security goals to counter anti-goals
	Secure Tropos	Identification of malicious actor's goals and plans, and analysis of each actor's security constraints.
Model Oriented	UMLsec	Refinement of security requirement into security mechanisms and their verification.
	SecureUML	Identification of authorization constraints
Problem Oriented	Abuse Frames	Identification of abuse frame concerns which need to be addressed for an attack to succeed. Security requirement for counteracting threats are expression a problem frame.
	Misuse cases	Misuse cases to address which behavior is not wanted in the system and associate security requirements during application development.
Process Oriented	SQUARE	Misuse case, attack scenarios, goals, and elicitation of security requirements from potential risks.

Table 2.2: Summary of security requirement engineering approaches

In general these approaches to SRE involve two main phases in their methodology/framework, namely (1) identification of security threats and (2) designing mitigation strategies to remove the possibility of threats causing harm to assets. In particular, most of these approaches consider different artifacts (i.e., goal, models, system behavior, risk, etc.) for identification of security requirements at different level of system conceptualization. However, as previously mentioned in the introduction, challenges unique to embedded systems require an integrated approach to SRE covering all aspects of embedded system design from architecture to implementation.

2.3 The Role of Ontologies in RE

The above arguments motivated us to look at ontologies as a solution to overcome the shortcomings and improve the state of the art approaches in this area. The main objective of ontology is to define an explicit formal specification that try to eliminate, or at least reduce, conceptual and terminological confusions in order to have a shared interpretation about security terms and concepts. Also, it provides a way to define dependencies and relationships among captured and stored knowledge. The use of ontologies for precisely expressing and building requirement knowledge and its relationships are under discussion since a long time ago [84, 12, 81, 127, 43]. The key to these approaches appears to relate to the creation and maintenance of ontologies for requirements that can be easily updated and utilized in a systematic way. These approaches addressed many diverse synergies between RE and ontologies and raised issues that can be solved by the use ontologies such as, traceability, completeness, consistency, unambiguous requirement specification, and managing evolution of requirement. Building specifications (documentation) is seemingly the most commonly evaluated application of ontologies [36]. The purpose of using ontologies in expressing requirement specification is to improve the structure of the document as well as having a concise vocabulary of terms and concepts used during the documentation. Notwithstanding, the requirement specification activity could additionally be benefit from ontologies by developing intelligent tools for requirement annotation that will for instance have capabilities for verifying validity of requirement specification with respect to the developed requirements artifacts. Ontologies also offer to have a clear semantic and interrelationships between different developed artifacts, and thus help in building requirement specification as intelligent knowledge vocabulary [122]. Actually, great expressive power of ontologies helps us to achieve several characteristics related to the semantics of a requirement specification (e.g., unambiguity, correctness, consistency, etc.).

The use of ontologies for checking requirement consistency as well as its particular support for managing the design rational for requirement engineering such as traceability, and verification properties is probably one of the areas that have attracted a lot of attention so far [140, 84, 166, 167]. One clear example is adapted by Siegemund et al. [140], for checking consistency and completeness of goal oriented requirement specification. They combined ontology consistency checking and rule driven completeness checks to measure the validity and consistency of the requirement models. Similarly, Lin et al. [84] raised several issues regarding the requirement specification that must address by requirement model such as traceability, completeness, managing the evolution of requirement specification, etc. In response to issues, they described an ontology driven solution for generating unambiguous, and precise requirement specification that can be easily extendable and support dependencies and relationships among requirements. Cranefield [23] promotes the synergy of ontologies and their association with software modeling languages such as UML. He described an approach to take benefit from the use of ontological reasoning to reason

over UML models. Given a lot of attention to model driven requirement engineering, ontologies are definitely a promising technology to reason about generated MDE models and their dependencies in terms of interactions and collaborations. As well as those mentioned above, ontologies could also be used to share and reuse requirement [28, 155] related knowledge with other models that are relevant to requirements.

From the security engineering point of view, the development and usage of ontologies in different contexts and for many purposes (i.e., risk assessment [30], threat and vulnerability ontology, security management [146, 33], security protocol designs [77], policy configuration [10], security requirements [73, 35, 33], etc.) has shown the strength and capabilities of ontologies to build the non ambiguous definition of terms representing the knowledge of the security. For instance, Fenz et al. [33], proposed several security ontologies to structure the security related knowledge like threat, assets, vulnerability, etc. However, this approach is only limited to knowledge repository and does not consider how the knowledge is generated and shared among different activities involved in the security engineering process. Tsoumas et al. [146] present a security ontologies using OWL and proposed a security framework in order to support security knowledge acquisition and management. Like others, this approach is also limited to "what" part of security requirement engineering then "how" the security requirement related knowledge is generated and used. These approaches and methodologies are well recognized in security communities and already showed interesting results for some of the security requirement engineering process (i.e., threat, risk, etc.). However, as pointed out in [142], the security ontologies vary a lot in the way they cover security aspects; the security requirement design, integration, implementation and maintenance are almost the dark side of security ontologies. In addition, none of these approaches analyze and evaluate collaboration of security ontologies in different aspects of SRE by following a comprehensive SDLC. Further exploration of semantic annotation mechanism of security requirements, integration of ontologies and meta modeling architectures, and a comprehensive security requirement engineering methodology, are some of the biggest challenges conceding the aspects of security knowledge acquisition and management.

2.4 Security Requirement Engineering Methodology

The goal of this section is to define security requirement engineering methodology, describe typical security engineering development lifecycle, security classes used and produced security metadata (i.e. the knowledge produced by each security class) in each activity, their dependencies and interrelationships. Based on the discussions, we define a unified methodology for the use of ontological concepts in security requirement engineering process. More precisely, we aim at building SRE methodology that is guided not only by a process but also knowledge about the each activity within the process is also developed and shared among other activities. We start

from defining security requirement engineering as an application context for ontologies, and proceed to defining a methodology that identified places in SREP where ontologies can contribute to improve current state of security requirement engineering. In this context, we take the view these state of the art approaches are primarily based on the same (small number of) artifacts. Our main intention is to identify the core artifacts of the security requirement and harmonize them together. In this context, we extracted the essence of the state of the art SRE models (cf. Literature review presented in Section 2.2), the core artifacts, and then aligned them semantically by defining ontology to construct a unified security requirement engineering model. The essential artifacts that we identify as common to SRE models are: goals, security attacks, system models (behavioral and structural), and use case oriented models. We order these artifacts under the form of security classes where each class is represented by ontology. We consequently have organized the structure of these security classes in such a way that the security metadata produced by these security classes, in different phases of SRE process, can be easily shared and reused among different activities of the process. For example, security goal related metadata could be used for identifying the system assets as well as this metadata could be used for analyzing the security attacks and vulnerabilities. Figure 2.7, resumes the ontology-driven security requirement engineering methodology.

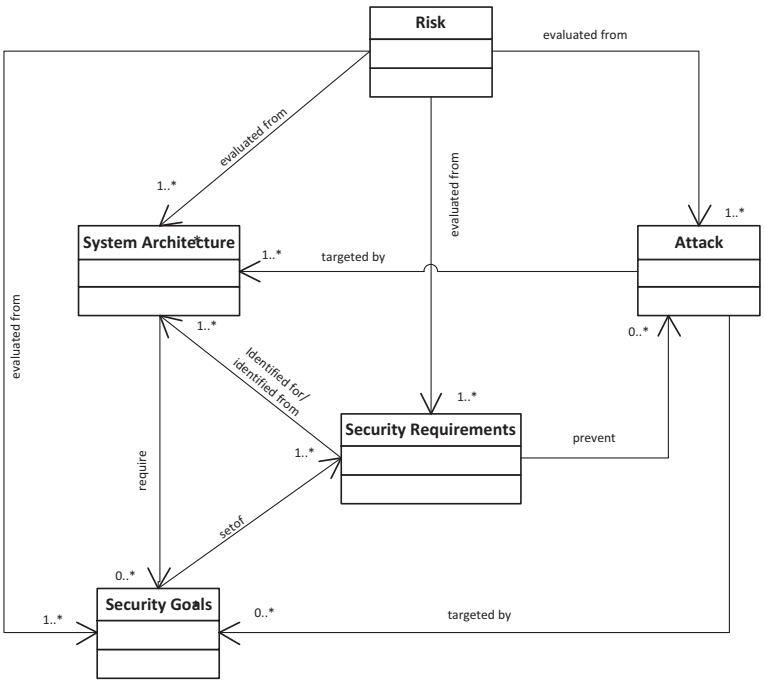


Figure 2.7: Ontology-driven security requirement engineering methodology

Our analysis have revealed that multiple SRE models can be expressed in terms of the security ontologies and their associated relationships that we identify, that the

degree of overlap amongst existing models is significant, and that many "novel" SRE models can potentially be developed by simply combining and sharing knowledge of these diverse models in systematic ways. Nevertheless, identifying a common SRE model is also desirable because it allows for various general syntaxes, relationships bindings and common semantics, to be developed in terms of the generic model, which facilitates the sharing of SR at the different levels of abstraction. For example, initially identified security requirements, through problem-oriented approaches, can be fed to models like UMLsec, which focuses on the verification of these requirements in terms of security mechanisms.

In a similar way, the output of goal-oriented approaches can be combined with model driven engineering approaches to build the process-oriented approaches. This is the case of SQUARE methodology, where SRE process starts with identifying the goals and then defining system architecture in order to identify requirements. Thus, we can facilitate collaboration of SRE models (e.g., mutual understanding among diverse models collaborating in RE process, especially in the context of system-wide security engineering). However, the major obstacle in having such a collaborative approach is that none of these SRE approaches usually allow for using different approaches collaboratively, since they are mainly constrained by the modeling languages and tools they use. Recognizing this problem, we decided to use the SysML, The System Modeling Language [107], which promotes cohesiveness between the models generated during different phases of system engineering and creates a shared understanding from multiple dimensions [50]. The SysML standard supports: "the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems"[107]. Although all of these feasters of SysML demonstrate many capabilities to different aspects of system engineering, none of them analyze and evaluated applications of SRE. In this context, we have proposed several extensions to SysML in order to integrate security aspects along with other system wide development activities. The central point of these extensions is to utilize the system engineering models along with the SRE and the associated security aspects (i.e., verification, testing, etc.) in a single viewpoint.

In order to make this chapter lighter, we only give an overview of knowledge driven security requirement engineering process and its associated security ontologies to which we are going to refer in the rest of the thesis. Other important features of our methodology are presented in the next chapter, namely the modeling language, SysML, and its support for our Knowledge-Centric SRE Process.

2.4.1 Knowledge-Centric SRE Process

In this section, we first introduce the knowledge driven security requirement engineering process (SREP) whose focus seeks to identify SR from the early design phases of the system conceptualization. Instead of following the general approaches like waterfall style [20], we build our SREP on the definition of iterative an in-

cremental construction of the security requirement specification. Then, we define security ontologies for each security class used during the SREP. The SREP is articulated into seven activities, which are iteratively performed throughout the system conceptualization, although with different focus depending on where the iteration is situated within the system development lifecycle. These activities are as follow:

1. **Agree on Definitions:** The first activity in the SREP is to define and to agree upon a common set of security terms and definitions. It helps to build a common knowledge base for the experts (i.e., system engineer, risk experts, security experts, verification and testing teams, etc.) involved at different stages of system development life cycle. The introduction of, as well as the agreement on some general terms and principles of IT security within a given RE process has proven to be very beneficial for all system wide activities: analysis, design, implementation, and validation/testing [62]. We may use the security terms and definitions mainly defined in security standards (i.e., ISO/IEC 15408, ISO/IEC 17799:2005, or ISO/IEC 27002:2005, etc.) to build a common knowledge base in order to bridge a potential miscommunication gap across different SRE phases. Appendix A, presents how some of the security terms are generally defined and understood in this thesis.

2. **Identify Security Goals:** The purpose of this activity is for the stakeholders to formally agrees on a set of concise and abstract statements of the intended solution (goals) to the problem defined by the security problem definition [20]. More precisely, goal captures stable information (correct behavior) and provides means to distinguish between stable and unstable (malicious behavior or anti-goals) information. Goals are usually identified by analyzing the key operational capabilities of the system specified by the stakeholders, determined from the security policy of the organization, or analyzing the problems and deficiencies in the system-as-is, as well as from legal requirements and other functional constraints [151, 25]. For example, the high-level goal "*avoid updating firmware when vehicle is moving*" is specified by having an interaction between the security requirements engineering team and the stakeholders. Once security goals are defined, they can be formally specified in the form of a goal document. Typically, this document corresponds to functional and non-functional aspects and range from high-level security goals to low-level ones [151, 150]. To facilitate efficient collaboration and coordination of goals with other SRE activities, we have defined the security goal ontology (cf. Section 2.4.2.1) in order to store, share, reuse, and manage goal knowledge base.

3. **Identify System Assets:** The purpose of this activity is to find all the system assets and artifacts, and system behavior in the system context that has direct or indirect value to the stakeholders. The importance of system assets when coming to discovering security requirements as well as analyzing security attacks and vulnerabilities is highlighted in many security specifications standards (ISO/IEC 15408:2009 ([62], sec. 3.1.2, 6.2), ISO/IEC 27000:2012 ([64],

sec. 2.3), ISO/IEC 21827:2008 ([63], sec. 3.38), ISO/IEC 27002:2005 ([2], sec. 2.17), etc.) and model driven SRE approaches (i.e., SecureUML [70], UMLsec [88]). Department of Defense [105] suggests the following definition of system architecture: "an architecture framework provides a foundational framework with guidance and rules for modeling, documenting, developing, understanding, analyzing, using, and comparing architectures based on a common denominator across a (virtual) development organization (i.e., value net)". Therefore, the system engineering approaches puts a lot of effort to the discipline known as model-driven engineering (MDE) to allow model driven development of system architecture.

We take into account the MDE approach composed of structural models as well as behavioral models in order to determine the valuable and/or critical system assets. The most important aspects for realizing a life cycle wide SR are the interaction with involved artifacts and the evolution of the system architecture [130]. One of the basic ideas of considering MDE, in SREP, is that models help requirements analysts to understand complex software compositions and identifying potential solutions through abstraction. Such abstractions make it easier to understand how the system is evolving and how it might be secured. This is because most of the information needed to express security requirements are already defined and formalized by the design models [88]. For this activity we also defined system architecture ontology (cf. Section 2.4.2.2). In particular, the objective of such synergy is to provide the system engineers with reasoning services to reason over system models. In addition, we can connect system models with other SRE activities. For instance, we can combine goals (Step 2) and system architecture knowledge base to, first, relate goals with system assets, and then reason about security aspects such as if the goal is enough to protect the system assets, or there is a need to refine goals or define more fine-grained goals. Actually, this allows us to combine goal-oriented approaches with model driven engineering concepts.

4. **Identify Threats and Security Vulnerability:** This activity is concerned primarily with identifying attack heuristics and addresses security weaknesses (i.e., threats and security vulnerabilities) of the system that are exploitable by an adversary. Security attacks can be identified from different sources. Figure 2.7, depict relationship of this activity with other SREP activities. Following our objective for having a unified SRE model, we can consider any relationship for identifying threats and vulnerabilities. For instance, the targeted by relationship between security goal class and attack class allow us to identify attacks on goals. From the goal oriented approaches viewpoint like the KAOS [151], this relationship can be considered as an anti-goal model, where anti-goals are identified either by simply negating the goals that are specified in goal knowledge base, or linking with the adversary's malicious goals. Similarly, we can use the targeted by relationship between system architecture class and attack class to identify attack on different system models including

structural and behavior models. For example, during initial phases of system conceptualization, we can use the use cases related knowledge base to identify attacks on the system. For instance, specifying the misuse cases, like done in the problem-oriented approaches [141]. Moreover, with the evolution of system models and architecture design, we can apply more sophisticated security analysis techniques (i.e., computational attack models [139] or Dolev-yao attack model [29]) to identify threat and security vulnerability on these system assets.

Security attacks are hard to understand, often expressed with unfriendly and limited details, making it difficult for security experts and for security analysts to create intelligible security specifications. For instance, to explain "Why" (attack objective), "What" (i.e., system assets, goals, etc.), and "How" (attack method), adversary achieved his attack goals. We introduced security attack ontology (cf. Section 2.4.2.3), by taking into account security standards and security dictionaries and deriving the features, in terms of classes and subclasses that were needed in such situations. Security attack ontology has been designed to enable the specification of security attacks in a concise, readable, and extensible way.

5. **Risk Evaluation:** The purpose of this activity in the SREP is to assess whether the threats or security vulnerabilities are relevant according to the security level specified by the security goals. In our approach, we estimate the security risks based on the relevant threats, their likelihood/probability that the threats will materialize as real attacks, any potential consequences on the system assets or possible severity of an attack for the stakeholders, and the resulting impact of that adverse event on the organization. To do so, we have specified `evaluated from` relationships between risk class and other security classes (see Figure 2.7). The objective of this relationship is to extract the knowledge from different security classes and to build the risk metrics with regards to the risk model. This will, later, allow us to infer and derive the risk associated with different direct or indirect valuable system assets to the stakeholders. For instance, by estimating the "severity" of the attack and its possible outcome for the stakeholders, and the "probability" that such an attack can be successfully mounted, etc.

6. **Security Requirements Elicitation:** This step is the core activity of the SREP. Here, we aim at identifying SRs in relation to the different security classes as defined in Figure 2.7. In particular, we put much emphasis on the relationships between different security classes and their association with one another for identification of SRs. Therefore, we consider two kinds of relationships; (1) individual relationships and (2) collaborative relationships between different classes to identify SRs. An individual relationship corresponds to the situations where we can identify SRs by analyzing different knowledge specified in the security class. For instance, use cases or goals also provide

constraints and assumptions, such as performance constraints for the security functions and may themselves suggest a number of security related user requirements. For example, the information received from another entity needs to be evaluated regarding security and trust (e.g. authenticity of data). In contrast, we use collaborative relationships to identify SRs in relation to different security classes. For instance, relating goals and attacks (anti goals) to identify requirements like done in KAOS, or following the model driven SRE approaches where requirements are specified in order to protect system assets from malicious threats and vulnerabilities, etc. The novelty of our approach is that we do not restrict security engineer to identify SRs only by considering specific constructs, but provide them detailed knowledge about different security classes as well as relationships between them to identify more concrete and more realistic SRs. We will exemplify this in more detail in Chapter 6.
Of course, this activity also requires documenting and building well-structured and well-understood requirement specification [61]. The IEEE 830 standard recommended eight characteristics for specifying good software requirements specifications, out of four are related to the semantics and documentation of a requirement specification (e.g., unambiguity, correctness, consistency, and Completeness). Following these recommendations and taking advantages of expressing requirement in ontologies (cf. The role of ontologies in RE presented in Section 2.3), we have defined SR ontology in Section 2.4.2.4.

7. **Categorize and Prioritize Requirements:** The purpose of this activity is to classify the requirements in two different categories based on the risk analysis as well as on security needs. The initial set of requirements can be organized into stakeholder-defined categories (i.e., essential, non-essential, etc.) or we may use the security standards and specifications (i.e., ISO/IEC 15408:2009, ISO/IEC 18045, ISO/IEC 27000:2009, ISO/IEC 17799:2005, or ISO/IEC 21827:2008, etc.) to determine and categorize SRs into security functional components, for instance, we can map authentication related SR into "Identification and authentication – FIA" class defined in Common Criteria standard [1]. The SRs that are selected for inclusion in the design must therefore be based on an objective assessment of potential threats and their anticipated implications. As a result, SRs are categorized and prioritized in a qualitative ranking in a way that the most important requirements are handled first. Nevertheless, we acknowledge the fact that, during SR prioritization, some of the requirements may be deemed to be entirely unfeasible to implement. SRs often conflicts and interact with other system requirements or functional requirements. For instance, what is possible to do in a reasonable timeframe or budget might conflict with what is required to implement and enforce SRs. In this case, the security engineers have an option: completely dismiss the SR from further consideration, or document and label the SR for "future consideration".

2.4.2 Security Ontologies

In this section, we define the security ontologies, modeled as Ontology Web Language [27], OWL classes, for each security class described in the previous section. The core of the security ontology (see Figure 2.13) is the security sub-ontology, consisting of the concepts (1) security goals, (2) system architecture, (3) security attacks, (4) security requirements, and (5), security mechanisms, which were derived from well, established information security standards and security dictionaries. Indeed, each security ontology is defined as high-level knowledge repository for capturing, classifying and sharing security related information. With regards to our objective, which is to reuse different ontologies and taxonomies vocabularies to structure and organize the utmost security related knowledge, the high-level definition of ontological classes offer interesting perspectives. Furthermore, our security ontologies use a flexible and easily extendable structure: additional concepts can be included without effort. Thus, in this section we also complements existing taxonomies as well as ontologies that focus on building security knowledge.

2.4.2.1 Security Goal Ontology

In this section, we construct security goal ontology that captures an objective which the system-to-be should meet in the form of security goals. Regarding the security goal ontology design, we use the KAOS [151] catalog of goal patterns that generalize the most common goal configurations. The structure of security goal ontology is illustrated in Figure 4.3, where the main class called `"Goal"` has five subclasses including AchieveGoal, AvoidGoal, SoftGoal, CeaseGoal, and MaintainGoal.

Figure 2.8: Security goal ontology

- **AchieveGoal:** Achieve goals specifies a property that the system will achieve "some time in the future". More precisely, an Achieve goal describes intended behaviors where some target condition must sooner or later hold whenever some other condition holds in the current system state. For example, "*authenticity of service station must be ensured while performing firmware updates*".

- **AvoidGoal:** Avoid goals specifies a property that must not hold "at all times in the future". In our approach, we consider Avoid goals as an adversary "AchieveGoals". For example, *"install malicious firmware"* is an adversary AchieveGoal, which we want to avoid in our system.
- **SoftGoal:** "Soft goals capture preferred behaviors; they are used to compare alternative options" [21]. In particular, Soft goals are goals that do not have a clear-cut criterion when they were formulated or whose satisfaction can be subjective. They may be judged as satisfied or unsatisfied to different degrees at different stages of system development. For example, "*in-vehicle security services must be offered with high availability*".
- **CeaseGoal:** Cease goals disallow achievement "some time in the future". More precisely, cease goals state that some target condition should not hold in some (bounded) future state. For example, "*prevent service station (adversary) from closing the re-programming session, during the firmware installation phase*".
- **MaintainGoal:** Maintain goals specify a property that must hold "at all times in the future". For example, "*all in-vehicle communication must be handled in a secure manner*".

2.4.2.2 System Architecture Ontology

System architecture ontology provides a detailed structuring that can be used as a basis for describing how system assets interact and work together to achieve total system goals. Regarding the security architecture ontology design, we have adapted several classes (see marked classes in Figure 2.9.a) from more formal definitions contained in ISO/IEC 42010:2007 [3] and defined an equivalent security architecture ontology as shown in Figure 2.9.b. These classes are:

- **System** is a collection of assets organized to accomplish a specific goal or set of goals. Following this definition, a system can be further classified into system assets subclass.
  - System Assets "are in the form of information that is stored, processed and transmitted by or systems to meet requirements laid down by owners of the information" [62]. Typically, the system assets can be classified into data, software, hardware, etc. Nevertheless, the notion of system assets in different domains is quite diverse. In order to give a complete overview of the system, we can integrate system asset's taxonomies (i.e., WAND Automotive Taxonomy [159], etc.) that describe different classes as well as subclasses of assets involved in building system architecture.
- **Architecture** of a system is the system's conceptualization, articulated in its system assets, their relationships to each other and to the environment. In

particular, it defines the structure, behavior, and more views of a system in the form of conceptual models. Subsequently, each architecture concept can be further categorized into several subclasses of systems architectures such as functional architecture, logical architecture, technical architecture, etc.

- **View** represents a system from the perspective of a set of architecture related concerns that are meaningful to one or more stakeholders in the system. Views can be seen as the content of a viewpoint, i.e., a description or modeling perspective used in defining the system architecture.
- **Model** represents the particular design of system architecture. We take into account the MDE approach composed of conceptual models as well as logical models in order to relate the valuable and/or critical system assets.
  - Behavioral model: In the behavioral model the internal structure of a system is neglected, and only its interaction over its system boundary to its context is considered.
  - Structural model: In the structural model the internal structure of a system is described in terms of selecting, connecting and characterizing generic components, describing the way the system is connected to its context and interacts, and decomposition into components or subsystems.

Based on these two models, one may now start to reason about what is going on within and beyond the system boundaries, i.e. what the effects of the systems with respect to its environment are, and on the other side how the system is internally structured, i.e. how the system behavior is reflected in terms of the internal system behavior.

(a) IEEE system architecture metamodel

(b) Security architecture ontologys

Figure 2.9: IEEE system architecture metamodel [3]and its equivalent system architecture ontology

2.4.2.3 Security Attack Ontology

The definition of a security attack ontology aims at building knowledge vocabulary for security attacks that could be described including their type, mode, consequences, and such details. Figure 2.10 sums up our analysis with respect to extracting different constructs and concepts defined in well-known security standards (i.e., ISO/IEC 15408:2009, ISO/IEC 18045, ISO/IEC 27000: 2012, ISO/IEC 17799:2005, NIST SP-800:30, etc.) and security dictionaries (i.e., CVE, CAPEC, OWASP, CLASP, etc.) in order to build the security attack ontology.

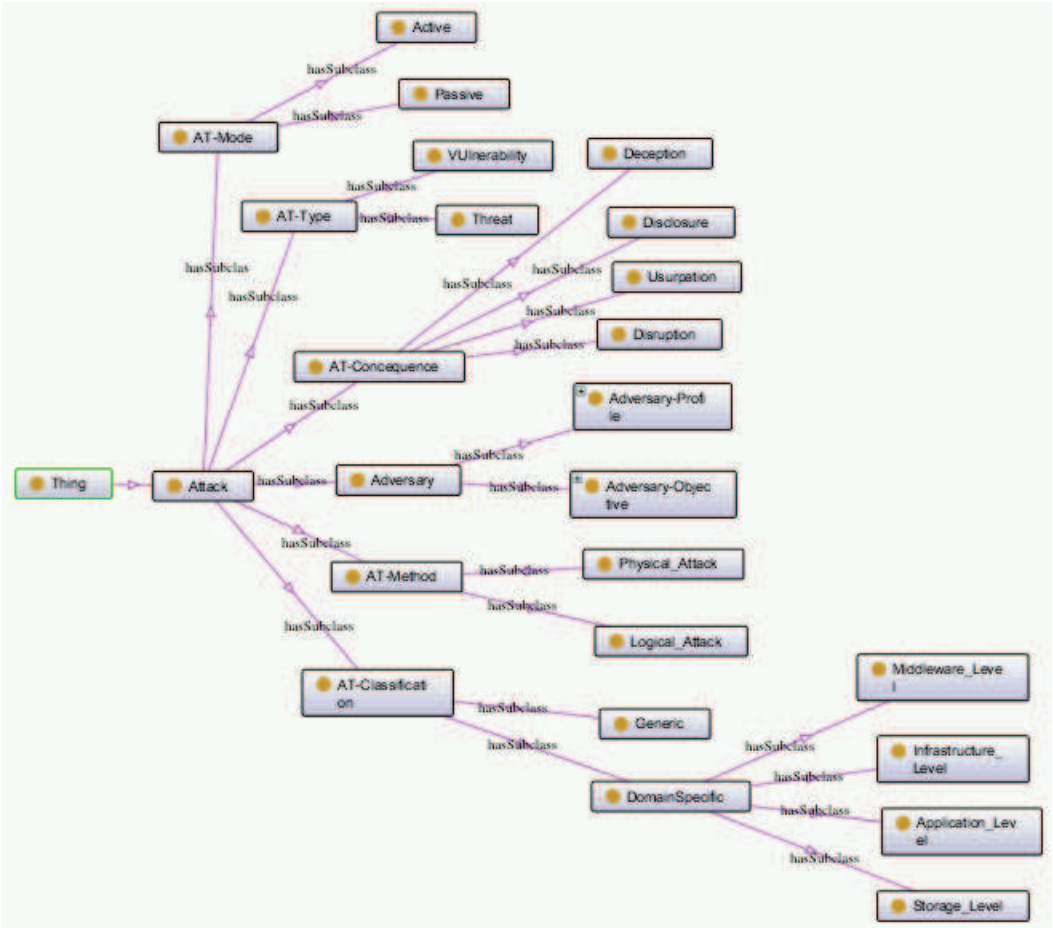


Figure 2.10: Security attack ontology

- **Attack Type** depicts an attempt to destroy, expose, alter, disable, steal or gain unauthorized access to make unauthorized use of system assets [64]. This abstract level definition of attack type allows us to further categorize attacks into the *Threats* and *Vulnerabilities* sub classes (see Figure 2.10). Information security standards describe these two main dimensions of attack types in many specifications. For example, threat terminology is described in the ISO/IEC

15408:2009 ([62], sec. A.6.2) standard, ISO/IEC 27002:2005 ([2], sec. 2.16) standard , and NIST SP 800-30 ([99], sec. 3.2) standard. In the scope of this thesis, we use the threat terminology as defined in section 2.45 of ISO/IEC 27000:2012 [64].

- Threat: A threat is a "potential cause of an unwanted incident, which may result in harm to a system or organization". In a certain sense, here, we can use the attack patterns approach [18, 24] to categorize different kind of threats.

  Instead, the vulnerability terminology is described in ISO/IEC 15408: 2009 ([62], sec. 3.5.7), ISO/IEC 27000:2009 ([64], sec. 2.46), ISO/IEC 21827:2008 ([63], sec. 3.38), ISO/IEC 27002:2005 ([2], sec. 2.17) and in other security dictionaries OWASP [108], CVE [24], we use the one described in CVE:

- Vulnerabilities: "An information security vulnerability is a mistake in software that can be directly used by a hacker to gain access to a system or network" [24]. Subsequently, we can use the CVE list to classify different security vulnerabilities and their consequences on the system assets.

- **Attack Consequences** refers to an impact of security breach or outcomes that are not the ones intended by a purposeful system action. The attack consequences can be classified as:

  - Usurpation is a derogatory term used to describe either a misappropriation or misuse of the system functionalities.

  - Disruption is an event, which causes an incapacitation, corruption, obstruction, and unplanned deviation from the expected system behavior, according to the functional and non-functional objectives.

  - Deception is defined as masquerade, falsification, and repudiation actions taken by an adversary, to thereby causes a system to accept as true a specific incorrect version of reality.

  - Disclosure enables an adversary to gain valuable information about a system and its functionalities either by exposure, interception, inference, intrusion, etc. that tries to uncover the details of a system.

- **Adversary:** An adversary is a threat agent according to the following ISO/IEC 15408: 2009 ([62] sec. 3.1.71) and the ISO/IEC 21827:2008 ([63], sec. 3.35) standards, who attempts to attack system assets that have value to the stakeholders. An adversary may range from a very unskilled individual to an expert or even to multiple dedicated groups. In order to anticipate and thwart the expected types of attacks, one must have a solid understanding of the adversary's perspective and his/her capabilities and know-how about attack potential. We

consider different attack objectives and corresponding adversary profiles and describe them as:

- Attack Objective: This class suggests particular types of adversary and his capabilities, as well as associated attack motivation. At the abstract level of specification attack motivations can be broadly categorized as:
  - Individual Benefits: Personal advantages can be gained in different ways and for different purposes. For instance, gain reputation as hacker, financial gain fraudulent commercial transactions, etc.
  - Economical Benefits: These motivations and underlying objectives should be envisaged at an organizational scale.
  - Political Benefits: The main goal of the attacker is to destroy the reputation of an organization or an individual system asset. For example, acquiring system design information or for the purposes of fraud, industrial/state espionage or sabotage.
  - Criminal Benefits: An augmentation of the attack motivation to harm an individual for the purposes of criminal or terrorist activity, destroy or financial harm, destructive attacks or intellectual property attacks, etc.
- Adversary Profile: The adversary profile depicts the attack potential that is a measure of the minimum effort to be expended in an attack to be successful. In ISO/IEC 15408:2009 ([62], sec. 3.1.5) the attack potential is defined as a "measure of the effort to be expended in attacking a TOE, expressed in terms of an adversary's expertise, resources and motivation". Essentially, the attack potential for an attack corresponds to the effort required creating and carrying out the attack. The higher the adversary's motivation is the higher efforts they may be willing to exert. After having performed a comparative analysis of several security specifications and standards, we suggest the following abstract level taxonomy (see Figure 2.11) to be considered during an analysis of the attack potential:
  - Elapsed Time: This is the total amount of time taken by an adversary to identify that a particular potential vulnerability may exist, to develop an attack method and to sustain the effort required for mounting the attack.
  - Expertise: This refers to the required level of general knowledge of the underlying principles for mounting an attack (i.e., system architecture, security components, etc.), product types or attack methods.
  - Location: This refers to the knowledge and the capabilities, which an attacker may have, depending of his/her location; this is typically reflected by the terminology for an Insider or Outsider attacker.

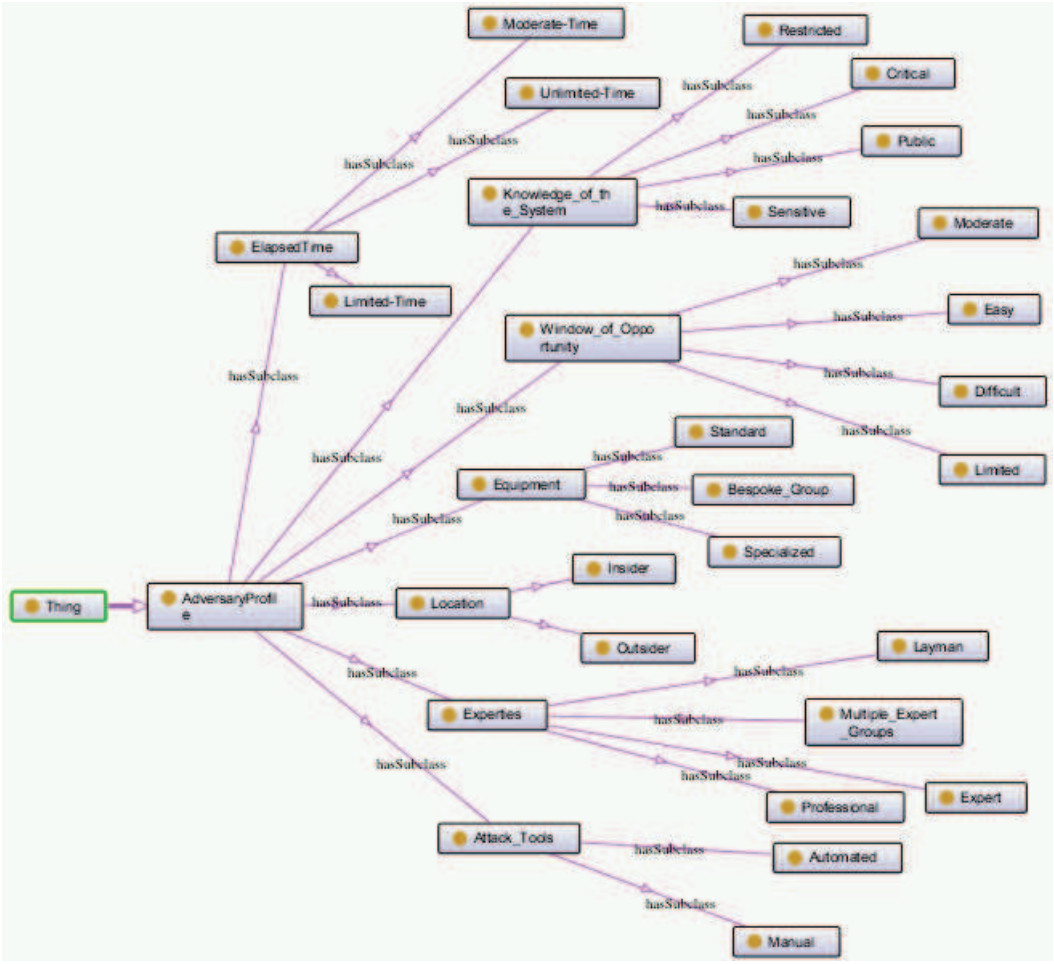


Figure 2.11: Adversary taxonomy

For instance, insider attack agents are likely to have specific attack objectives, potential, and have legitimate knowledge and access to the system.

* Window of opportunity: This concept has a relationship with the elapsed time factor. Identification and exploitation of vulnerability may require considerable amounts of accesses to a system that may increase the likelihood of detection of the attack. In contract, some attack methods may require considerable effort off-line, and only brief access to the target to exploit.

* IT hardware/software or other equipment: This refers to the equipment required to identify and exploit vulnerability.

* Knowledge of the system under investigation: This refers to the specific expertise required in relation to the system under investigation.

Though it is related to general expertise, it is distinct from it.

- **Attack Mode:** The attack mode refers to the actions ([62], sec. 3.1.1, A.6.2) that an adversary takes during the execution of an attack and that can be labeled as active or passive attacks:
  - Attacks modifying the behavior of the system (active attacks).
  - Attacks aiming at information retrieval without modifying the behavior (Passive attacks).
- **Attack Method:** This class is related to the attack mode class. The attack method can be classified into either functional (logical) attacks or physical attack methods:
  - Attacks physically modifying the behavior of the system (physical attacks) and
  - From the functional point of view, attacks aiming at logical manipulation of information without physically modifying the system behavior.
- **Attack Classification:** The attack classification class is define to categorize and systematically aggregated into a set of well-defined classes that provide a comprehensive description of attacks and its objectives. A collection of ways, including security dictionaries (i.e., CVE, CAPEC, OWASP, CLAP) can be used to determine and cluster security attacks and vulnerabilities. However, at the abstract level, we can classify security attacks into:
  - Generic attack descriptions that represent a general class of security attacks and vulnerabilities, which can be reused and adapted to any application specific instantiation such as a *birthday attack*, a *preimage attack*, a *collision attack*, etc.
  - Domain Specific attacks and vulnerabilities depict the particular attack objectives of an adversary and target specific assets (i.e., application, middleware, infrastructure, and storage), and parameters of the system.

2.4.2.4 Security Requirements Ontology

In this section, we construct security requirement ontology with respect to the different constructs and concepts defined in well-known security requirements specifications and security standards (i.e., ISO/IEC 15408:2009, ISO/IEC 18045, ISO/IEC 27000:2009, etc.). The SRO, which is independent of the existing conceptual SRE foundations, aims to detect the missing security construct in SR frameworks and facilitates their enhancement. The core classes and the concepts identified for security requirements ontology, summarized in Fig. 2.12, are:

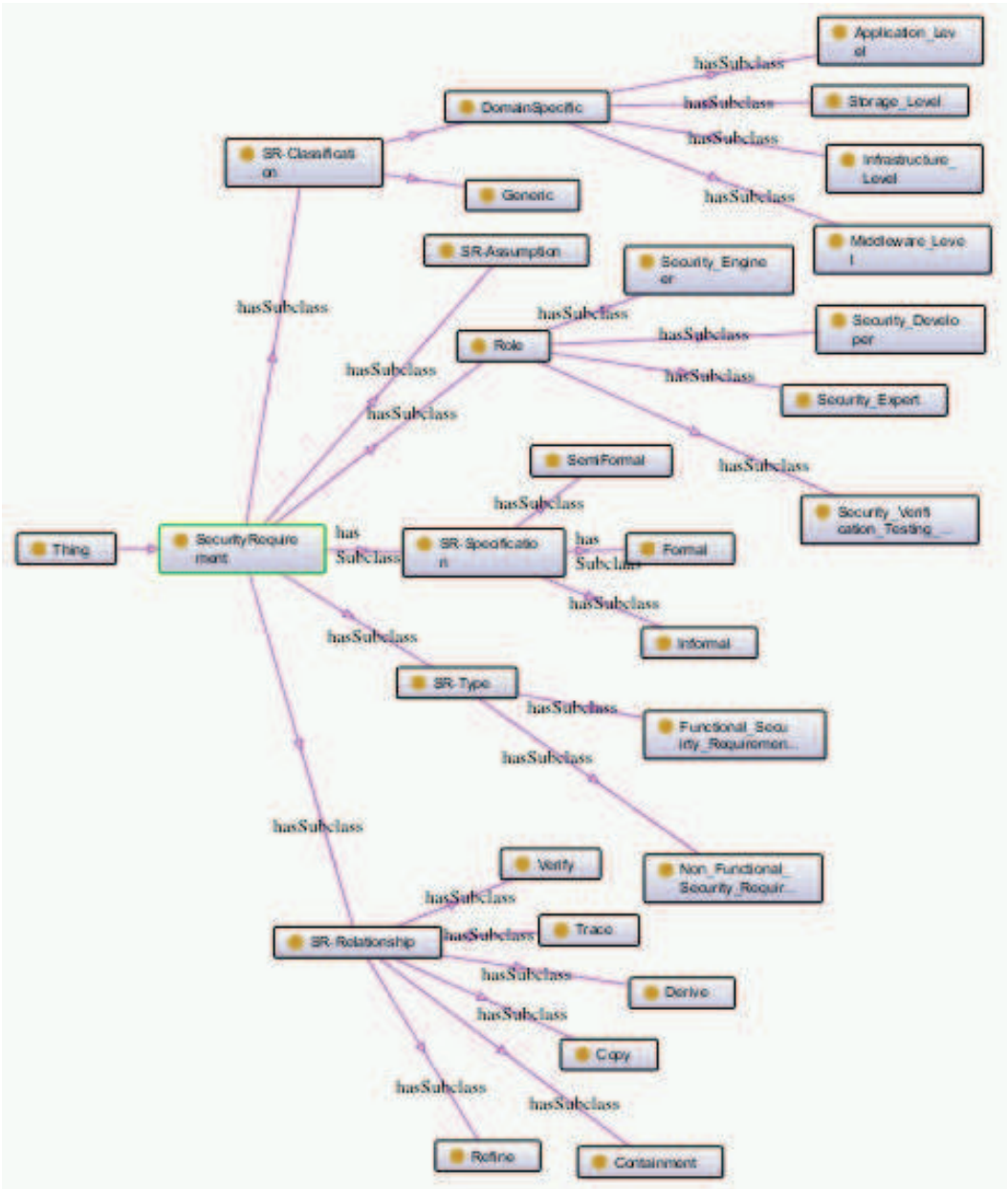


Figure 2.12: Security requirement ontology

- **SR Type:** This class is described to specify and distinguish more concretely what kind of security requirements should be defined and implemented in order to assure the security of a system and its data. These types are:

  - *Functional Security Requirements (FSR)* are security services or security capabilities that a systems or components must be able to perform. Put simply, it describes a positive functional behavior related to specific security feature. Examples for FSR are authentication, authorization, integrity, and so on. Here, we can use the classification specified in ISO/IEC 15408:2009 (part 2 [1]) in order to classify and map FSR into well-structured security functional components (SF components).

  - *Non-Functional Security Requirements (NFSR)* are typically security requirements essentially stemming from attack mitigation. Typical examples for NFSR could be "password or key strength", "system logs", or requirements derived from best practice standards.

- **SR Classification:** This class is defined to classify and systematically aggregate requirements into a set of well-defined classes and functions of security requirements that provide a comprehensive description of requirements and its objectives. We can use collection of ways, including stakeholder-defined categories (i.e., essential, non-essential, architectural constraints, etc.) , or security standards (i.e., ISO/IEC 15408:2009, ISO/IEC 18045, ISO/IEC 27000:2009, etc.) to determine and cluster security requirements into security functional components (SFC). However, at the abstract level, we can classify security requirements into:

  - *Generic:* requirement description that represents a general class of security requirements, which can be reused and adapted to an application specific instantiation.

  - *Domain Specific:* requirements or a set of SRs that are specific to the system and that provide for protection of essential services and assets of the targeted application.

- **SR Specification:** This class presents how numerous approaches can be used for building and modeling security requirements specifications. Categories of requirement modeling include:

  - *Informal:* The formal representation of SRs, correspond to techniques where natural languages (i.e., text based approaches) are used to present security requirements.

  - *Semi-Formal:* The semi-formal representation of requirements, correspond to techniques where diagram and tabular techniques are used to present security requirements in structured form.

  - *Formal:* A formal specification of SRs includes the mathematical logics (i.e., set theory, proof theory, first-order logic, etc.), and model transformation techniques to present requirements in formal logic, in which the syntax, semantics and manipulation rules for the requirements are explicitly defined.

- **Assumptions related to SR:** Assumptions are just one part of the, usually hidden, reason for a design decision and are frequently made during requirement engineering process. From the SRE point of view, assumptions are required to provide an extensive and rich description about "why" and "what" tradeoffs were made during requirements identification or refinement, and "how" to realize certain security requirements.

- **SR Relationships:** The purpose of this class is enable relationships that allow security engineers to relate SRs to other requirements as well as to other model or a set of model elements. We, in particular, consider relationships defined in the SysML specification [107]. The structure of this class is illustrated in Figure 2.12, where the "relationship" class has six subclasses including refine, derive, copy, containment, verify and trace. A detailed description of each subclass is present in Section 3.3.3.1.

- **Role:** The role relates to individuals and/or teams (i.e., security engineers, test engineers, verification team, etc.), involved in the SREP. We take into account the role to determine by whom and at what level of system abstraction SRs are specified.

2.5 Conclusions

In this chapter, we presented the main building blocks of our knowledge-centric SRE methodology. More precisely, we integrated within a single model the notions of the goals driven SRE, model driven SRE, security attacks, and risk assessment, with the view to propose a kind of unified methodology which offers means to overcome the limitations of these state of the art approaches. The introduction of this knowledge-centric SRE methodology meets the three following objectives. Firstly, we are able to build the well-structured and well-formed specification of security concepts (i.e., goals, requirements, attacks, etc.) through the definition of security ontologies. Secondly, it offers the possibility to obtain more concrete requirements as we start sharing knowledge base between different SREP activities. Thirdly, it increases the compatibility between concepts of several SRE approaches that share common concepts. However, we still have to explain which kind of modeling language as well as tools can be used to support our methodology. These issues are addressed in chapter 3.

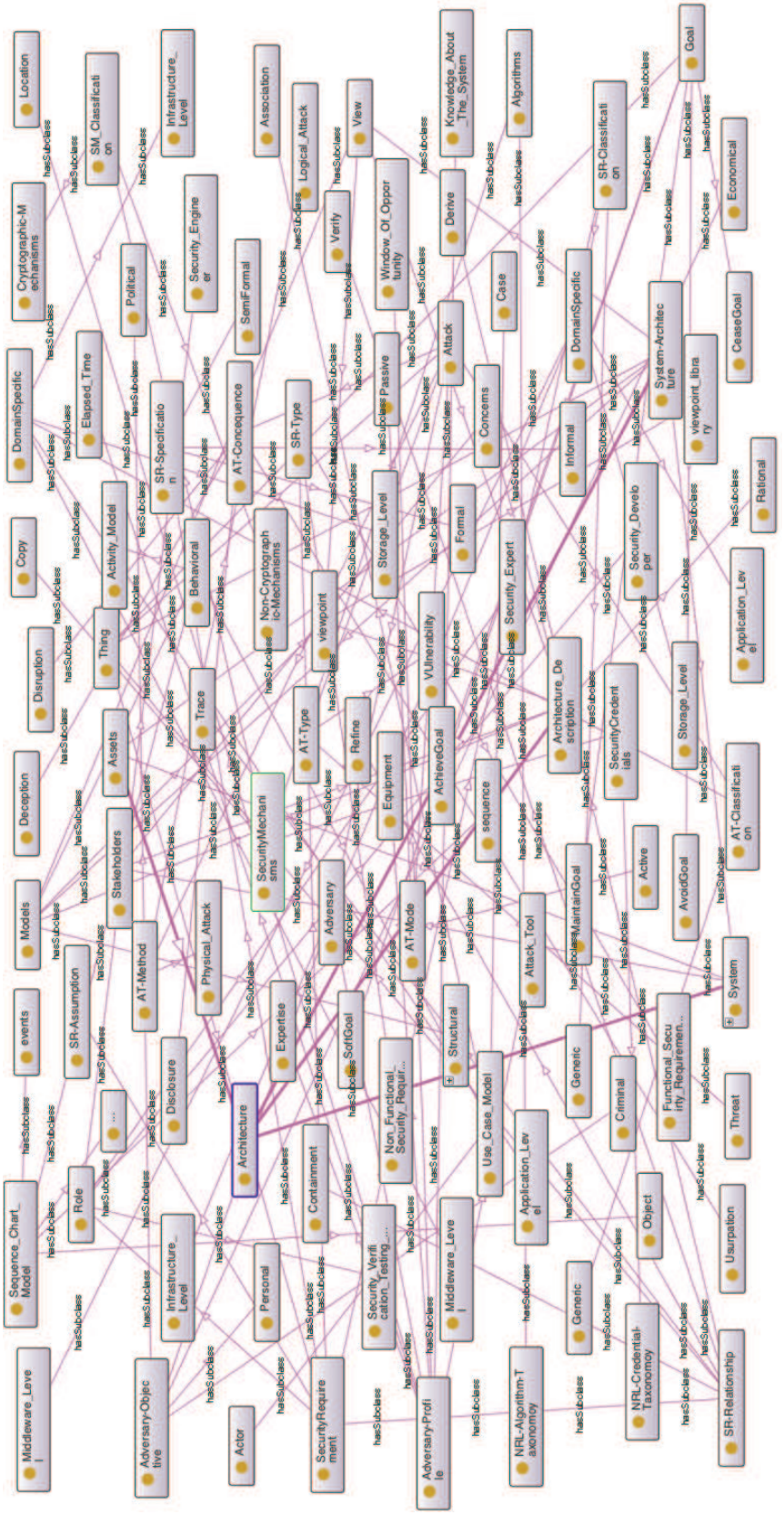


Figure 2.13: Security ontology

Chapter 3

System Modeling Language for Security - SysMLSec

3.1 Introduction

We described the main building blocks of the security requirement engineering methodology in the previous chapter. To pave the way for system-wide SREP, we first have to breath live into a collection of conceptual stages following the mapping of stakeholders needs into product functions and use cases. Also, preceding the design of these functions across the engineering disciplines (i.e., hardware, software, etc.). In this context, a number of modeling languages (i.e., UML, SysML, etc.) have been proposed to help engineers from different system development stages to communicate, share, and compare their perspectives, to reason about properties of a system. These modeling languages for software engineering practices express different concepts to serve different development purposes like use case modeling, requirement modeling, protocol modeling, etc. However, security issues involve special concerns that these traditional software engineering languages do not consider. Consider, for example, a general behavior modeling notation that expresses interactions of entities in the system without considering the harmful behavior of an adversary. Thus, the models do not convey the impacts of the malicious behavior of the adversary on requirements, design, and architecture to the next phases of system development lifecycle. As we have reviewed in Section 2.2, to model specific security aspects such as threats, vulnerabilities, assets, and security requirements several security modeling languages have been developed. A number of extensions of UML (i.e., UMLsec [69], SecureUML [88], Misuse cases [141], Abuse cases [86], etc.), allow to express security relevant information within the diagrams in a system specification have been proposed. Yet, to best of our knowledge, none of them provides the expressivity required to deal effectively with system-wide SRE. Another major group of contributions to the conceptual modeling of security requirements like KAOS [151] and Secure Tropos [93], etc., have defined their own graphical formalism each of which allows to express security relevant information (i.e., goal, anti-goals, requirements, obstacles, etc.).

Table 3.1, summarizes existing modeling notations based on the concepts they express and usage of the models. As can be seen that, each SRE approach is able to specify certain features of requirement modeling and lacks conceptual modeling as-

pects to express some other. In addition, these approaches do not provide constructs to link and map these requirements with other system development activities, and leave us with two different models that are either difficult to combine or that can not be cross-referenced. Because engineers express the SRs based on the functionalities they want to protect, they need to see the security requirements in the design. We claim it's necessary to combine objects and functions into the security requirement engineering process. This is possible with SysML, a general-purpose graphical modeling language. Section 3.2 is specifically dedicated to the evaluation and analysis of SysML for handling system wide requirement engineering process. The remainder of this chapter is organized as follows. Section 3.3 explains how to go from standard system modeling language for engineering to the notion of security-oriented system modeling language. Section 3.4 presents an approach for annotating SysML specification with security reasoning.

SRE Approaches	Core Modeling Construct	Structural Modeling	Behavioral Modeling			Attack Modeling	Security Requirement Modeling
		System Modeling	Use Case Diagram	Activity Diagram	Sequence Diagram		
KAOS	Goal (anti-goal, obstacles, agent)	□	□	□	□	■	■
Secure Tropos	TROPOS (Task, Actor, Resource, Goal, Soft Goal, Dependency)	■	□	□	□	■	■
UMLsec	UML	■	■	■	■	□	□
SecureUML	UML and RBAC	■	□	□	□	□	■
Misuse Cases	UML	□	■	□	□	■	□

Table 3.1: Summary and comparison of security modeling approaches. "■" for available properties, "□" indicates that the modeling notation does not consider the concept in its conceptual modeling.

3.2 System Modeling Language – SysML

SysML, The Systems Modeling Language, specification is defined and promoted by the Object Management Group (OMG). The goal of the OMG is to provide a "standard modeling language, SysML, for systems engineering to analyze, specify, design, and verify complex systems quality, improve the ability to exchange systems engineering information amongst tools, and help bridge the semantic gap between systems, software, and other engineering disciplines" [107]. SysML aims at unifying the various types of modeling languages currently used in system engineering in

a similar manner to how UML combined diverse modeling languages used in software engineering. SysML allows system engineers to model and follow system wide development concepts including system requirements, system behavior and system structure. SysML is often interpreted as being a new system engineering modeling language, but there have been several languages that have been used in the past such as block and internal block diagrams of UML are used to model the parametric constraints. In fact, SysML is based on UML and cannot be considered an entirely new language – more a large set of useful additions (extensions/modifications) to the existing core UML modeling concepts and diagrams. However, SysML, notably the object diagram, the deployment diagram, the component diagram, the communication diagram, the timing diagram, and the interaction overview diagram, does not require some diagrams of UML. In contrast, SysML includes some new diagrams and constructs not found in UML: the parametric diagram, the requirement diagram and flow ports, and the flow specifications and item flows. In addition, SysML also includes an allocation relationship to represent various types of allocation, including allocation of functions to components, logical to physical components, and software to hardware [107]. SysML classifies these different diagrams into four categories: behavior, structure, system requirements, and parametric relationships. These are known as the four pillars of SysML as illustrated in Figure 3.1.

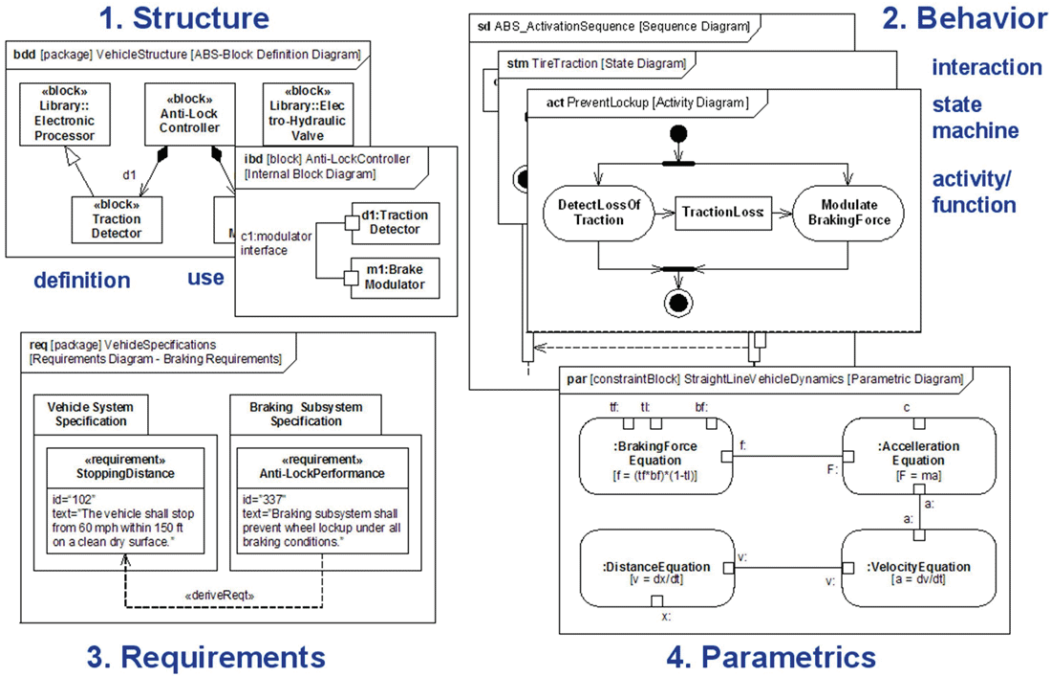


Figure 3.1: The four pillars of SysML [107]

These four pillars represent a system engineering process, with a system model being made up of one or more modeling diagrams (i.e., system behavior, system re-

quirements, system structure, etc.), each of which interacts with one or more other to support model-based systems engineering (MBSE). For instance, the system structure can be represented by block definition diagrams and internal block diagrams, the behavior of the system can be depicted by the use case diagram, the activity diagram, the sequence diagram, etc. The latter modeling constructs are imperative to support requirements engineering and performance analysis. The focus of the new constructs in SysML is geared towards some fundamental system engineering concepts, such as requirements engineering and system behavior [50]. Let us discuss these two new modeling constructs (i.e., Requirement Diagram and Parametric Diagram) in more detail, which are the core activities in requirement engineering process.

Requirement Diagram: Although requirements have traditionally been realized using use case diagrams, which consider requirements from a behavioral point of view, the introduction of the requirements diagram in SysML allows the structural relationships between requirements to be modeled [50]. The requirement diagram is used to integrate the system models with text based requirements that are typically captured in requirements management tools. Figure 3.4 shows the metamodel for requirement diagrams. From the model it can be seen that a "Requirement diagram" is made up of one or more "Requirement" and zero or more "Relationships" elements. The dark background in the objects represents the core elements of the SysML metamodel. While others elements represent our contribution to the metamodel, which we will explain in Section 3.3.3. Here, a requirement element is used to represent text-based system requirements that can be related to other requirement or system models/elements via the relationship element. For instance, a requirement can be expressed and assigned to a system model or set of model elements that is intended to realize or satisfy the requirement. Thus, it provides a bridge between typical requirements management tools and the system models. As the SysML specification states: "The requirements model describes the SysML support for describing textual requirements and relating them to the specification, analysis models, design models, etc. A requirement represents the behavior, structure, and/or properties that a system, component, or other model element must satisfy" [107]. With respect to the requirement representation, SysML provides the graphical, tabular, or tree structure format for modeling requirements. The SysML requirement graphical notation that allows security engineers:

- To express and attribute individual requirement in a precise way,
- Make each requirement understandable (the interpretation of each requirement is clear),
- Relate requirements with other system development activities throughout the system's life cycle.

- Group and to decompose requirements according to a hierarchical set of concepts, and
- Trace the source of requirements and reasoning for requirements existence.

These various properties allow the modeller to document the requirement in well structured and in a consistent way.

Parametric diagram: The Parametric diagram is the second new type of diagram introduced to describe constraints on system properties to support engineering analysis. The parametric diagram is a specialized variant of an internal block diagram that restricts diagram elements to represent constraint blocks, their parameters and the properties of block that they bind to. Parametric diagrams are made up of one or more constraint blocks, zero or more part, and one or more connectors [107]. This is illustrated in Figure 3.5. The constraint block is used to show which constraints are being used. The SysML specification describes constraint blocks in terms of conditions that are represented by mathematical equations. More precisely, the constraints block contains an equation, expression or rule that relates together the parameters given in the parameters block. The concepts behind constraints can be extended to cover general rules that constrain system properties and behavior such as authentication should be performed BEFORE authorizing entity to access system resources, etc. The use of a constraint block is called a constraint property and is depicted on a parametric diagram. The interconnection between constraint blocks and part or constraint blocks is shown on a parametric diagram using zero or more binding connectors as shown in Figure 3.5. Binding connectors depict an equality relationship between the two connected parameters or between a parameter and a value property. In the parametric diagrams, a standard part element includes properties to specify its unique identifier and text description.

We can draw some conclusions from the aforementioned properties of the SysML. SysML provides a support to establish a good understanding of the processes that exist and that are needed to realize the requirements of a system throughout the system engineering process. In addition, SysML provides a well-integrated and powerful notation, requirement diagrams, to specify expressive requirements. This provides us a guarantee to obtain complete, consistent, organized, and verifiable requirements, which are traceable at any stage of system conceptualization. It also provides means to capture the rationale for a specific requirement or relationship. Moreover, SysML provides an excellent set of extension mechanisms that can be used to augment and enhance the capabilities of the modeling language. SysML thus seems to be the right tool to map and relate security requirements to other system development activities. However, a major obstacle towards the usage of SysML is the lack of support from the security viewpoint. In particular, there is no explicit modeling construct to express security weaknesses such as security attacks

and vulnerabilities in the system that may lead to modeling attacks, nor for modeling security requirement.

3.3 SysML for Modeling Security Aspects

The purpose of this section is to develop syntactic, semantic, and methodological extensions to SysML, that are needed to support the modeling of security aspects and their relationships. We use different SysML diagrams as well as extended requirement and parametric diagrams to model security aspects like security requirements and attack tree modeling. We group these set of diagrams (diagrams used for modeling security aspects) in a profile, **SysMLsec**, The System Modeling Language for Security. Figure 3.2, presents the SysMLsec diagram taxonomy. Among the many tools (i.e., Enterprise Architect [144], Papyrus [110], Rational Rhapsody Developer [56], OMEGA2 [102], ARTISAN [126], etc.) that support SysML, we have extended an in-house SysML tool called TTool [82] to supports our SysMLsec profile. TTool is an open-source toolkit that supports several UML profiles, including DIPLODOCUS [8], and AVATAR [114], TURTLE [6], and Network Calculus [7]. TTool targets the modeling of embedded systems with time and security constraints and offers additional capabilities such as simulation, formal verification, and code generation.

SysMLsec is mainly composed of four methodological steps and use different SysML diagrams:

3.3.1 Structural Modeling

Structure diagrams in a SysML specification are a type of diagram (block definition diagrams and internal block diagrams that represent) that are used to represent the internal structure of a system, and the collaborations that this structure makes possible [107]. Following our model driven security requirement engineering approach, in which models are the main artifact during system development, structure diagrams play a central role in modeling and expressing these system models. They also provide a common root for user defined or domain-specific hierarchies of system component types. We in particular use block and internal block diagrams in order to represent hardware, software, facilities, or any other system element. More precisely, we use:

- A block definition diagram to describe the system hierarchy and system/component classifications (Chapter 4).
- The internal block diagrams are used to describe the internal structure of a system in terms of its parts, ports, and connectors (Chapter 4).

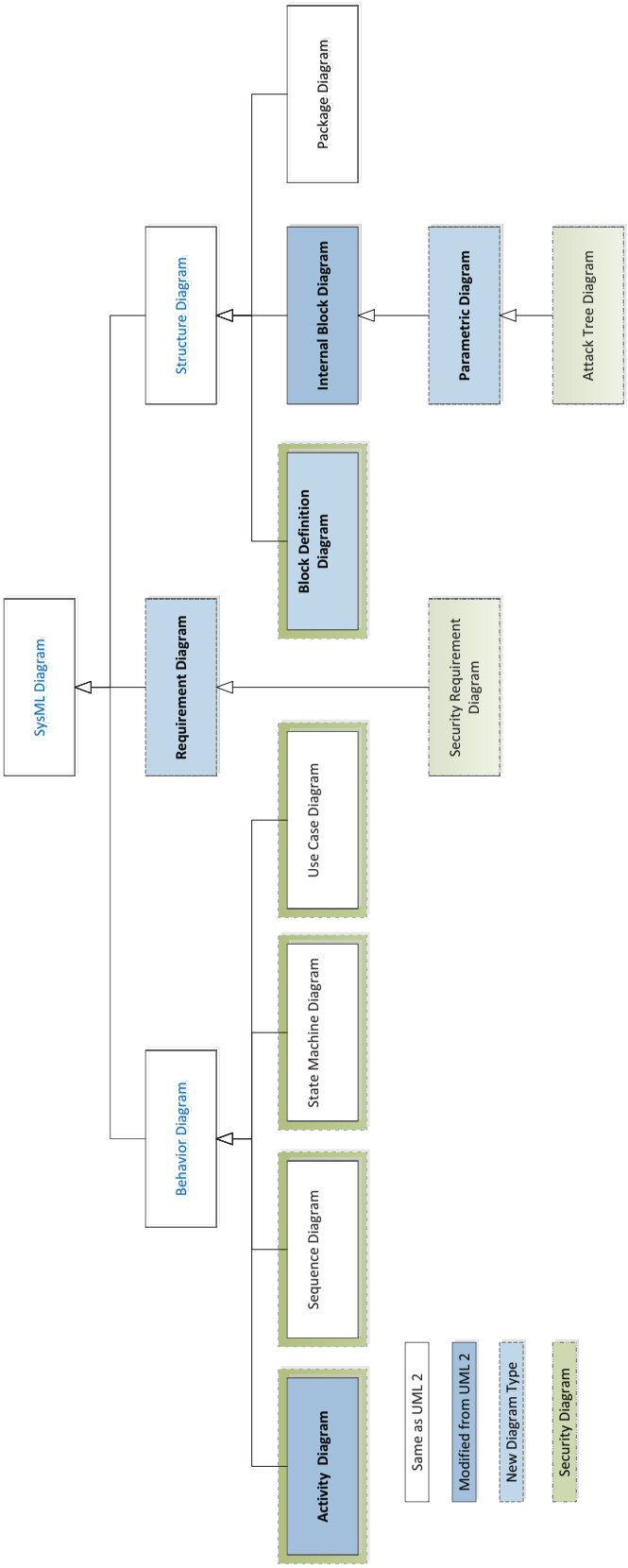


Figure 3.2: Extended SysML diagram taxonomy – SysMLsec

3.3.2 Behavior Modeling

The behavior diagrams include the use case diagram, activity diagram, sequence diagram, and state machine diagram.

- The use case diagram provides a high-level description of functionality that is achieved through interaction among systems or system parts" [107]. We rely on use-case diagrams (Chapter 4) to model anything that performs a function. It is thus a useful tool for the security analysis phase of security requirement engineering.
- We use activity diagrams to explain use cases in more detail. More precisely, we use activity diagrams when the use case scenario contains considerable control logic, inputs and outputs, and/or algorithms that transform data (Chapter 4);
- In our SysMLsec approach, we use the sequence diagram to represent the interaction between collaborating parts of a system, the objects and classes involved in the scenario, and the sequence of messages exchanged. We typically use message sequence charts to model the security protocols (Chapter 7).
- The state machine diagram describes the state transitions and actions that a system or its parts perform in response to events [107]. We can use the state transition diagrams to express and analyze an abstract description of the behavior of a system during security analysis as well as during formal verification and testing activities.

3.3.3 Security Requirement Modeling

In order to take advantage of SysML requirement modeling capabilities (e.g., graphical modeling of requirements, relationships between requirements, traceability, etc.), we have extended the SysML requirement diagram to model SRs. A meta-model, which is an extension of the SysML meta-model for requirement, showing the organization of the security requirements is depicted below in Figure 3.4. The gray background in the objects represents our extensions to the meta-model for modeling security requirements, whereas the objects in black background presents the core element of SysML requirement diagram. A common way to define new modeling constructs is by extending existing SysML constructs with new properties and constraints. In this context, the SysML specification [107] suggests to use the stereotype mechanism to define new diagram notations. More precisely, a stereotype extends the SysML metamodel, allowing one to create new kinds of diagrams/classes derived from existing ones, but specific to a class of problems. We used the stereotype mechanism to extend the requirement class (see Figure 3.4) in order to create the «SecurityRequirement» diagram which allows us to model and distinguish SRs from

other system requirements. After creating a stereotype, specific properties and constraints can be created. Properties add information to elements of the model, and are normally associated to tagged values. Properties are displayed inside braces, with the tag and the value encoded as strings. Tagged values add extra semantics to a model element. We have created several properties (i.e., SR Type, SR Kind, Risk, a reference to Attack Tree Node, etc.) in order to enrich SR with additional but compulsory details. This is illustrated in Figure 3.4. In particular, we mapped concepts from our SR ontology (cf. Section 2.4.2.4) to the tagged values to see the concepts and properties in our SysML SR diagram that we will explain in more detail in Section 3.4. Such concepts and properties are useful in particular with respect to the construction of fine-grained and detailed SR specification.

3.3.3.1 Relationship

A relationship is a connection that enables the security engineer to relate SRs to other requirements as well as to other UML/SysML model elements. In the SR diagrams, we use several SysML relationships to define the structure between model elements. Examples of relationships include the containment, refine, verify, derive, satisfy, copy, and trace relationships. Based on the modeling construct of these relationships, we group them into two categories:

- **Related To.** Relationship types are used to show how one SR is related to another requirement.
  - *«Containment»* enables a complex SR to be break down into its child requirements [107] – that is an aggregation association which contains sub requirements in terms of a requirements hierarchy. The decomposition relationship (see Figure 6.1) is shown with a «containment» relationship.
  - *«Copy»* relates SRs that are reused across model elements (versions/-variants) [107]. A «copy» relationship between SRs establishes a master/slave relationship between them in such a way that the text of the slave SR is a read-only copy of the text of the master requirement: the slave requirement cannot be changed but, if the master requirement changes, then so will be the slave. For each slave a unique master should be related but several slaves are possible for each master requirement. In particular, the objective of this relationship is to reuse generic SRs in multiple contexts. For instance, enforce access control is a generic SR which we can reuse for multiple applications by using the *«Copy»* relationship.
  - The *«DeriveReq»* relationship relates a derived SR to its source requirement [107]. This typically involves some analysis to determine the multiple derived SR that support a parent SR. The derived requirements generally (see Figure 6.12) correspond to the refinement of SRs at the

next level of details of the system analysis.

- **Connected To.** some relationships can be used to connect any type of model element to a SR and are used in the following ways:

  – The *«Refine»* relationship can be used to describe how a model element or set of elements can be used to further refine a security requirement [107]. For example, details about the architecture of an embedded system and the attacks it may be subjected to may be used to refine the analysis about a SR. The refine relationship in the SysML specification is meant to be used for that exact purpose with the difference that we use the refine relationship to relate security requirements to different security classes (i.e., system assets, attack trees, etc.), whereas in the SysML specification, the refine relationship used to link requirements (i.e. system requirements) to use cases [107]. Figure 3.3 shows our extended definition of the refine relationship, representing the SR object and the referenced class objects. Here, we focus on the reference class (see Figure 3.4) that

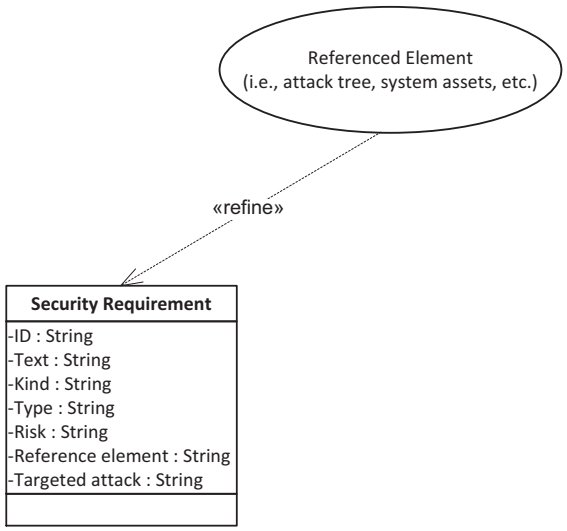


Figure 3.3: Extended definition of the «refine» relationship

  enables us to link and map SR to the models/elements. More precisely, the object of reference class is to link and trace the source of SR, reason for requirements existence, as well as understanding as to why and how the system meets the current set of SR.

  – The *«Satisfy»* relationship is used to show that a model element (or will) satisfies a security requirement. A «satisfy» relationship will appear in SysML SR diagram that relate elements of a design or implementation model to the security requirements that those elements are intended to satisfy [107].

– The «*Trace*» relationship is used to show that a model element can be traced to a security requirement. The trace link is a general-purpose relationship free-form link connecting a requirement element to any requirement/model element [107]. We use «trace» relationship specifically to link security requirements to their related source.

These various types of relationship allow us to relate explicitly different parts of a model to the SRs as a way of ensuring the consistency of the SRs and other system models. We will exemplify these various relationships in Chapter 6.

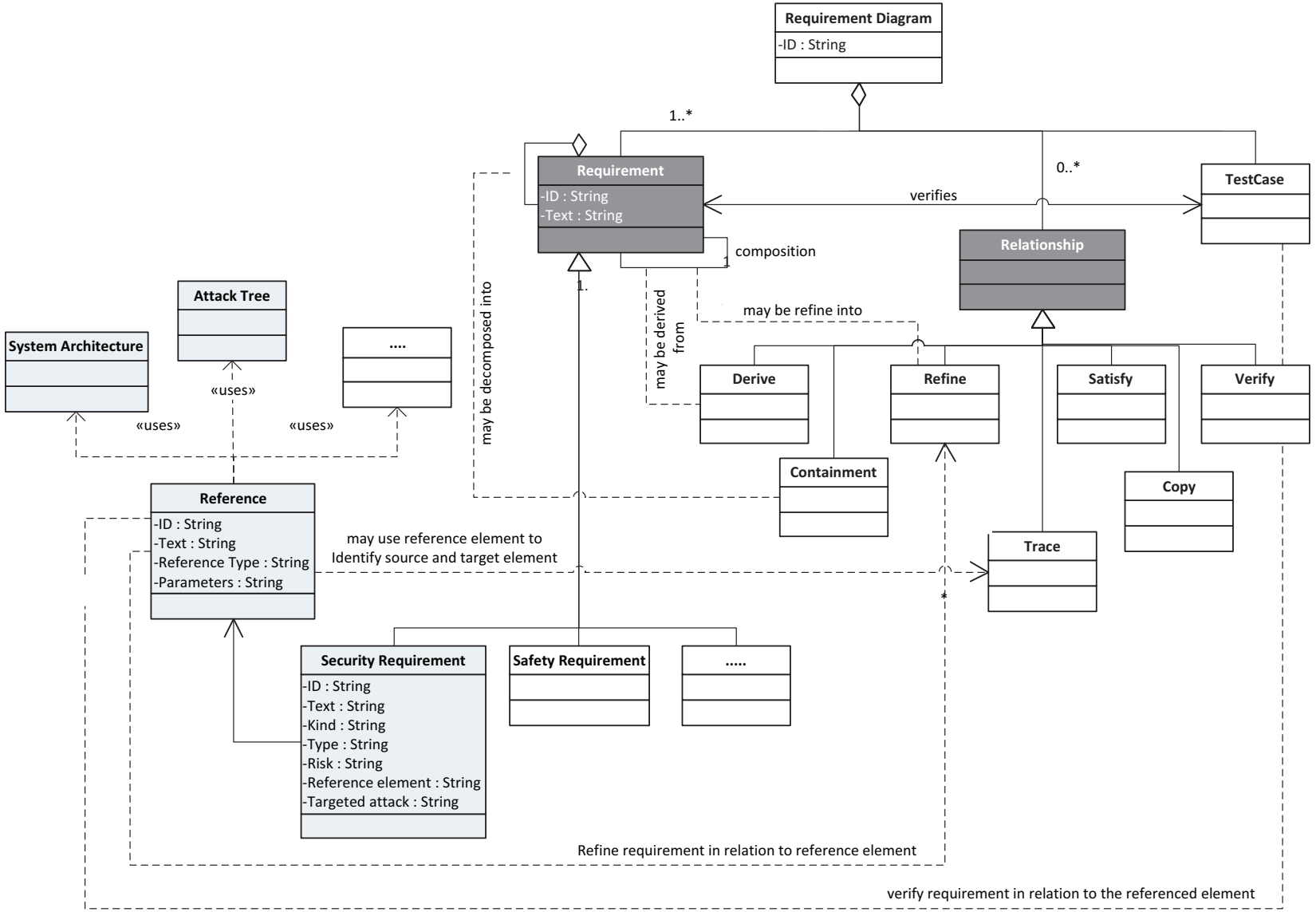


Figure 3.4: Meta-model for the security requirement diagram

3.3.4 Attack Modeling

As seen in the previous section, the definition of parameters and constraint blocks offer convenient means to constrain aspects of a system in a rule-based fashion that can relate the parameters of different parts of a system. The parametric diagram depicts how the parameters of the equations are bound to each other, to the properties of the system (i.e., the system performance, desired physical characteristics, etc.) that is being analyzed. The concept of security analysis is similar to the concept of trade-off analysis in that there is also more than one way to attack system assets, and an adversary may be trying them simultaneously or just a subset of them. More precisely, an adversary can use distinct attack paths or alternative approaches until reaching his attack objectives. This is often illustrated through the attack trees, which form a convenient way to systematically categorize the different ways in which a system can be attacked.

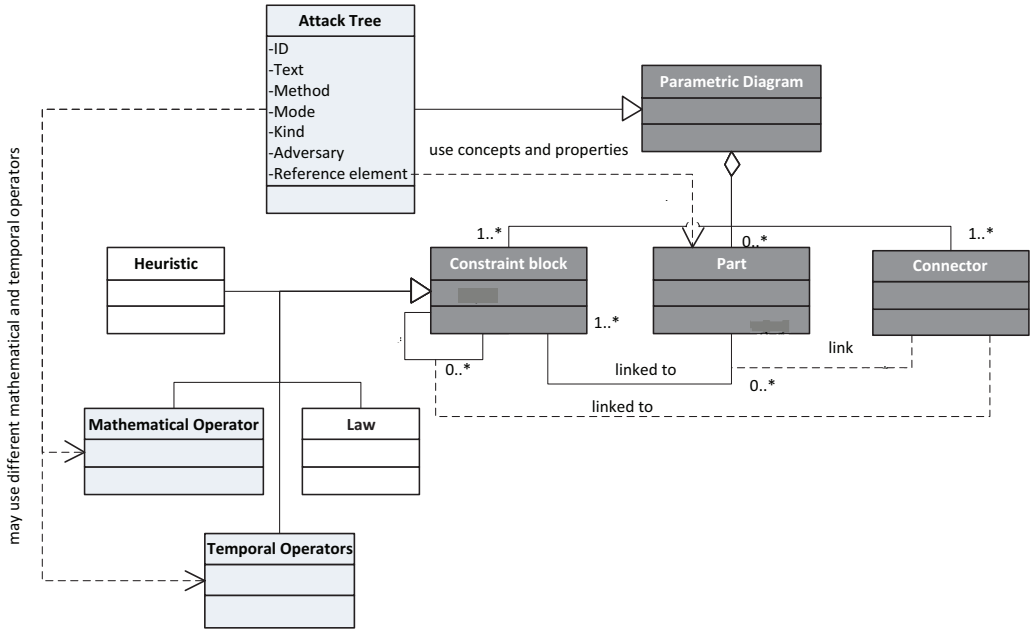


Figure 3.5: Metamodel for the SysML attack tree diagram

Basically, attack trees (the term was introduced by Schneier [133]) are multi-leveled diagrams consisting of one-root, leaves, and children nodes. In addition, different node values can be combined with AND, OR relationship to learn even more about a system's security flaws and weaknesses. Specifically, the purpose of an attack tree is to define and analyze possible attacks on a system in a structured way. This structure is expressed in the node hierarchy as well as in the form of logical operators (i.e., conjunctive (aggregation) or disjunctive (choice), etc.) for expressing interrelationship between different attack tree nodes. Thus, using both

logical operators and node definition retains the natural way security experts build the attack trees or fault trees [143, 129, 133, 17, 156]. Actually, these two building blocks (nodes and logical operators) of an attack tree can be modeled with the definition of constraint block with a object functions and the part element of the parametric diagram. Thus, at a conceptual level we can use parametric diagrams to model attack trees. Let us present how we suggest representing attack trees in SysML using the above-mentioned modeling constructs.

3.3.4.1 Attack Trees in Parametric Diagrams

Let us first focus on the extension of the parametric metamodel (see Figure 3.5) that is necessary for modeling attack trees. Following the extension mechanism suggested in the SysML specification where the stereotype mechanism is defined to extend the existing SysML classes, we create a new stereotype to represent security attacks: the «attack tree». This is illustrated in Figure 3.5. As mentioned earlier, we mainly focus in this thesis on expressing security knowledge to be shared and reused throughout the system development process to design a secure system. In order to integrate attack related knowledge, we extend the parametric diagram's "part" element with ontological concepts and properties from the attack tree ontology, presented in Section 2.4.2.3. We argue that such a representation is indispensable to precisely understand how attack trees can be manipulated during their construction and analysis. More details are given in Section 3.4 is about the introduction of security reasoning into SysML models. We use the "constraint block" element for the definition of set of constraints such as mathematical expressions (i.e., AND, OR, etc.) among the pieces of the security attack nodes. The objective of these operators is to show the relationship among difference attack nodes. More precisely, we use OR operator to represent alternatives ways an adversary tries to achieve his attack objectives. For instance, an adversary has to perform either one of the attacks "hijack authenticated session" OR "disconnect client" to accomplish his attack goal. AND relationship represent different steps toward achieving the same goal, for example, by assuming an adversary can gain root access of vehicle Communication Unit (CU) if and only if he can tamper the on-board communication unit. In our attack tree modeling approach, rather than considering only these two types of logical operators, we also consider temporal operators (i.e., AFTER, BEFORE, SEQUENCE, etc.). We in particular allow security experts to capture temporal dependencies between attack nodes and sequences in an attack. For instance, in order to install the bogus authority keys (see attack tree node AT.4.b in Figure 5.2), an adversary first have to switch an ECU into a re-programming mode. We can represent the ordering between attacks by using the SEQUENCE relationship as shown in Figure 5.2. We will exemplify these relationships in Section 5.3. We use a "connector" element to link zero or more "part" with constraint block. The use of a "constraint block", "part", and "connector" element for building attack trees is shown in Figure 3.6.

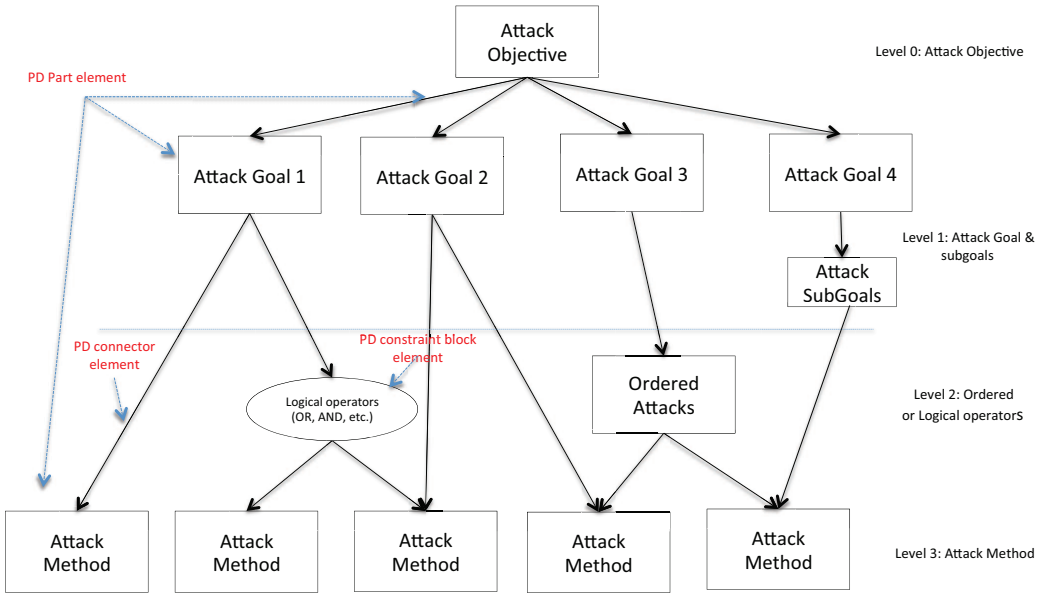


Figure 3.6: Generic attack tree structure

Attack Tree Modeling: An overall procedure for attack tree modeling looks like this:

1. Build attack tree rooted (Level 0) on an abstract "attack objective". We use the "part" element to model each attack tree node.
2. Its child nodes (Level 1) represent different "attack goals" that could satisfy this attack objective. Attack goals and attack objectives are linked via a binding "connector".
3. For each attack goal node:
   - Decompose into a number of "attack methods" (Level 3) that could be employed to achieve the attack objective
   - Specify the logical relationships (Level 2) between different attack methods, if there are. We use the "constraint block" to specify these logical expressions. At this stage, we also consider intermediate steps that represent attack method at a certain level of abstraction.
4. The attack tree terminates when leaf conditions (basic operations are described that gives all details of the attack) are reached that meet the adversary's capabilities.

The attack tree modeling approach that we advocate provides a bridge between the typical attack trees modeling approaches [133], and the anti-goal models approaches [151]. More precisely, the first two steps of our attack tree modeling approach are

equivalent to the KAOS anti-goal model [151], which provides the top down approach for modeling attacks. The next two steps correspond to the standard attack tree modeling approach, where attacks are identified from bottom up perspective. Figure 3.6 sums up these two approaches. Parametric attack trees are considered in much greater detail in Chapter 5, which gives a worked example showing the construction of attack tree diagrams.

3.4 Integration of Ontology Reasoning on Security with SysML

Every system engineering activity, whether it is a system model such as a structural model, a behavioral model, or a requirement model, defines and organizes concepts and properties into meaningful classes and relationships. As seen in previous sections, these concepts and relationships can be particular to a model (e.g., «deriveReq» relationship is specific to the requirement model), or they can be common to a family of models (e.g., a «trace» relationship is used to relate requirements with its source such as use cases, system architecture, etc.). In other words, models can be shared, contrasted, and their knowledge can be adapted in different models (e.g., knowledge related to use cases can be used in modeling activity diagram or in sequence diagrams). Similarly, ontologies define a common set of concepts, and terms that are used to represent knowledge by a vocabulary and, typically, logical construction of knowledge in terms of classes, subclasses and relationships among them. Syntactically SysML and ontology languages (i.e., OWL, OIL, etc.) have a lot of syntactic overlap. SysML uses a graphical formalism with elements for blocks, associations, part properties, and relationships between models as well as set of model elements. In contrast, ontology languages use classes, properties, relationships, and individuals as basic knowledge constructs. For instance, OWL defines classes by appropriate and implicit logical constraints on properties of their subclasses and concepts. Both approaches provide compelling benefits, system engineering and in particular security engineering, should make use of both. Recently, the integration of reasoning and ontological concepts in SysML has been discussed in the literature. For instance in [41, 39, 158, 40, 83], the authors reasoned about embedding SysML within a knowledge base, knowledge can be used to maintain models consistency as a design evolves. More precisely, the purpose of such integration is to enable engineers to employ reasoning, explicit documentation about system models, and to define more precise relationships in the course of a typical model-based development process. Similarly, several approaches [111, 112, 149, 11] are visualized towards using ontologies with UML modeling. However, these approaches are neither concerned with system wide engineering aspects, nor related to the reasoning on security aspects such as checking the relationships among different security artifacts (i.e. attacks, system architecture, security goals, security requirements, etc.).

3.4.1 Annotating SysML Diagrams with Ontological concepts

In this section, we focus on annotating security concepts and terms defined in the security ontologies (cf. Section 2.4.2) particularly security requirement ontology and security attack ontology with security requirement and attack tree diagrams. For other SysML models, such as system architecture, we consider [41] as a supporting approach for integrating ontological knowledge. Figure 3.7, presents an overview of an integration approach for embedding ontological concepts and terms into SysMLsec models. As previously stated in section 3.3.3 we can add the ontological concepts and terms into SysML models by extending the SysML metamodel by including the user defined stereotypes or properties and tagged values. Let us first consider the SR diagram (presented in section 3.3.3), and map SR related ontological classes (cf. Section 2.4.2.4) to the SR diagram. We build the integration approach based on three core ideas:

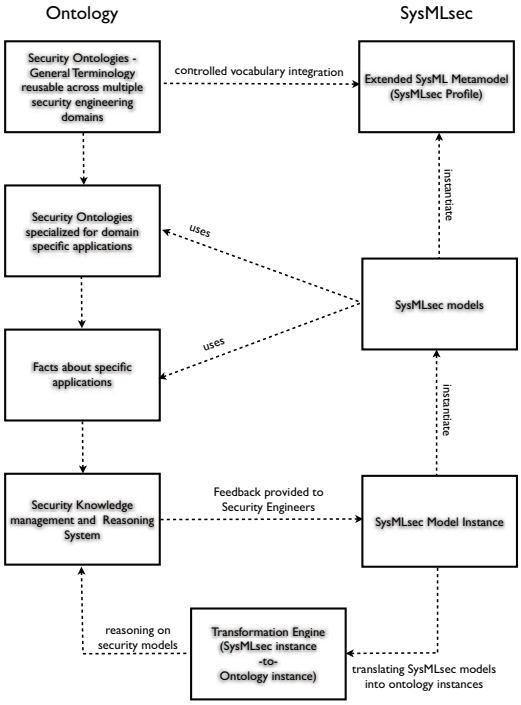


Figure 3.7: Integration of ontology reasoning on security with SysML

- We have defined the «SecurityRequirement» stereotypes (see Figure 3.8) to represent the security requirement ontology in the SysML RD.
- We integrate high-level ontology classes (see Level-1 in Figure 3.8.b) as a SysML requirement's diagram properties (i.e., type, kind, etc.) as shown in Figure 3.8.a.
- We use ontology subclasses (see Level-2/n Figure 3.8.b), as tag values of the SysML requirement's diagram property element. This is illustrated in Figure

3.8.a. These values constitute a controlled vocabulary. Thus, it provides a canonical set of mapping mechanism in order to deal with integration of ontological concepts into the SysML.

According to these rules, every SR diagram extended with a «SecurityRequirement» stereotype is also associated with ontology concepts and terms as shown in Figure 3.8.a. The diagram consists of three parts; standard SysML requirement properties (e.g., id, text), extended ontological properties (e.g., kind, type, role), and properties for our reference concept (e.g., targeted attack, reference element). The discussion here will be limited to the extended requirement's property constructions that can be directly translated to ontology classes. It can be seen that, for each high-level class of the security requirement ontology (see Figure 3.8.b.) we basically define a new property element in the requirement diagram. This is illustrated in the Figure 3.8.a. These properties are then populated with the subclasses and concepts defined in the security requirement ontology as its tagged values. In particular, the properties and tagged values are specified in the same manner as the classes and subclasses concept described in the security requirement ontology. For instance, for the Type property, we have constructed the list of tagged values (i.e., FSR, NSFR, etc.) by using the sub classes of the "Type" class that depict the tree-based overview of all classes and sub classes and their relationships (see Figure 3.8.b) within the SR ontology. This allows security engineers to browse different ontological concepts and terms within the SysML models, and to reason about them (cf. Section 3.4.2) at different levels of abstractions from multiple dimensions. In particular, such integration will help security engineers to keep the security requirement description and security requirements related knowledge such as type, kind, etc., consistent as development proceeds. For instance, during the initial stages of system conceptualization, security requirements are often generic like "prevent denial of service attack", these requirements are refined to more concrete security requirement description (i.e., "availability of an application"), or to security mechanisms such as "Filter messages that are making an application unavailable", as the system evolve. In this case, security engineer can map these requirements to more specific classes and concepts (i.e., application specific) by selecting appropriate tag values from the controlled vocabulary as shown in Figure 3.9. In this way it is possible to make sure that, for example, particular information set belongs to specific knowledge branch, and is confined within its hierarchical structure.

Following the same line of reasoning suggested above where high-level ontology's classes are used to represent the properties, and its subclasses represent tagged values, we have extended the definition of "Part" element (presented in Figure 3.5) of the Attack Tree diagram. Figure 3.9, is a schematic diagram that shows the integration of concepts and terms from the security attack ontology in the SysML Attack Tree diagram.

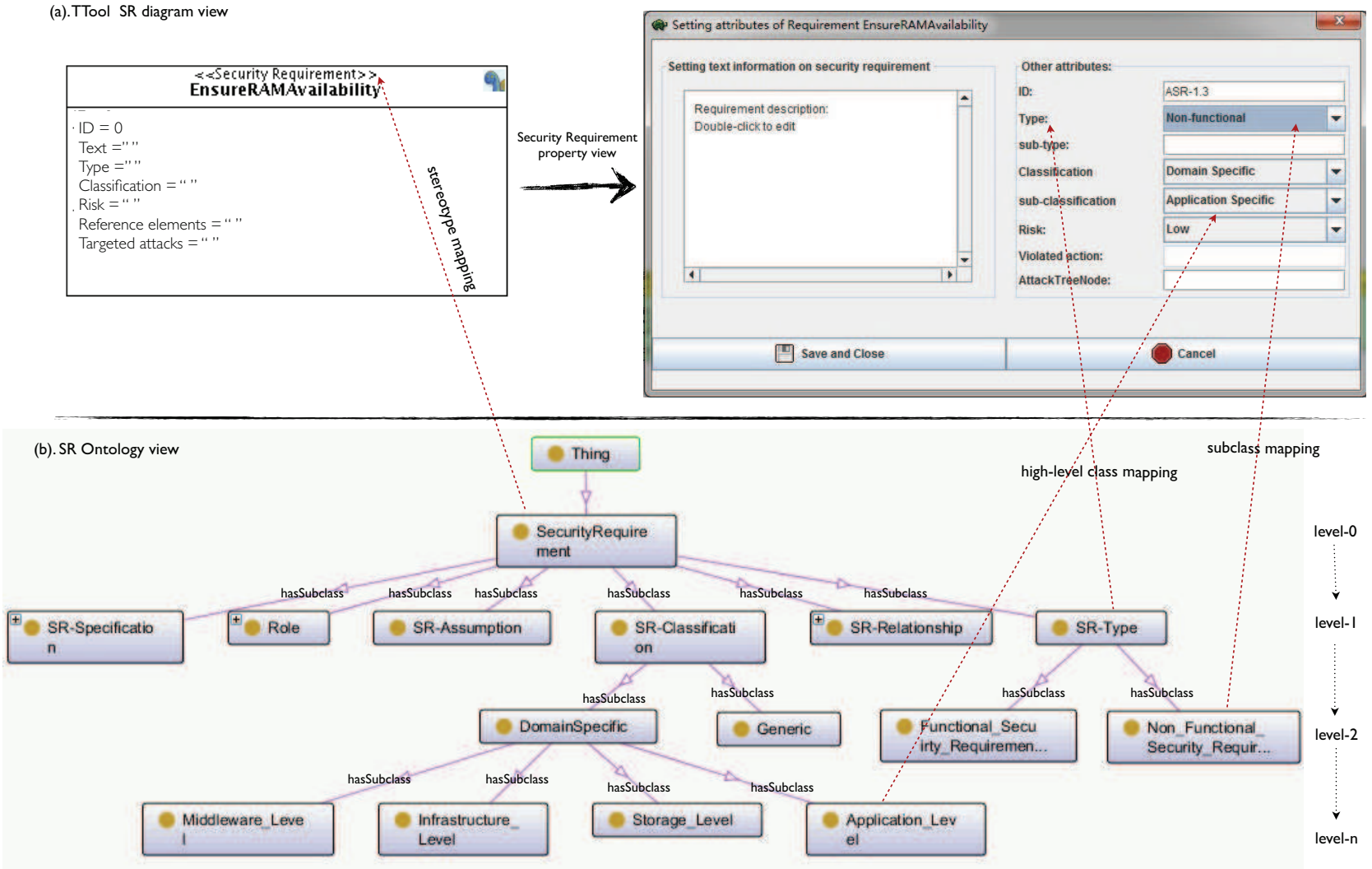


Figure 3.8: Mapping of the SR ontology concepts into the SysML SR Diagram

We have defined the «attack» stereotypes to represent the security attack ontology in the "part" element. While the properties of the "part" element shown in Figure 3.9.a are concepts and terms defined in the security attack ontology shown in Figure 3.9.b. For instance, the different high-level classes of the security attack ontology such as attack method, attack mode, attack type, etc. are mapped to a corresponding property name (see property view in Figure 3.9.a), and their tagged values corresponds to subclasses of these classes. For example the "Adversary" class takes *Layman, Expert,* and *Professional* as its tagged value. Accordingly, we map all other concepts and terms defined in the security attack ontology into the "part" element. From a modeling perspective, security engineers can include this information by double click on the attack tree node which shows the property view of the part element as depicted in Figure 3.9.b. In a similar way, this information can be viewed in the SysML diagram by a simple click on the attack tree node as shown in the Figure 5.2, for *DenialOfService"* attack.

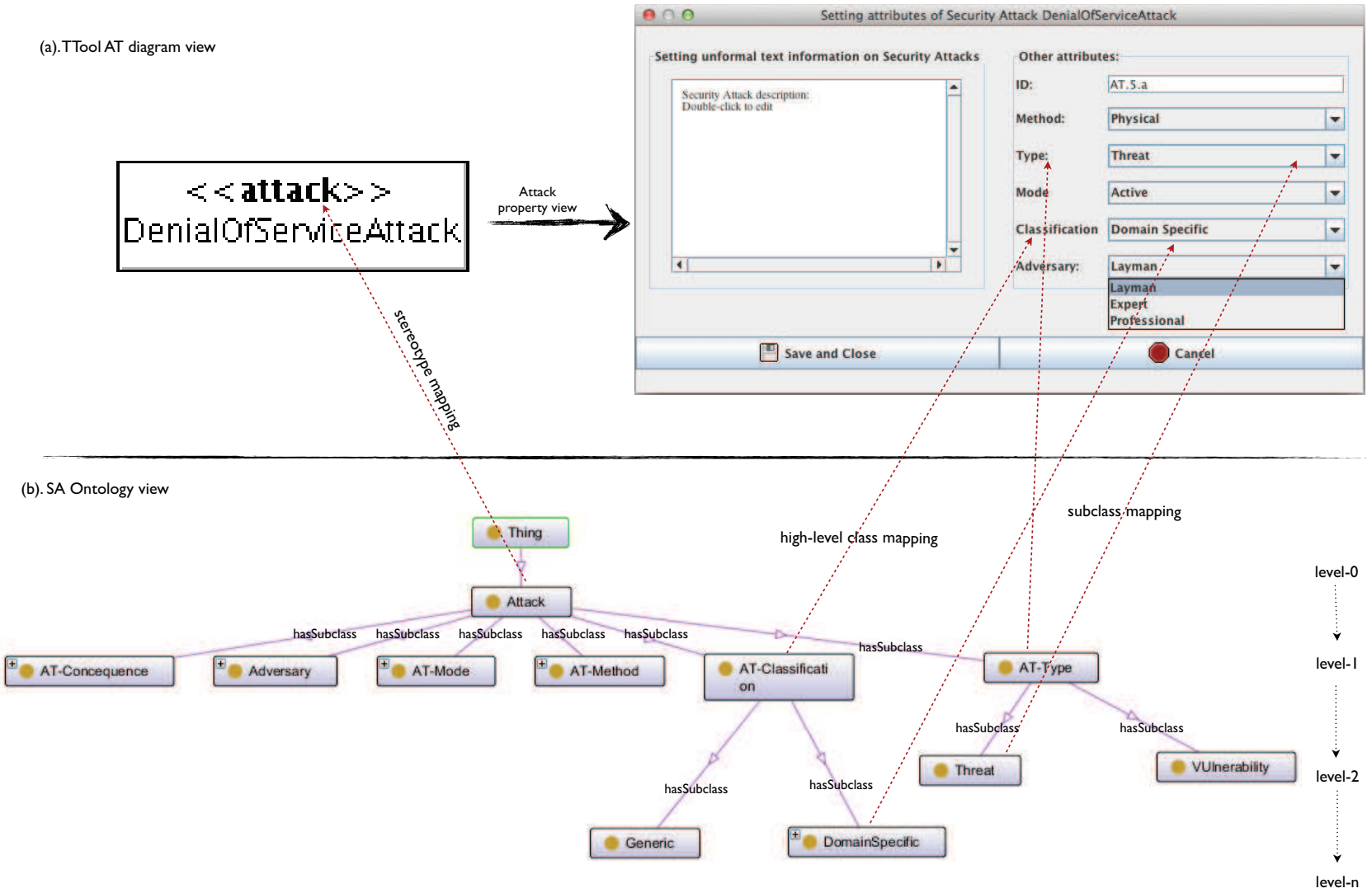


Figure 3.9: Mapping of the security attack ontology concepts into the SysML attack tree diagram

3.4.2 Reasoning with SysMLsec Models

In this section, we describe the extent to which we can use the capabilities of ontologies to reason about these different security concepts defined in SysMLsec models. In particular, our objective is to enable the security engineers to have access to various ontological concepts and terms, and to reason on these models. Although, with integration of ontology classes and subclasses into the SysML diagrams, we already provided the partial reasoning capabilities to reason about different security concept within the SysML models. More precisely, when security engineer select a particular concept in the SysML diagram, for instance, SR is a "Domain specific" requirement, we annotate the structure of the sub-classification with the tagged values that belong to the domain specific class such as an application specific, middleware specific, etc, as shown in the property view of Figure 3.8. In a similar way, for each ontology class we apply the same approach and limit the knowledge space for security engineers to specify only those concepts and terms that belong to the super class or the parent class. Thus, provide means to reason about security concepts within the SysML models, which brings additional power to the development of security models like consistency checking (i), concept satisfiability (ii), and concept classification. The shortcoming is that we cannot specify the reasoner calls in relation to one another or in relation with other security models such as security goals, attack, system architecture, etc., which is our core objective. In order to fulfil this design objective, we have implemented the "SysMLsec-to-Ontology" translation engine as shown in the Figure 3.10. The translation engine, we have implemented for mapping from SysMLsec models to the OWL description, contains a set of rules that match security constructs and transform them into equivalent instance of ontology such as shown in Figure 4.5.

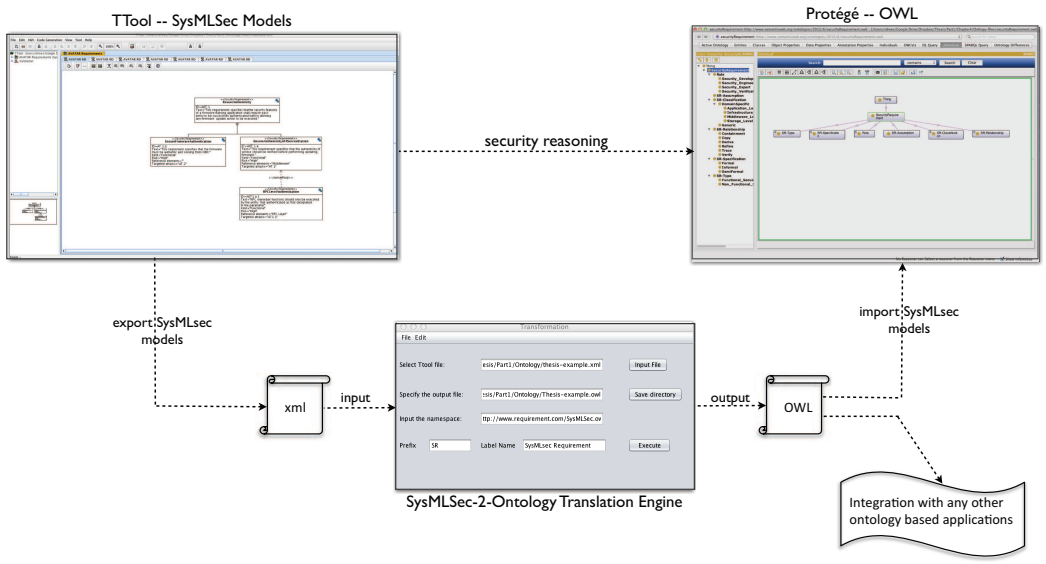


Figure 3.10: SysMLsec to ontology translation engine

The primary purpose of this translation engine is to make the security engineers able to reason about their security models using well known and efficient reasoning engines such as SPARQL [121], OWL-QL [34], RACER [44], SQWRL [104], etc. Engineers can directly make use of reasoning capabilities of these engines within the context of current engineering practice and tools without building and using some separate ontological models. In particular, our objective is to give the security engineering a more precise way to employ reasoning in the course of a typical model-based development process. On the other hand, since we obtain an OWL described document, we can integrate our SysMLsec models with any other ontology based applications such as integrating the security requirement knowledge with security resource annotating approaches like [77], or providing input to the ontology based risk analysis approaches [30] in order to compute risk metrics.

In this dissertation, we refer to SQWRL (Semantic Query Web Language) [104] as a query language because of its concise, readable, and semantically robust semantic. SQWRL is a SWRL-based [53] query language that can be used to query OWL ontologies and provided in Protégé 4.2 beta version [147]. To retrieve knowledge from OWL ontology, SQWRL provide SQL-like operation. The form of rule is

$$antecedent \rightarrow consequent$$

In this rule an antecedent part is referred to as the body, and a consequent part is referred to as the head. Both the body and head consist of positive conjunctions of atoms:

$$Atom \ \wedge \ Atom \ \rightarrow Atom \wedge \ Atom$$

This rule can be read as if all the atoms in the antecedent are true then the consequent must also be true. Here, an atom is an expression of the form P(arg_1, arg_2, ...arg_n), where p is a predicate symbol and arg_1, arg_2, ..., arg_n are the terms or arguments of the expression. In our approach, the predicate symbols can include security ontology classes (i.e., asset, goal, attack, security requirement, etc.), properties (i.e., hasFunction, hasSequence, hasAvoidGoal, etc.) or data types. Arguments can be class individuals (i.e., SR-Type, AT-Classification, AvoidGoal, etc.) or data values (i.e.,), or variables (i.e,) referring to them. In the further course of this thesis, we will use the above-mentioned SQWRL query expression to retrieve, manipulate, and reason about different security-related information.

3.5 Conclusions

The interest of a SysML should be measured by how it benefits the system engineering process. Given that SysML is a powerful approach to an essential part of this process, the provision of means to better model security aspects, in particular, SRs and attack tree modeling is obviously beneficial. Potentially, SysML could reach

a wider user community. Modeled in SysML, security aspects can be incorporated into the rest of a system engineering process, seamlessly linking the security engineering process part of a project to the whole. Furthermore, we implemented these capabilities into the TTool engine that also supports our extended SysML model for defining security requirements and attack tree modeling. This tool readily supports the SRE methodology that we advocate.

Part II

By Design Security Requirements Engineering

Chapter 4

Running Example: The Firmware Flashing Process

4.1 Introduction

Diagnosis of cars is not a new goal in the automotive industry. It has existed since cars were first designed. During the last 20 years car diagnosis gained more importance because of the increasing use of electronics in cars. Standards were defined not just to allow different manufacturer's ECUs to communicate with each other in an in-vehicle network system, but also to allow different diagnosis tools to have access to diagnosis data. In addition, there is a shift towards multipurpose ECUs and usage of flash memory technology in the microcontrollers.

Besides these trends in the design of automotive on-board IT architectures, new external communication interfaces, fixed and wireless, are becoming an integral part of on-board architectures. One key factor for this development is the integration of future e-safety applications based on V2X communications (external communications of vehicles, e.g. with other vehicles – V2V, or with the infrastructure – V2I) which have been identified as one promising measure for increasing the efficiency and quality of operational performance of all vehicles and corresponding intelligent transportation systems.

These connected cars are revolutionizing the automotive industry. Yet as the amount and the complexity of ECU and firmware inside cars increases, so too does the need to update the firmware in order to provide new functionality and perform firmware maintenance. Firmware updates are crucial for the automotive domain, in which recalls are a very costly activity and thus should be avoided where possible. Google has showed the practicability of remotely updating devices for their Android telephones. With this, they have a powerful tool to react on discovered security flaws in very short time [137].

In the automotive domain, update intervals are calculated in quarters of a year and not quarters of a day right now. This paradigm is about to change and security mechanisms within the car provide the necessary building blocks. With the arising "always-connected" infrastructure, it will be possible to perform over-the-air (OTA) diagnosis and OTA firmware updates (see Fig. 4.1), for example. This will provide several advantages over hardwired access, such as saving time by faster firmware

updates, which improves the efficiency of the system by installing firmware updates as soon as the car manufacturer releases them.

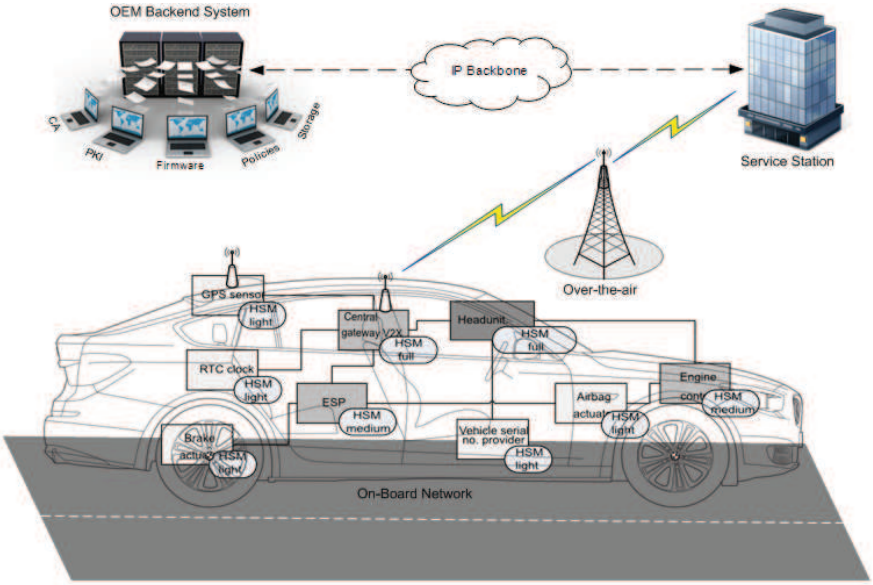


Figure 4.1: Over-the-Air firmware flashing process

In this dissertation, we are going to use firmware-flashing application as a running example. There are several reasons for choosing this as the subject of the example.

- The firmware flashing is an excellent example of automotive systems that may be possible in the real world. While this functionality herald a new era of safety in transportation, new security requirements need to be considered in order to prevent attacks on these systems.
- The firmware flashing is an example of a system that consists of both hardware and software as well as is composed of internal and external interfaces. Moreover, it contains a number of cooperating sub systems such as service station system, in-vehicle system, and OEMs backend systems. Thus, allow us to perform in-depth security analysis and evaluation of complex automotive architecture.
- There are many aspects that can be modeled, from the structure of the firmware flashing process, the parts used to build it, to the behavior that the flashing process has to achieve. These various aspects lend themselves to the use of most of the SysML diagrams.

The remainder of this chapter is organized as follows: Section 4.2 outlines the firmware flashing application scenario. In Section 4.3, we will present security goals that we have identified by analyzing the description of firmware flashing application.

Section 4.4 presents a detail description of system architecture design including behavioral and structural models. Finally, the chapter summarizes the results that we achieved regarding modeling of firmware flashing application.

4.2 Firmware Flashing Use Case Specification

The firmware flashing use case description is based on a common architecture [74] and topology for the in-vehicle communication networks consisting of ECUs, sensors, and actuators as shown in Figure 4.2. However, the use case of course can also be mapped to other instantiations of the reference model. The purpose of this use case is to describe the possibility of installing/updating firmware in the vehicle from an external device. In particular, this use case demonstrates how after receiving the request of the vehicle owner, a service station using a DT, the Diagnosis Tool, will try to assess the state of a vehicle located in their area without making any physical connection to the vehicle. The diagnosis of the vehicle should even be possible if the vehicle is not in the area of the service station, by using an Internet connection. This is necessary since real time data when a vehicle is moving can help to discover malfunctions, which are not detectable when the car is in the service area. The use case involves three communication entities, two within the vehicle (CU and ECU) and one outside of the vehicle (DT). The DT uses a wireless connection to connect with the Communication Unit – CU. The CU is hard-wired connected to the ECU within the vehicle. To simplify the modeling and description of firmware flashing process, the use case is split into two sequential phases named remote diagnosis and remote flashing, respectively.

- **Remote Diagnosis:** In this phase, the service station has to first connect via Internet and Wireless LAN to the in-vehicle network. An employee of the station using the DT, sends a connection request to the vehicle. A connection response is sent back to the DT. Once the connection is established, the DT sends, depending on the option chosen by the employee of the service station, requests to read out diagnosis information from the ECU it wants to check such as ECU type, firmware version, and date of the last update. Assuming the ECU type is known, a comparison is made to figure out the need of an update of the version.

- **Remote Flashing:** A possible consequence of previous step (diagnosis) would be the update of the software version of the ECU to remove bugs or to improve the functionality. The DT sends a request to open a re-programming session at the ECU level. Once the re-programming session is open, the DT sends the new firmware version to the RAM of the ECU. The communication still goes through the CU. The firmware is flashed in the ROM, and the date is saved. At the end the DT closes the re-programming session at the ECU level and the connection with the vehicle.

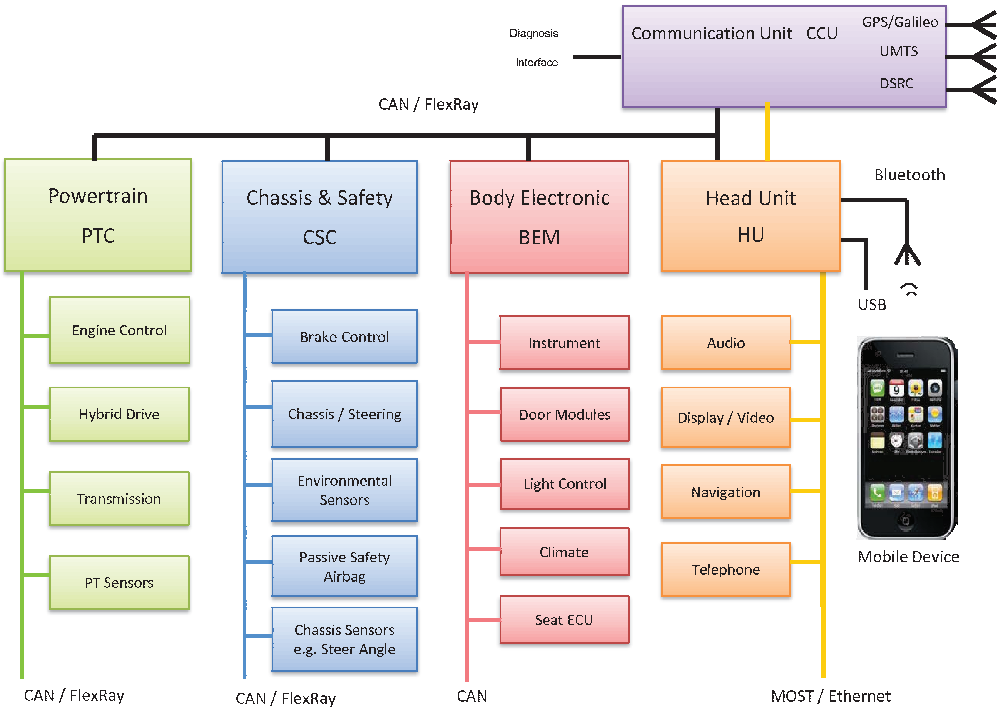


Figure 4.2: EVITA Use Case reference architecture [74]

4.3 Security Goals

We start our SREP by analyzing the problems and deficiencies in the firmware flashing specification. At the early stages of system conceptualization, we have identified the following preliminary goals:

- **Avoid Goals:** are related to the malicious behavior that an adversary can perform during the firmware flashing process. For instance, an adversary can "install older version of firmware", or "flash malicious firmware". If it is not the case, it is still possible to an adversary can" abort the firmware flashing process" by jamming the wireless communication channel. In a similar way, we go through each detail of the specification and identify goals that may prevent us from successfully installing firmware in the vehicle.
- **Achieve Goals:** We have applied a similar approach like the above to find the achieve goals, that firmware flashing process must achieve throughout its lifecycle. For instance, "Only valid users are allowed to access firmware flashing functions" goal is identified for example to avoid the unauthorized access to firmware flashing functions as well as from sending flashing commands. Some of these goals are listed in figure 4.3.

As a result of this activity, we use this knowledge and instantiate the goal ontology (cf. Section 2.4.2.1) for storing the knowledge about different goals related to

firmware flashing process as shown in Figure 4.3. In addition, we associate AchieveGoal with SRs (see figure 6.1), whereas, AvoidGoals are modeled as attack goals in the SysML Attack Tree diagram, as shown in figure 5.2. This goal knowledge base will later help us to identify more concrete security attacks and security requirements. We will exemplify this in coming chapters.

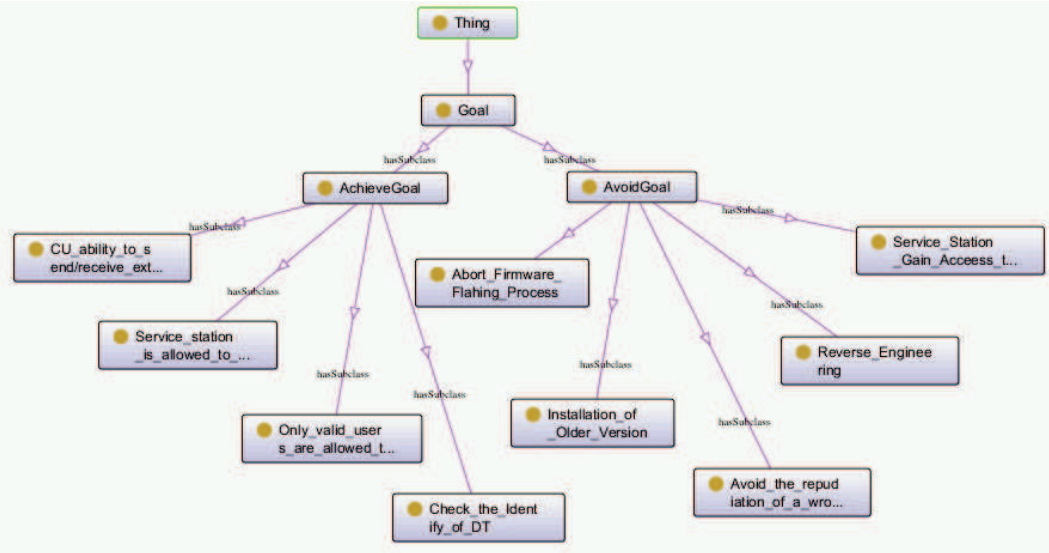


Figure 4.3: Ontological representation of the security goals

4.4 System Architecture Design

We now dictate how the most important assets within the firmware update process can be identified and modeled from the high-level security goals and scenario specification, and what interactions and collaborations are required in the system to allow for their successful integration and function. A combination of mechanisms are used to accomplish this including behavioral and structural models.

4.4.1 Behavioral Models

In this section, we provide a detail description of firmware flashing application from its behavioral aspects. We particularly consider three combinations of behavioral models (Use case Diagram, Activity Diagram, and Sequence Diagram) to describe the functionality of firmware application.

4.4.1.1 Use Case Diagram

In the first step, we start with modeling use case diagram in order to describes the functionality provided by a firmware flashing application in terms of actors, their goals represented as use cases, and any dependencies among those use cases. Figure 4.4 shows what the above use case might look like in SysML schematic form. As you can see, several firmwares related scenarios are extracted from the high-level specification (cf. Section 4.2) and joined together in one use case. In our firmware flashing application, the names of some use cases are: "ConnectionRequest", "DiagnosisCheck", and "Re-programmingSession". We further associated these use cases with actors by including everyone and everything that needs to exchange information with the system. In our example, the actors included in the specification are the DT, the CU and the ECU.

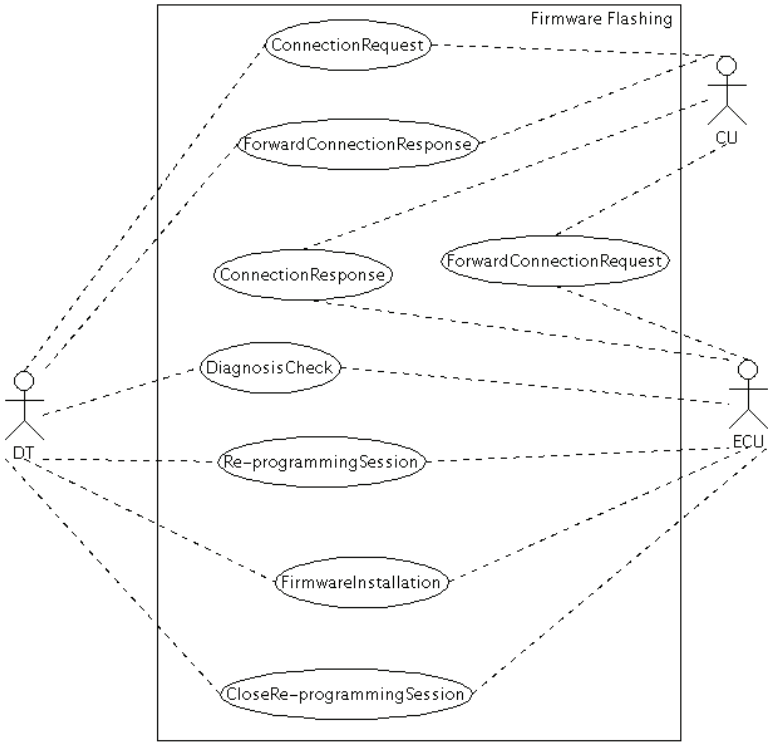


Figure 4.4: Use Case Diagram - Firmware flashing process

The general idea of using use case diagrams is to express different knowledge related to scenario and then asking what outwardly visible, measurable result of value that each actor desires. Following our ontological mapping principal (cf. Section 3.4.2), we extract the knowledge modeled in the Use case diagram, and generate a corresponding instance of the system architecture ontology (cf. Section 2.4.2.2), particularly, instance of a behavioral model. Figure 4.5 illustrate the ontological

representation of Figure 4.4, where "actor" subclass of use case class is used to hold the different actors modeled in the Use case Diagram and "case" subclass is used to hold values of different cases. In addition, we also capture association between "actors" and "cases" by using a "hasAssociation" relationship as shown in figure 4.5.

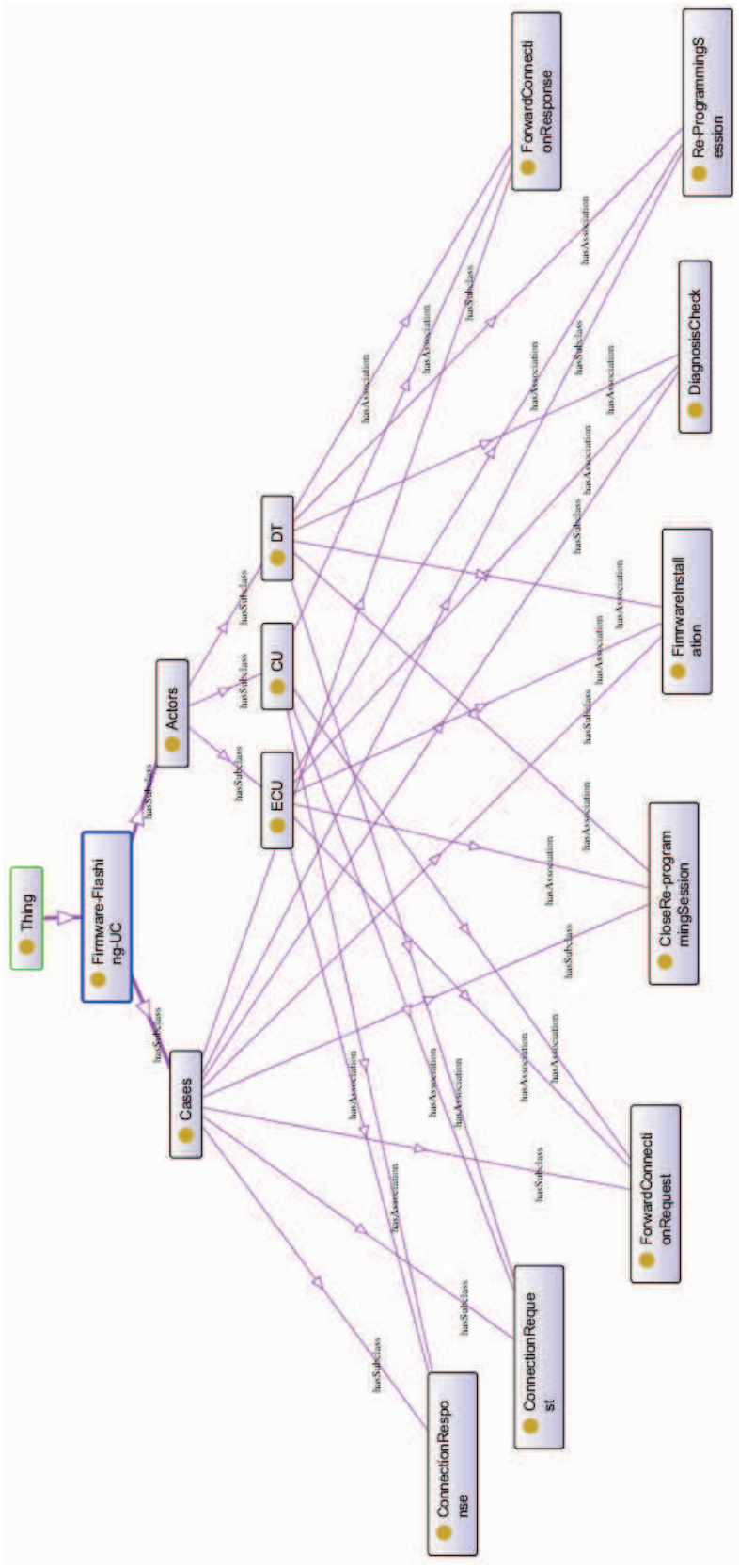


Figure 4.5: Ontological representation of the firmware flashing process Use Case

4.4.1.2 Activities Diagram

We now adopt the SysML activity diagram to discover and reason about the different activities/actions of the firmware flashing process. The main reason behind using this diagram is to analyze a use case by describing what actions need to take place and when they should occur. More specifically, we use activity diagram to describe the operational progression that defines the sequences of firmware flashing operations and the realization of operation. Following our example, figure 4.6 explains the use case of figure 4.4.

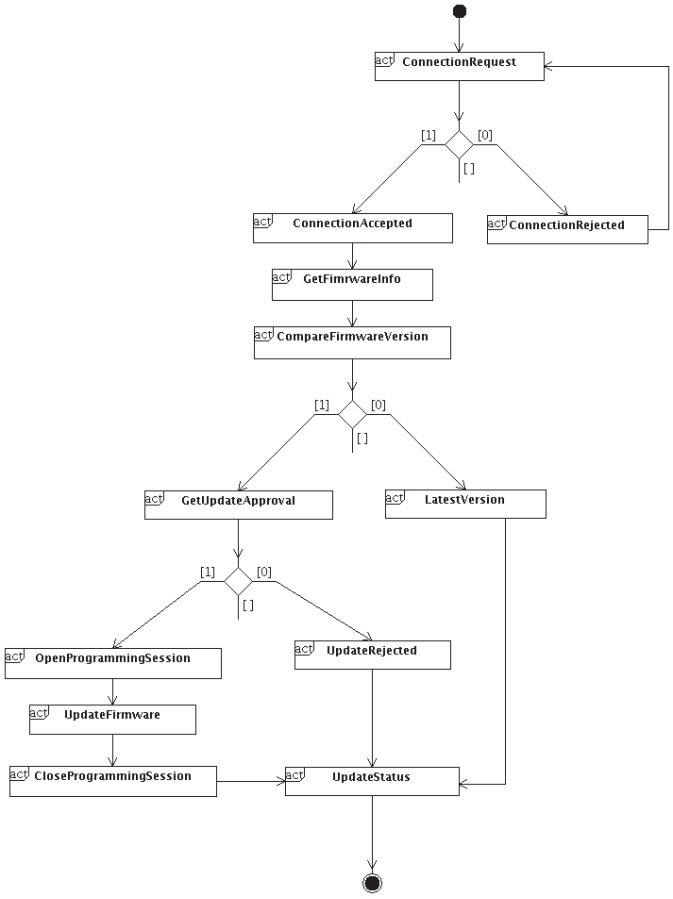


Figure 4.6: Activity Digram - Firmware flashing process

However the behavior of a task is sequential and we characterize each task by the following elements:

- Behavior that describe the task's functionality.
- A set of input communication connectors
- A set of output communication connectors

- A set of attributes, that are used by the task's behavior such as integer or boolean (see choice element in Figure 4.6), and
- A name that represents the identifier of the task, which must be unique.

In addition, we use activity diagram to link up with the blocks and the structural modeling to model continuous streams, which we will explain in the next section. We feel that the usefulness and effectiveness of SR can be increased manifold by considering the SysML activity diagram in the SREP. Our view is that, using activity diagram we are in a position to identify and express SR that are not restricted to the current state of the system but also to its past and future history.

4.4.1.3 Sequence Diagram

As part of the system architecture design, we also specify the system behavior using SysML Sequence diagrams, which in the case of the firmware-flashing example are used to describe the communication between the DT and the in-vehicle components such as CU and ECU. A sequence diagram shows object (DT, CU, and ECU) interactions arranged in time sequence. It depicts the objects involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Objects required for accomplishing a firmware flashing tasks are shown as lifelines as illustrated in figure 4.7.

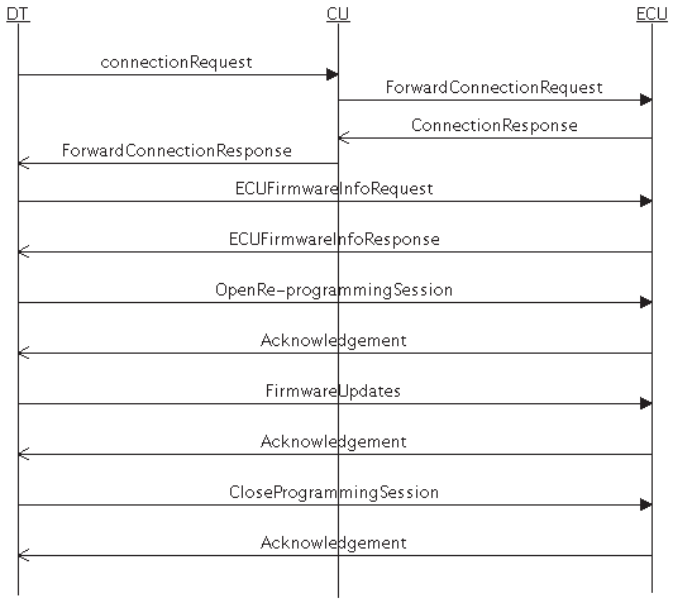


Figure 4.7: Sequence Chart - Firmware flashing process

To be more precise, we use Sequence diagrams primarily to design, document, analyze and validate the architecture, interfaces and logic of the system by describing the sequence of actions that need to be performed to complete firmware flashing

process. Figure 4.7, clearly depict the sequence of events and show when objects are created and destroyed. As like Use case model, we also generate ontological instance of Sequence diagram as shown in the figure 4.8. In particular, we capture the objects (DT, ECU, and CU) in our "Object" class, events (i.e., connection request, connection response, etc.) in our "Message" class, and sequence of events (i.e., msg1, msg2, etc.), in our "Sequence" class, defined in the ontology (see Section 2.4.2.2). Moreover, in order to capture the relationship among all these elements, we have used the "hasMessage" relationship to relate events with objects, and "hasSequence" relationship is used for capturing the order of events as shown in figure 4.8. However, further information such as time outs, synchronous/asynchronous messages types, etc., can also be expressed in the ontologies.

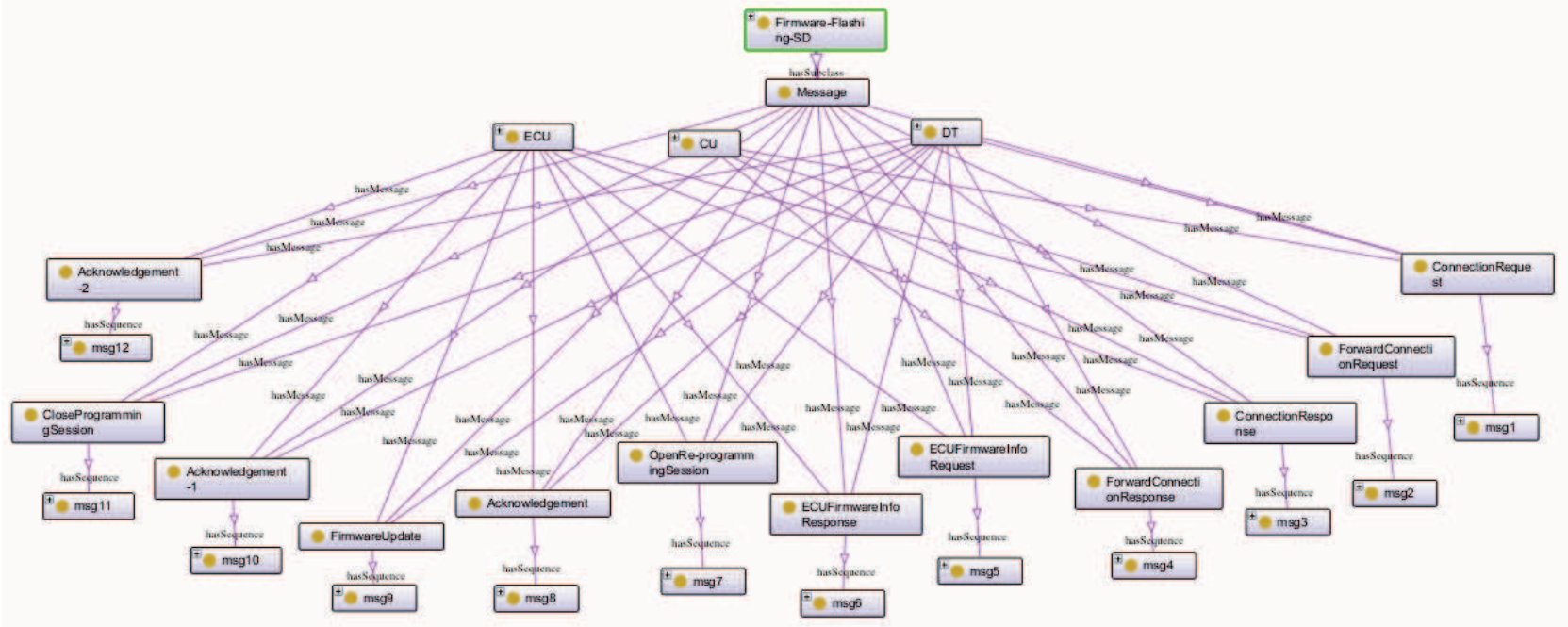


Figure 4.8: Ontological representation of the Firmware flashing process Sequence Diagram

4.4.2 Structural Models

Previously we have discussed the point that the most obvious concept in embedded system architecture design is that of the structural models. A variety of architectural structures are used to introduce technical concepts and fundamentals of an embedded system design like Y-chart approach [76], SHE Methodology [148], COSYMA approach [31], etc. Most of these approaches are concerned with designing embedded system architecture by focusing on representing system architecture from the high-level specification through to hardware synthesis and software compilation. The general approach used in these approaches includes the exploration of hardware software characteristics such as area, speed, memory limitations, power consumption, maintainability, upgradability, testability, reliability, etc. Note that most of these approaches are concerned with analyzing the non-functional or quality related requirements of embedded systems. Therefore, a useful characteristic would be to use these approaches also from the security requirement-engineering point of view. This would permit us to analyze the security aspects from the early design stages and in relation to the different architecture levels and their interrelationships. On the contrary, analyzing the security concerns with design exploration step allows system engineers to investigate the influence of security aspects on the system performance (such as latency, throughput and resource utilization). The results may inspire the designer to improve the architecture, restructure/adapt the application, or modify the mapping of the application.

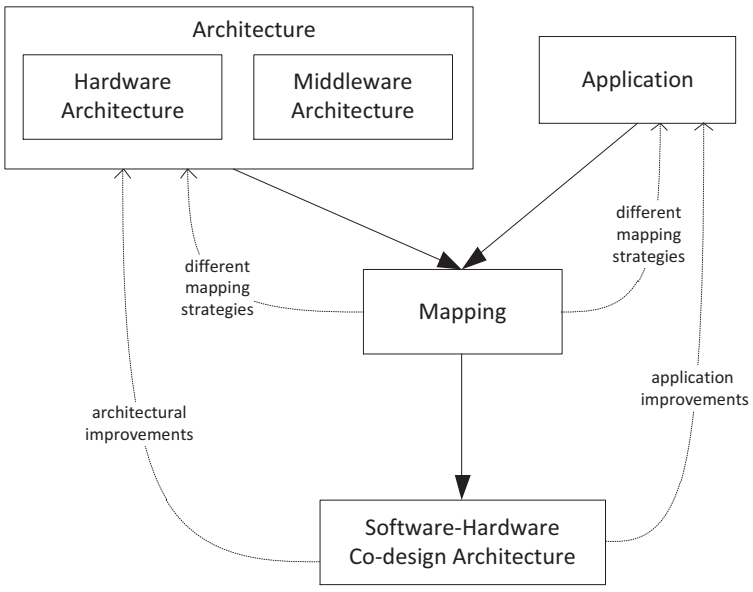


Figure 4.9: Extended Y-Chart approach

We particularly use the Y-chart approach, because of its capabilities towards enabling the designer to address separately functionality, architecture and mapping

issues to develop a fully mapped system. It is important to notice that the Y-Chart approach clearly identifies three core issues (i.e., architecture, application, and mapping) that play a role in finding feasible architecture. Be it individually or combined, all three issues have a profound influence on the system design [76]. It fits conveniently with our system architecture description, where we need to define potential security requirements for targeted application expected to run in co-designed hardware and software. However, as our objective is to identify security weaknesses and security requirements in relation to different architecture levels from the early stages of system conceptualization, we slightly modified the conventional Y-chart approach and also consider mapping of functions on the software/middleware layer. Even if the architectural structures are rough and informal at the early stages, it is still better than nothing. As long as the architecture conveys in some way the critical components of a design and their relationships to each other, it can provide us with key information about whether the device can meet its requirements, and how such a system can be constructed successfully. This is depicted in figure 4.9. The design flow embraces the following three steps:

4.4.2.1 Application

Application represents the functionality to be performed by the targeted system. In particular, application is structured around the notion of "task" that holds a functionality (task's behavior). We describe the behavioral description of task in terms of a SysML Activity Diagram (see Figure 4.6). However, note that one behavioral description per task is required. A task has a dedicated behavior that will define how it will execute and define its communication scenario with the other tasks. Figure 4.10 depicts a firmware flashing application example modeled with the TTool toolkit. The application in the example is the composition of a set of tasks (DiagnosisConnectionInitiation, DiagnosisRequestManagement, etc.) and the set of Communication Connectors such as connection request, connection response, etc.

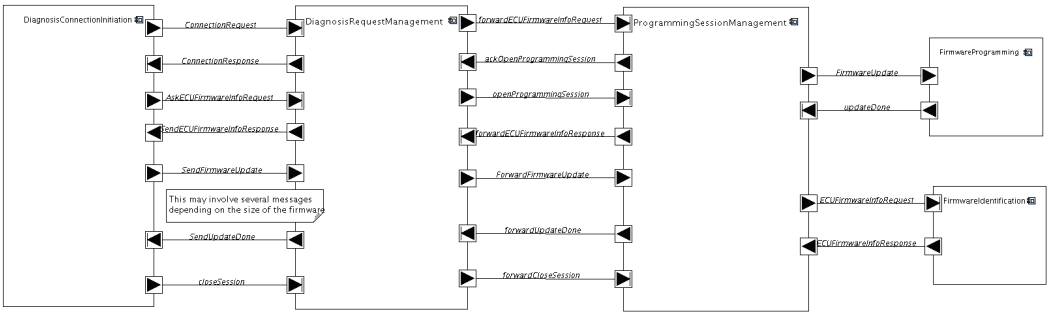


Figure 4.10: Functional view - firmware flashing process

These different tasks functionalities are interdependent as each task is an origin and destination for different communication connectors. For instance, "Diagnosis-ConnectionInitiation" exchange data with "DiagnosisRequestManagement" through the channel "ConnectionRequest", but it is a destination for the event "Connection-Response" sent by "DiagnosisRequestManagement". With the same logic, we build the detail functional view of the application. In particular, we want to extract the functional path (information flow path) of the firmware flashing application from this view in order to analyze what data and messages are exchanged between different tasks. This will certainly help us to perform more detailed security analysis as well as to express more concrete security requirements. To be more precise, we define functional path as:

The functional path of an application is a tuple consisting of a set C of events and data channels, and of a set F of functions. C and F are defined as follows:

- F_c is included into F.
- C contains all channels which destination is a function of F.
- F contains all functions that output messages in channels of C.

Therefore, the functional path of an application includes all data and events that are taken as an input by all functions involved in the direct or indirect production of the application.

4.4.2.2 Architecture

Following the line of reasoning suggested in Y-Chart approach, targeted architectures are modeled independently from applications as a set of interconnected generic hardware nodes. A set of parameters permits to calibrate components for their application area. At this stage of system architecture development, we use the DIPLODOCUS profile for modeling diagrams hardware architecture, as well as for specifying high-level description of software/middleware layer, which is already integrated in TTool engine [5]. A hardware architectural description is a collection of interconnected hardware nodes. Those hardware nodes are computing nodes (CPUs, I/O devices, hardware accelerators), storage nodes (RAM, etc.), sensors and actuators. The interconnection between those nodes is described in term of busses, networks and wireless links (see Figure 4.11). In the context of software architecture, we define it from the functional/logical-modeling viewpoint. Functional modeling, focuses on building the functional architecture of the system by breaking the problem domain into a set of non-overlapping and collaborating components. In other words, software architecture is focused on organizing components to support specific functionality such as processing tasks, sending message/data, etc. In order to have common mapping, we classify hardware and software components into three activities: 1) Computation, 2) Communication, and 3) Storage activities.

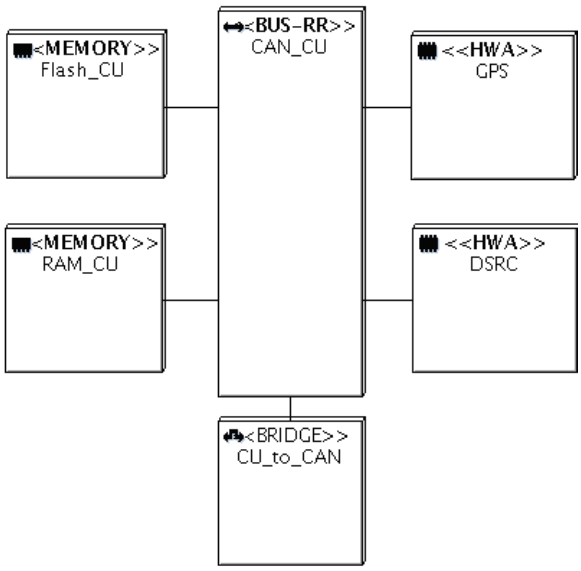


Figure 4.11: A partial view of the hardware architecture - Firmware flashing process

Note that, from a functional point of view, communication and storage look very similar (sending and writing could be considered as the same operation; receiving and reading too), but they are different: Communication takes place between different tasks while storage is dedicated to a single task, for its own needs. Moreover, reading is an action while receiving a message is an event: A task decides to read or not but has no control on messages reception, even if received messages can be ignored. Of course, when considering the physical view, it may be that communications are implemented through read and write operations in a memory and, in most cases, read and write operations of a task are implemented as transactions on a physical communication link between a processor and its external memories. Each of the three activities has a physical and functional counterpart, as outlined in Table 4.1.

System Activities	**Physical View**	**Functional (Logical) View**
Computing	CPU or dedicated hardware accelerator	Processing task
Communication	Wired bus or network, wireless link	Send/receive messages on logical channels
Storage	Memory (RAM, ROM, flash)	Read/write data from/to address spaces

Table 4.1: Physical and Functional Viewpoints

4.4.2.3 Mapping

A mapping process defines how application tasks are bound to execution entities and similarly how abstract communication channels between tasks are bound to

communication and storage devices. The general DIPLODOUC mapping framework is the following.

- Each abstract task of the application is mapped on exactly one computation node (i.e., CPU), which is a combination of both physical and functional view.
- Abstract communication entities are mapped on communication and storage nodes. A channel is usually mapped on buses, bridges and exactly one storage element [5].

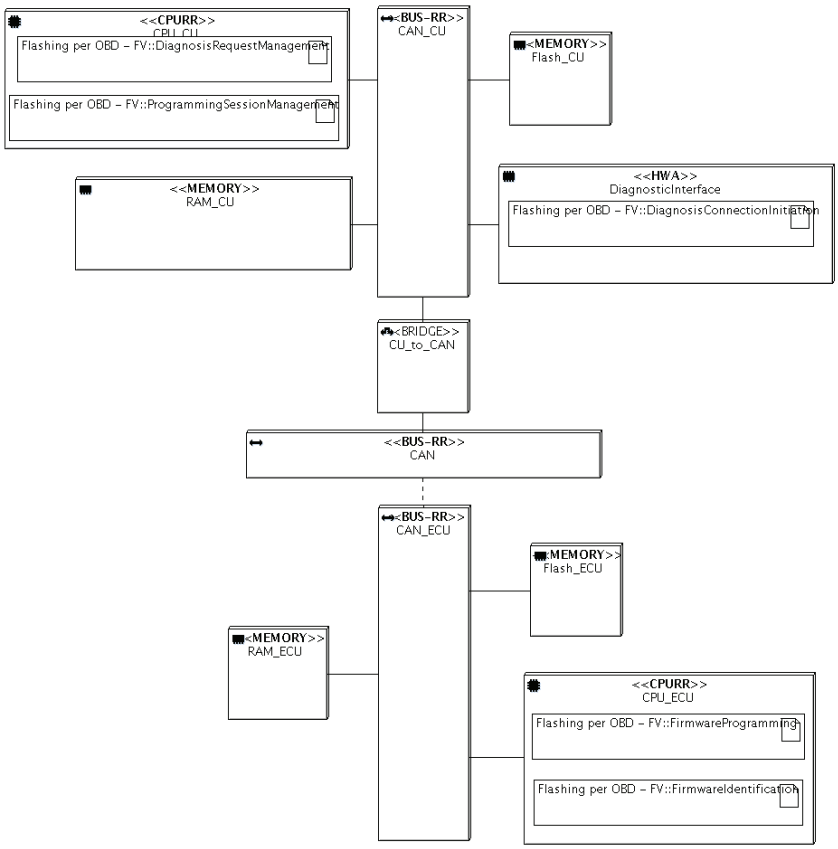


Figure 4.12: Mapping view - Firmware flashing process

The mapping activity is carried out based on previously created DIPLODOCUS architecture diagrams (Figure 4.10, and 4.11). The output of this activity is shown in Figure 4.12, where artifacts representing tasks and channels are simply bound to architecture components. Similarly like other activities, we also capture the knowledge about structural model and generate an equivalent ontological instance

of system architecture ontology. Figure 4.13, represents the knowledge about system assets and mapped functions on these assets. In particular, we have used the "system asset" class to store the knowledge about different system assets as well as knowledge about their subclasses (i.e., CPU, RAM, Hardware accelerators, etc.), and "function" class to capture different functions involved in the firmware flashing process. The "mappedTo" relationship class specified in the ontology captures the mapping relationship between system assets and function.

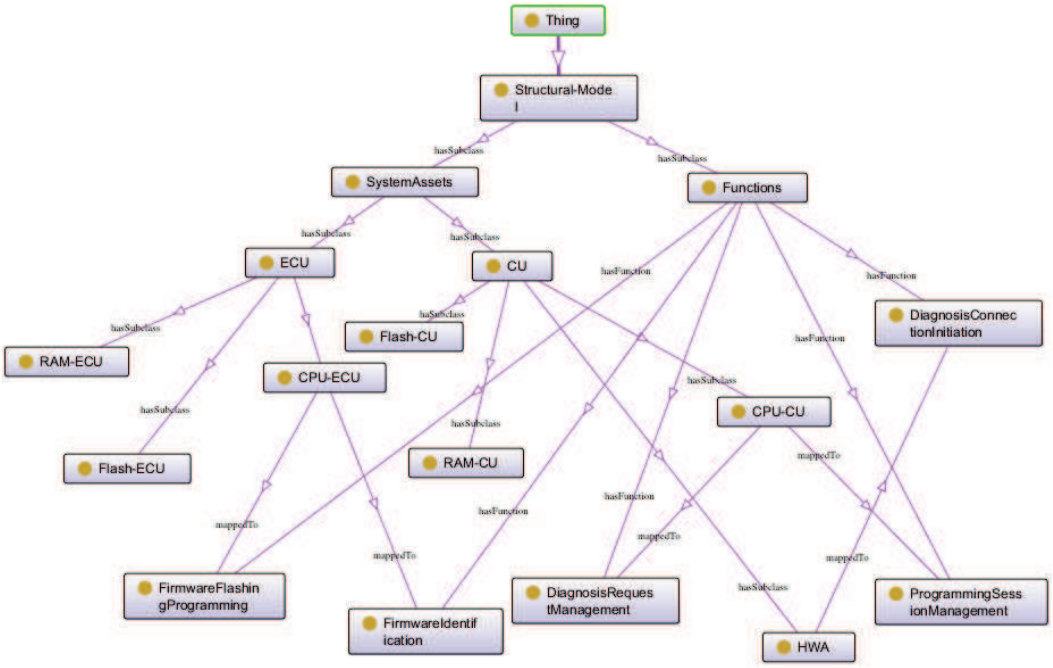


Figure 4.13: Ontological representation of the Mapping view of the Firmware flashing process

4.5 Conclusion

In summary, this chapter made the reader familiar with our running example, the different behavioral and structural models of the system, its methodology and operators. Furthermore, while creating system models (i.e., Use Cases, Sequence Diagrams, Structural Models, etc.), we also generated the corresponding ontology instances of these different models. Thus, allow security engineers and analyst to use these models and reason about different concepts and terms specified in these models. For instance, during SR identification activity, security engineer can query the use case knowledge base about different "actors" involved in the system, or he can search for different "functions" mapped on a particular system asset. Our objective for the next steps of the SREP is to use these models as well as the knowledge base generated from these models for performing the in-depth security analysis and also

identifying and prioritizing security requirements. Furthermore, we also use these models for building our cryptographic protocols and enforcement of access control related security requirements in Chapter 7 and Chapter 8, respectively.

Chapter 5

Security Analysis and Knowledge-Based Attack Trees

5.1 Introduction

The rapidly increasing need to integrate business applications deployed across distinct architecture layers reflects the reality of how software is being consumed nowadays [60]. Such applications must also be compliant with SRs and regulations, which can change and/or evolve according to the business context. For instance, access control and monitoring for intrusion detection are prime examples of functionalities that are subject to this problem: they cannot be properly modularized, that is, defined in well-separated modules, especially if they cross administrative or technological boundaries [58]. In such a context, it is not sufficient to discover security attacks only at overlooked weak point of the system; there is also a need to analyze the information flow control issues, especially when the underlying platforms and infrastructures are also made of services themselves. Security analysts also need to consider threats to these underlying infrastructure and middleware for a particular security realization, as the assets to be protected originate both from the horizontal (i.e., between different entities and components) and vertical (i.e., multiple layers) compositions.

A related problem is that it is easier to analyze the protection level at each separate layer in the system architecture stack, but become vulnerable to various security exploits and flaws in a coordinated manner [58, 129, 143]. Because of their complexity and of the varying degrees in which system assets are deployed and executed, it is often the case that a system is compromised through a path its developers never have thought of. What is worse, a local security attack and vulnerability or a mismatch between the security mechanisms adopted at different locations can have dire consequences, potentially putting the security of large system at stake. Most of such security attacks stem from the limited knowledge shared between various security-engineering activities that collaborate with each other and the expression of their interdependencies. One thing is that it is not easy to discover all parts of a system that are relevant for its security. In mainstream practice, this knowledge is often spread across different architecture layers, and correspond to various system development activities such as system architecture design, goal specification, In general, for a thorough security evaluation, one needs to take into account these dif-

ferent knowledge perspectives. In this context, in Section 5.2, we aim at proposing a security analysis model derived from the conceptual constructs of security ontologies that will serve as the common knowledge repository for discovering, analyzing, and sharing attack knowledge with other system development activities. Thus, it will offer means to analyze the security of the system in such a way that it is possible to discover simple and complex security attacks and vulnerabilities at different levels of system abstraction. Furthermore, the concept of attack tree, modeled in SysML Attack Tree Diagrams (ATD), is brought in as the foundational graphical representation for modeling and embedding the collected security attacks knowledge into the security attack ontology. In this manner, the attack trees are completely parameterized by the ontological concepts so that it is possible to handle simultaneously several knowledge bases associated with security attacks and vulnerabilities. In particular, the knowledge based attack trees ease the process of keeping security attack specifications clear and understandable, minimizing the inconsistencies and helping to achieve maintainability – even when security attacks are drafted cooperatively by several entities as well as at different system development stages.

5.2 Security Analysis Process

In this section, we reason about instance of the security attack ontology (cf. Section 2.4.2.3) to discover and share a common understanding of information about security attacks and vulnerabilities among different system development activities. In contrast to related research activities what we have in mind is to extract the knowledge and relationships between different security ontologies, for the purpose of being able to combine and analyze them together and discover security weaknesses. In our approach, we perform a security analysis in the following sense: given system development phase, we capture the knowledge about core (valuable) system components including hardware and software components, security goals, as well as security requirements, and analyze their security relative to security objectives and other functional and non-functional constructs, and we perform this analysis iteratively and incrementally at each system refinement stage. We thus, focus on high level security attacks (or anti goals) and relationships and evolve them in accordance with other system development activities, and in particular the level of refinement of the system architecture. In this context, it is also useful to think about threats in terms of what the adversary is trying to achieve and what are the adversary capabilities. This changes the focus from the identification of every specific attack – which is really just a means to an end – to focusing on the system-wide attacks. A security analysis process is composed of two parts: knowledge extraction and its evaluation, and a security attack modeling. We define the security analysis process as a systematic process performing the following steps:

1. Extract the knowledge from various knowledge bases such as security goals, system architecture, security requirements, and relationships among them

through inferences rules and questions on available material.

2. Define every "attack goal" that is associated with a benefit (cf. Section 2.4.2.3) to the adversary of some kind.

3. Identify classes of adversaries and their capabilities.

4. Decompose attack goals into a number of "attack methods" that could be employed to achieve the attack objective; refinement terminates when leaf conditions are reached that meet the adversary's capabilities.

5. For each attack method, specify different properties that as described in the attack ontology such as "attack mode", "attack type" (i.e., generic or application specific), assumptions, consequences, etc.

Let us illustrate these steps in more detail with our firmware flashing case study presented in Chapter 4. At the early stages of the system conceptualization, the major inputs to our security analysis process are security goal knowledge (see Section 4.3) and system architecture knowledge (see Section 4.4). However, during each refinement stage we extend the current knowledge as well as include the knowledge from other system development activities (e.g., security requirements, security protocols, etc.) as a basis for exploring more detailed security attacks and vulnerability. Let us start from the analysis of the security goals presented in Section 4.3. Based on the results of knowledge extraction phase (cf. Step 1), we move on to analyzing different "attack goals" that can be associated with these system architecture components. One obvious option is to browse the security goal knowledge base (see Section 4.3) systematically in order to determine whether there is any "AvoidGoal" that could be wished by malicious agents. In this case, we use the following rule to select "AvoidGoals" from the security goal knowledge base:

$$Goal(?g) \wedge hasAvoidGoal(?g, ?AvoidGoal) \rightarrow select(?g, ?AvoidGoal) \quad (5.1)$$

The above query retrieves (cf. reasoning with SysML model presented in section 3.4.2) all goals in ontology with a known subclass that is AvoidGoal as shown in the Figure 5.1. In this case, we start analyze each AvoidGoal and determine how an adversary can attack this goal, which in his (adversary) case is an AchieveGoal. For example, an adversary can "abort firmware flashing process" by jamming the in-car communication or by shutting down the Communication Unit (CU) [129]. We analyze each AvoidGoal to identify the adversary objectives as well as determine additional AvoidGoals, if any. In addition, while browsing the security goal knowledge base, we might stop on the "AchieveGoal" stating, "service station is allowed to install firmware". This goal is obviously going to be of interest for a number of adversaries. For instance, an adversary goal might be to "install malicious firmware". We can also directly determine security flaws and weaknesses by negating the "AchieveGoals". For instance, the statement "service station is not able to install firmware", actually corresponds to a Denial of Service (DoS) attack.

Figure 5.1: Avoid goals - Firmware flashing process

In a similar way, we extract the knowledge about system architecture models and analyze different aspects of the system in order to determine additional attack goals and attack points. For instance, we can extract the knowledge (see Rule 5.2) stored in use cases knowledge base (see Figure 4.5) to define the behavior not wanted in the system to be developed. This corresponds to the misuse cases approach [141], where analyzing the interaction between actors and cases specifies attacks. For example, an adversary's goal might be to cease the firmware diagnosis process by sending fake information about the version of the installed firmware to the DT. Similarly, we determine various attack goals by reasoning on instances of these different knowledge bases.

$$UseCase(?u) \wedge hasActor(?u, ?Actor) \wedge hasCases(?u, ?Cases) \rightarrow select(?u, ?Actor, ?Cases) \tag{5.2}$$

Once initial attack goals are identified, we start building the attack tree in SysML Attack Tree diagram and group these different "attack goals" under the root node called "attack objective". This is illustrated in Figure 5.2, where the root node (e.g., "Attack Firmware Flashing") depicts the attack objective and its child nodes (e.g., "install malicious firmware"," service station is not able to install firmware") represent attack goals. The identification of adversary instances is obviously intertwined with the identification of attack goals. In particular, the attack goal raises the question of who might profit from it. In this case, we can use the adversary taxonomy presented in section 2.4.2.3 to identify different the adversaries and their capabilities. For example, the following query retrieves all adversaries with the capabilities that can a achieve a particular "attack objective":

$$\begin{aligned} &Adversary(?e) \wedge hasExperties(?e, ?expt) \\ \wedge\, &hasEquipment(?e, ?equp) \wedge hasKnowledge(?e, ?kwlg) \\ &\wedge Asset(?a) \rightarrow select(?e, ?expt, ?equp, ?kwlg, ?a) \end{aligned} \tag{5.3}$$

However, in many cases these different capabilities are not independent, but may be combined/substituted for each other in varying degrees. For instance, expertise or equipment may be a substitute for time. In this example, we assume an expert adversary who knows everything about the firmware flashing process as well as have

knowledge about in-vehicle system being attacked. In the next step (step 4), for each initial attack goal and adversary class identified, we build the attack tree by decomposing attack goals into a number of attack methods that could be employed to achieve the attack objective. As mentioned previously, our objective is to identify security attacks and vulnerabilities that are targeting single or multiple architecture layers. In this case, we follow our functional path (cf. Section 4.4.2.1), and Mapping approach (cf. Section 4.4.2.3) to analyze security attack and vulnerabilities. In particular, we extract (Rule 5.4) the knowledge from System architecture knowledge base (see Figure 4.13) about different functions, their mapping on the different system assets, and details about information flow path in order to analyze how an adversary can perform attacks to achieve his attack goals.

$$\begin{aligned} &Architecture(?e) \ \wedge \ hasFunctions(?e, \ ?Funtions) \\ \wedge\, &hasSequence(?Functions, ?Sequence) \wedge \ hasAssets(?e, \ ?Assets) \\ &\wedge \ mappedTo(?Function, ?Assets, ?Mapping) \\ &\rightarrow select(?e, ?Functions, ?Assets, ?Sequence, ?Mapping) \end{aligned} \tag{5.4}$$

The result of this query corresponds to a set of functions (e.g., diagnosis initiation, firmware identification, etc.), information flow path in terms of sequences (e.g., msg1, msg2, etc.), and their mapping on different system assets (e.g., CU, ECU) as shown in figure 4.13. Based on the result of this query, we start analyzing the available information and different attack methods that correspond to attack goals. For example, the above-mentioned attack goal (e.g., "Install Malicious Firmware") can be require a "Man in the Middle" attack. As a result of attack identification that corresponds to the attack goal, we add attack node (see attack tree node – AT.2.a in figure 5.2) as a child of the attack goal. In addition, following our knowledge based attack tree modeling principle (presented in section 3.3.4), the security expert has to document the attacks using about different ontology concepts (e.g., attack type, attack method, attack mode, adversary, etc.). This information is available in SysML Attack Tree diagrams as a controlled vocabulary (see Section 3.4.1). Let us consider another case where an adversary is trying to achieve the same attack goal ("Install Malicious Firmware") by "Injection of forged transactions" (attack tree node – AT.2.b). In this case, we use the definition of constraint block (cf. section 3.3.4) and link these two attack methods using "OR" operator as shown in figure 5.2. Thanks to the expressive power of ontologies, we can freely combine rules as antecedent patterns to capture complex topological structure, and analyze different system activities altogether in order to identify different attack methods. A description of various attack methods that we identified by analyzing different knowledge bases and interrelation between them are listed in Section 5.3. We can also decompose these attack goals by querying different well-know attack and vulnerability dictionaries (e.g., OWASP, CVE, etc.). However, as indicated previously, for a thorough security evaluation and validation, we require an explicit

knowledge of security experts to decompose attack goals into child attack methods. The security analysis process terminates when realistic attack methods are obtained that meet the adversary's capabilities.

5.3 Attacks on the Firmware Flashing Process

In the following, we use the above-mentioned security analysis process and analyze the attacks against our three core activities of system architecture (see Table 4.1) that we have defined for our firmware flashing process. In particular, we first analyze the system from the active and passive attack viewpoints, and then classify security attacks.

- **Attacks Related to Active Mode:** One of the ways to attack the firmware update process is to modify its behavior. In the following, we show security attacks that are associated with modifying the behavior of different system assets and corresponding activities involved in the firmware update process.
  - Communication: The attacks on the communication activity can be broken down into threats on physical and functional (logical) communication links that are used during the firmware update process. There are two main means to implement physical[1] attacks against communications: tampering with it and the injection of forged transactions (see attack tree node – AT.2.a in figure 5.2). Because on-board computing devices and memories are usually connected through buses, attacks against physical communication links can be used to tamper with the communication. The consequences of physical attacks on communication links are on the receiver side only (attacks aiming at modifying or canceling a firmware update message before it is actually sent are, in fact, attacks against the sending computing node). From the functional (logical) point of view, attacks comprise Denial of Service (DOS) attack (see Attack Tree node – AT.5.a). For instance, one way to do this would be attack the wireless communication module by jamming the signal (see attack tree node – AT.5.a.3.a). As we previously mentioned (cf. section 4.4.2), an architectural description is a collection of interconnected layers. Thus, the security attacks targeted for one particular layer may span across other layers and system activities. For instance, when a memory bus is attacked, it can be in order to modify the function of a task (software code modification) or the data it processes. There are three classes of memory bus injection attacks (see attack tree node – AT.2.c.2): spoofing (the injected information was forged by the attacker), splicing (the injected

[1]The text in bold represents different classes and subclasses of the security attack ontology as well as relationships between these classes. While, the text in italic characterizes the concrete attack methods.

information was taken at a different location in the memory) and replay (the injected information was taken at the same location in the memory but at a previous moment in time, where it differed from the expected one).

- Computation: The computational capabilities of an attacker encompass several different abilities such as decrypting incoming messages, encrypting outgoing messages and computing secrets. There are two components that affect these abilities: hardware capabilities (pure computation) and available information. In particular, communication attacks are targeting computing nodes (CPUs, hardware accelerators) that are involved in the firmware update process (see figure 4.11). They consist in physical modifications of the component (like modifying the content of an embedded ROM or the structure of an operator), its replacement or even its destruction. Transient fault injection is another possibility (see attack tree node – AT.1.c.a). The consequence is the production of results that differ in some way from those that would have been produced in normal operation, including failure to produce results when expected or the converse. From the functional point of view, these physical attacks can translate into: it seems easier (and more likely) to attack the on-board units with a DoS attack (see attack tree node – AT.5.a) to prevent or delay the computation/detection of events needed for the firmware flashing process.
- Storage: Storage attacks consist in modifying the regular content of a memory. As a consequence the read operations performed by the tasks accessing the address space do not return the expected information, that is, the last one that was written at the same location. The consequences are very similar to the consequences of attacks against memory buses. The means used to achieve content modification depend on the technology: ROMs can be replaced (see attack tree node – AT.3.b.1.a), non-volatile writable memories (EEPROMs, flashes) can be replaced or reprogrammed, volatile memories (static and dynamic RAMs) are much more difficult to attack in a conscious way but more or less random bit flips can be induced by voltage, clock frequency, temperature modifications, or more active fault attacks. In some cases, volatile memories can even be cooled, removed from their PCB and plugged onto another host without losing their content which can then be read out and/or modified before the component is plugged back in its regular host system.

- **Attacks Related to Passive Mode:** In the following, we illustrate security attacks aiming at information retrieval without modifying the behavior of the firmware update process.
  - Communication Communication can be spied upon and sensitive messages or read/written data exposed. On-board or on-bus probing (see

attack tree node – AT.2.c.2) is a very effective and attractive mean for wired communications. Wireless communications are even more sensitive to this kind of attack as they can be conducted in a completely remote and undetectable way. On-chip probing requires package removal, expensive equipment and very skilled attackers. Another possible attack would consist in trying to gain access to the on-board units, for example by personalizing the car with an external device, without changing the behavior of the flashing process or vehicle, to exploit rekeying protocol vulnerability in the diagnostic interface (see attack tree node – AT.4.e), to extract the firmware data.

– Computation: Attacks against the computing activity aim at retrieving either a secret quantum of data (secret key) or the processing definition itself through software code extraction (see attack tree node – AT.3.b). As every computation is actually performed by a physical device, measurable syndromes are produced, like its power consumption, computing time, or electromagnetic emissions that can be exploited to guess what operations are performed or what is the value of some sensitive data. This kind of analysis is referred to as side channel attacks in the literature. Observing the external communication or the exchanges (see attack tree node – AT3.b.1.b) with memories is another mean to get information about the computing but fall into the passive communication attacks category.

From a security point of view they are all potential targets of attacks but by different means and consequences. Security requirements shall therefore address all system elements that might be involved in attacks identified in attack trees.

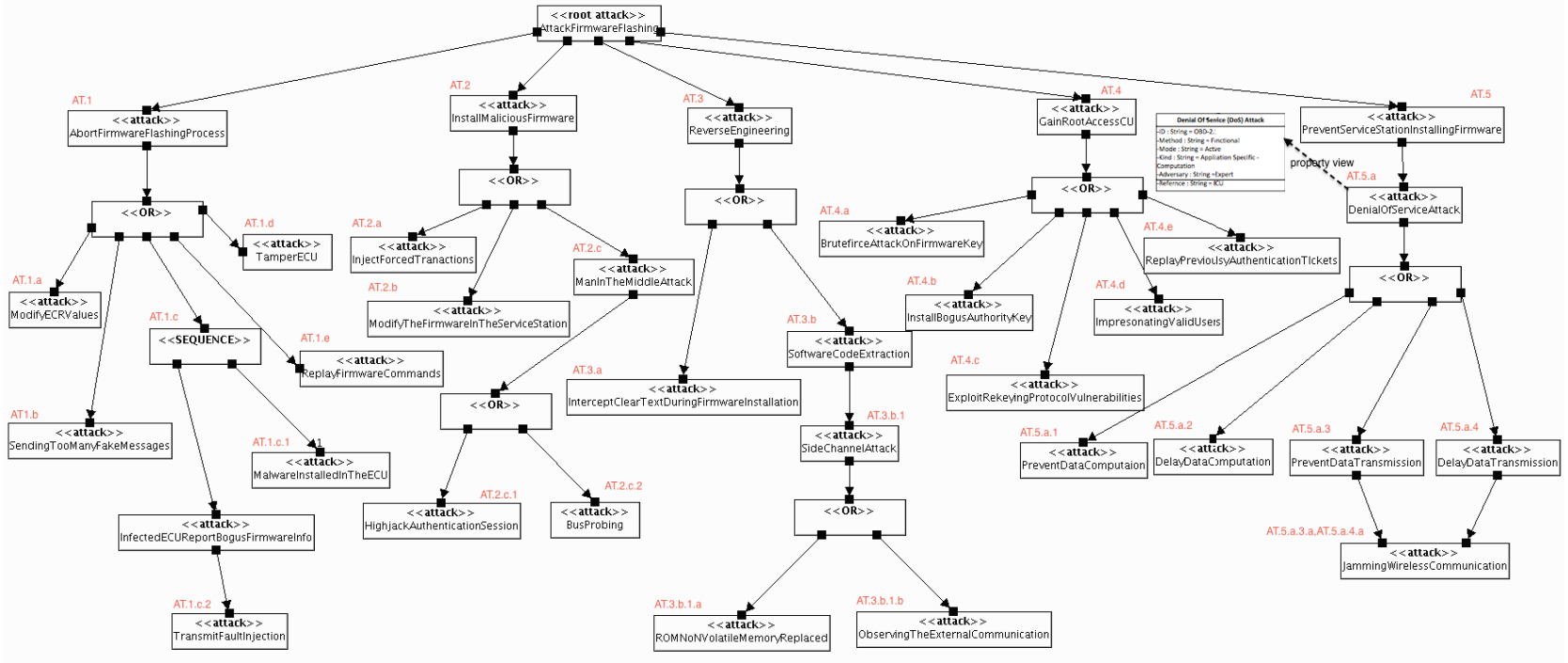


Figure 5.2: Attacks on the Firmware flashing process – Knowledge base Attack Trees representation

5.4 Multilayer Security Analysis

This section describes an example of an actual exploitation that we have identified during the multilayer security analysis. As we previously mentioned, the loose coupling between different architecture layers leads to various security exploits and flaws in a coordinated manner. This was the case in our example, where a loose security binding between architecture layers (i.e., Hardware Security Module, middleware – EMVY RPC Library [125], application, etc.) makes it possible for an adversary to impersonate valid users[2]. In particular, the security attack that resulted was due to the improper design of the middleware layer, EMVY RPC Library, and it's handing of security related data. EMVY RPC library allows applications to use functionality on the client itself and also to access higher-level security functionality (i.e., install applications, set security policies, etc.) through the Master node as part of the RPC invocation.

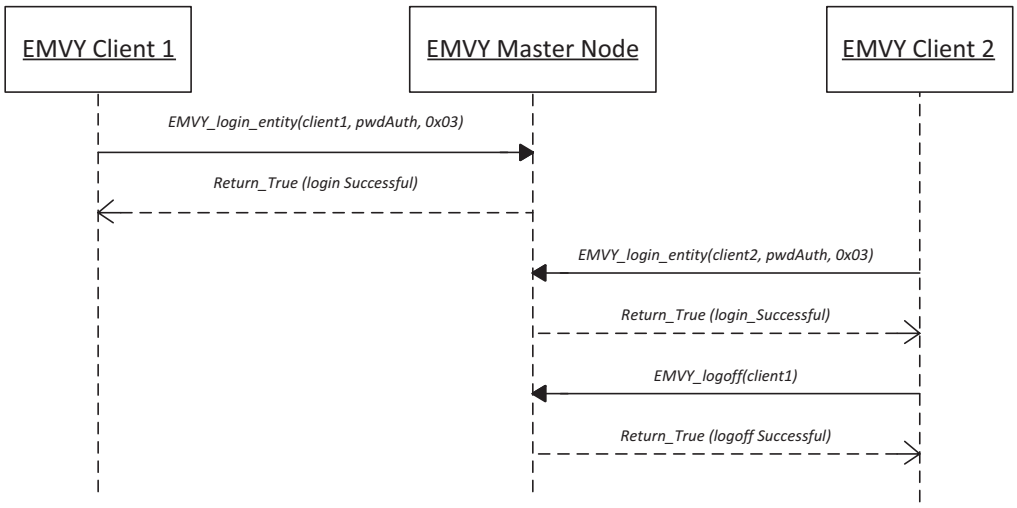


Figure 5.3: RPC Logoff attack scenario

In order to perform this attack (impersonate valid users), we have created two EMVY clients: Client 1 and Client 2. The Client 1 is considered as a legitimate entity, whereas, the second entity, Client 2, act as a malicious entity. In the first step, both entities send a login request (see figure 5.3) to EMVY Master node in order to invoke several other security services. The authenticity for the connection is verified in the Master node, by calling its hardware security module, which verifies the authenticity of the Client 1 and the Client 2 at the transport layer. Once the authenticity of both entities is successfully verified both entities are allowed to

[2]User impersonation allows an entity/application to execute a task using the security context of another user.

invoke any RPC functions. In the next step, Client 2, who is an insider adversary, scan the connected entities in the network to extract their identifiers. Lets assume that Client 2 managed to capture an identifier (i.e., Client1) of the Client 1. It then invokes an RPC logoff function (see Listing 5.1), which only requires Entity as a parameter. An adversary uses the identifier of Client 1 as a parameter of the function in order to stop all the services accessed by the Client 1. The transcript in figure 5.1 shows the use of the RPC Entity_logoff(client1) request from Client 2 to EMVY Master node in order to logoff Client 1.

```
1 EMVY_logoff(const Entity* entity);
```

Listing 5.1: EMVY Log_off RPC function

On the Master side, it only verifies that Client 1 is in its "Entity Authentication List". If its so, it removes the Client 1 from "Entity Authentication List" and close the connection with the Client 1, thereby allowing the adversary to impersonate the valid user. This attack happens since authentication is only performed at the transport layer and not further considered at the upper layers. More precisely, the need to authorize operations based on RPCs together with the fact that only channels, not RPC messages are authenticated has forced us to piggyback the transport-level authentication on internal framework calls from the components like Communication Control Module[3] (CCM) to the application layer.

5.5 Conclusions

Given an input for our knowledge centric design methodology, the security analysis process helps to both classify identified attacks, but also to think about new ones, given a category. Security analysis process is a combination of both top-down and bottom-up approach to provide a support tool to security analysts. The purpose of developing the ontology driven security analysis process is to identify possible security threats and to allow aspects such as the desirability (to the adversary), opportunity, probability and severity of attacks to be assessed in order to share knowledge among various system development activities (i.e., security requirements engineering, protocol design, testing, etc.). We believe that, on the one hand, ontology based security analysis is expressive enough to describe several real-world security attacks with a multi faceted approach; at the same time, it provides constructs to map and relate security attacks with other system development activities.

[3]The central communication module provides a high-level interface for secured communication. It can integrate various communication protocols.

Chapter 6

Security Requirement Engineering

6.1 Introduction

We described the main building blocks of our SRE process in the previous chapters. We now move on to the next stage of this process and illustrate the approach in the context of SRs identification, refinement, and present a way to trace SRs. We first discuss in section 6.2, why security requirements should not be considered independently from the architecture of the system they relate to, or from the threats and vulnerabilities that may arise on that system. We highlight how the strengths of ontological approach can be used to drive the security requirement identification process. In section 6.3, we investigate a fundamental flaw in state of the art approaches to security requirements refinement. We expose in this chapter in what respect the different security artifacts (i.e., security attacks, system architecture), and its evolution involve challenging refinement problems, in particular with respect to the understanding of security requirements.

Security requirements refinement cannot rely on only high-level definition of security goals or preliminarily constructed SR specification and disregard the evolution of other security artifacts on top of which security requirements are based. In this perspective, we first illustrate why security requirements can be refined with enough precision for supporting the design of security architecture only if they are extensively linked with security attacks and the system architecture. All existing solutions have so far fallen short in that they only consider refinement separately within the SR requirement model and/or within the attack model. We present a refinement model to combine and jointly annotate all security artifacts and how it can be used to develop an iterative refinement process.

In section 6.4, we also propose a very simple but fairly effective approach for the traceability of security requirements. In particular, we provide insights on how a traceability links empowered with cross-reference capabilities can provide clean modularization to security assurance. We show how cross-reference traceability links help in achieving the impact analysis of prescribe changes in the different system development activities, by improving the efficient tracking and management of security requirements.

6.2 Security Requirements Elicitation

In this section, we aim at identifying security requirements in relation to the aforementioned system development activities. In the process of going from steps 2 to 5 of requirements engineering process (see section 2.4.1), we make use of the knowledge base, which we developed, in the previous chapters. Let us recapitulate what we have defined so far.

- In step 2, we have specified several security goals (see section 4.3) as key operational capabilities of the system specified by the stakeholders or determined from the security policy of the organization. For example, an achieve goal (see figure 4.3) "only valid users are allowed to access firmware flashing functions" is specified by analyzing the firmware flashing specification. As a result of this activity, we accumulated a goal knowledge base and hold the relationships between ontology related classes and subclasses, as illustrated in figure 4.3.

- In step 3 of the SREP, we presented the system architecture models (in section 4.4) for building the behavioral and structural models for firmware flashing application, during the early stages of system development. This step of the SREP is performed incrementally, by iterative analysis of the functional models and of the mapping view. Figure 4.4, 4.7, and 4.12 gives an overall view and results of how the step 3 is applied in the context of identifying system assets and their behavior. Furthermore, we instantiated the system architecture ontology to build a knowledge base about structural and behavioral models (see figure 4.3, 4.5, 4.8, and 4.13) that are constructed during the system architecture modeling.

- In step 4 of the SREP, we have used the results, in the form of knowledge base, from step 2 and step 3, and performed a complete assessment and security analysis of system assets and goals in chapter 5. It involves a security analysis process (presented in section 5.2), that is, a dual model of threats to the system model: it shows how system assets can be attacked within multilayered system perspective and how this attack knowledge (i.e., attack knowledge base) can be shared with other system development activities. As a result of this activity, we accumulated knowledge about potential malicious activities into an attack knowledge base in the form of a SysML Attack Tree Diagram (ATD) (see figure 5.2). We highlighted the different threats in terms of what the adversary is trying to achieve (i.e., attack type, attack method, etc.) and what are the adversary capabilities.

- Based on the results of the above steps, we have performed a risk analysis in section B.1.1. In particular, we have used the risk model [129] developed in the EVITA project [117], which indicate the risk level (see Table B.4), based on the potentiality (equal to severity level + likelihood/probability) and the maximum impact level on the concerned system assets.

After gathering knowledge about different system development activities, we now show how these knowledge bases can be adopted in SR identification step. Following the line of reasoning suggested in section 2.4.1, we start the requirement identification process by querying and applying inferences rules on the already developed knowledge bases of different security classes. For instance, let us turn back to the case of identifying SRs from the security goal and let us show how our model can integrate capabilities of goal-oriented [38] approaches to identify SRs. In this case, we use the following types of rule to select suitable security goals from the goal knowledge base (cf. section 4.3):

$$Goal(?g) \ \wedge \ hasAchieveGoal(?g, \ ?AchieveGoal) \\ \rightarrow select(?g, \ ?AchieveGoal) \tag{6.1}$$

This query will return goals (i.e., "only valid users are allowed to access firmware flashing functions") with a known ontology class type that is an "AchieveGoal". Thus, based on the result of query, we analyze goals and derive SRs. For instance, "only valid users are allowed to access firmware flashing functions" goal can be refined into "ensure authenticity" security requirement as shown in figure 6.1. In addition, queries can also operate in conjunction with rules from other security classes (i.e., system architecture, use cases, etc.) and can be used to identify SRs inferred by those rules. This for instance is the case of model-driven [54, 88, 151] engineering approaches where security requirements are identified in accordance with goals and system architecture. Consider for example that during the system architecture design, system architect specified the "AvoidGoal": "avoid service station from gaining root access of CU" (see figure 4.3) in order to protect the CU. Thus during the SR identification phase, we can use the following rule (equation 6.2) to retrieve a set of "AvoidGoals" and their relationships with specific component of the system architecture.

$$\begin{aligned} Goal(?g) \ \wedge \ hasAvoidGoal(?g, \ ?AvoidGoal) \ \wedge \\ Architecture(?a) \ \wedge hasComponent(?a, \ ?component) \wedge \\ Component(?c) \ \wedge hasHardware(?c, \ Hardware) \ \rightarrow select(?g, \ ?c) \end{aligned} \tag{6.2}$$

Based on the result of this query, we decide what kind of SR is required to avoid service station from gaining root access of the CU. As you can see from the query that we also extract the knowledge about the system architecture like system component, and its subclasses (i.e., type of system asset, in/out parameters, functions, etc.) in order to analyze different properties of systems. In particular, the object of this query is to extract the knowledge from two different knowledge bases (i.e., goal and system architecture), analyze them together and define a security requirement. In this case, for example, we can specify "controlled access rights to CU" requirement to restrict the rights of service station to access different functionaries of the CU, as shown in figure 6.4. We can further identify SR by analyzing the security attacks on the system assets and goals. For example, while browsing the attack knowledge

(using Rule 6.3) about firmware flashing application we might stop on the attack that corresponds to our above mentioned example, where an adversary (i.e., service station) gain root access of CU by "installing bogus authority keys" (see attack tree node – AT 4.a in figure 5.2).

$$\begin{aligned} &Attack(?a) \ \wedge \ hasActiveAttack(?a, \ ?Active) \\ \wedge\, &hasFunctionalAttack(a?, ?Functional) \ \ \wedge Asset(?CU) \\ &\rightarrow select(?a, ?Active, ?Functional, ?CU) \end{aligned} \tag{6.3}$$

In this case, we can specify "restrict access rights to install keys in the vehicle" security requirement as shown in figure 6.4. The purpose of this SR is to prevent service station to install any kind of authority key in the vehicle, which might allow him to gain access to different parts of the on-board architecture. In a similar way, we can go through each ontology class, as well as a combination of different relationships in order to infer and derive SRs. In the next section, we model these identified SRs in the SysML SRs diagram, and enrich them with ontological details (cf. SR ontology presented in section 2.4.2.4), such as type of the requirement, its kind, etc.

6.2.1 Security Requirement Modeling

We now illustrate the modeling of security requirements in the SysML SRD and show how we can use ontological concepts. We will continue our discussion using the firmware flashing example from the previous chapters. Using the two main SR types (i.e., FSR, NFSR) introduced in the section 2.4.2.4, the first subsection is dedicated to Functional Security Requirements. Next, we present the Non-Functional Security Requirements that are also necessary to secure the firmware flashing process.

6.2.1.1 Functional Security Requirements – FSR

This section describes the security requirements that a system or system component has to ensure during the firmware flashing process.

- **Ensure authenticity:** This security requirement specifies that the security features of a firmware flashing application shall require each entity (i.e., internal or external) to be successfully authenticated before allowing any firmware update action to be executed. This involves: ensure the identity of a service station, and firmware authentication. We use the SysML «containment» relationship and group all these requirements together under the abstract security requirement "ensure authenticity" in order to organize requirements in well-formed structure, as shown in figure 6.1. From the ontological point of view, we analyze each security requirement and map them to appropriate ontological classes. For instance, "ensure the identity of a service

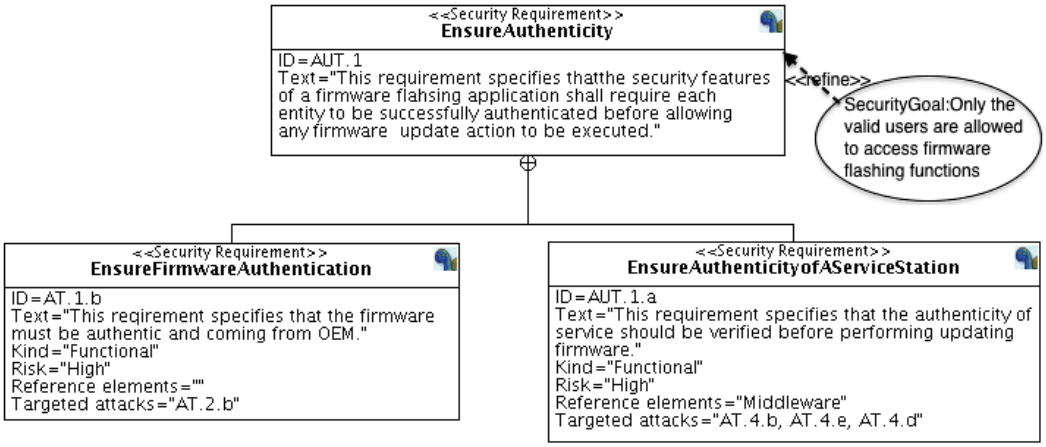


Figure 6.1: Authenticity security requirements

station" SR is a specific requirement defined for the firmware flashing application. During modeling this requirement we classify this requirement as a "Domain Specific" SR, and specify this in SR diagram by selecting the tagged value of the "Classification" property as Shown in the Figure 6.1. In a similar way, we also specify the type of SR by selecting an appropriate value from its list of tagged values. For instance, in the case of authenticity security requirement, we select the "User Identification and Authentication" as a "Type" of SR. In addition, following our reference concept, introduced in section 3.3.3, a reference to the attack tree node (i.e., see attack tree node – AT.4 in figure 5.2), a reference to the system assets (i.e., Middleware), and a value about the risk level (i.e., High) is also computed for each SR and specified in the SysML SR element.

- **Ensure Integrity:** This security requirement should of course be monitored to check whether a message sent between service station and on-board components is unaltered, but also with respect to guarantee that the content of a storage facility are not modified between two given read operations, or even to ensure that the execution of the software implementing a service is not being attacked through a modification to the execution environment or the code it runs. Figure 6.2 depicts an aggregation of these security requirements using the SysML «containment» relationship. As before, we analyze each ontological concept for each of these requirements and map them to appropriate ontology terms and concepts defined as a controlled vocabulary.

- **Ensure Freshness**: This security requirement prevents an adversary from performing replay attacks in which a valid data is maliciously or fraudulently repeated or delayed. In this context, this requirement is defined to ensure that

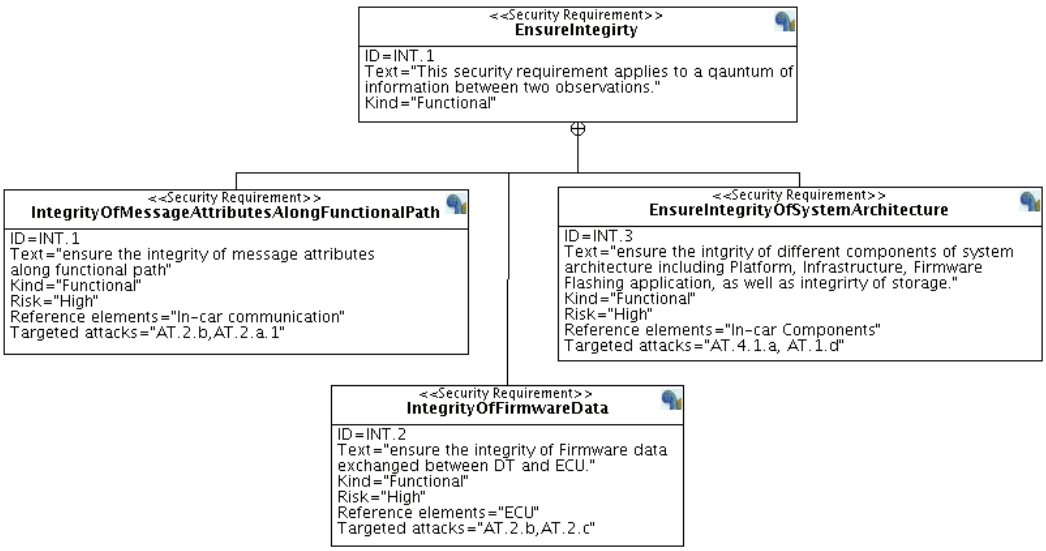


Figure 6.2: Integrity security requirements

all the messages and data exchanged between entities (i.e., DT, CU, ECU, etc.) fulfill the freshness property. Even so, we also need to ensure the freshness of firmware data and the freshness of all the messages along functional path. Besides this, it is also imperative to ensure the freshness of flashing commands sent from diagnosis tool. Figure 6.3 depicts the SysML representation of these requirements along with mapping of ontological terms and concepts.

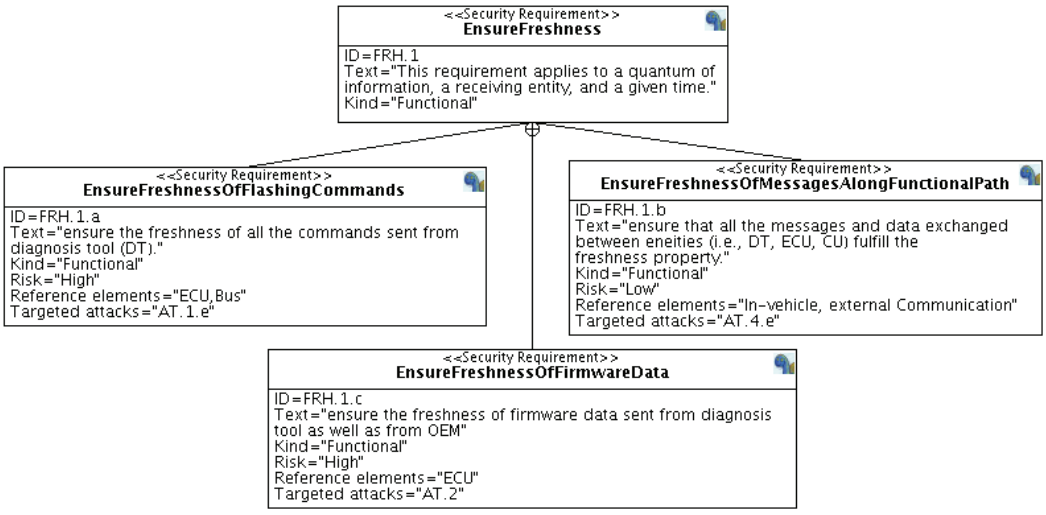


Figure 6.3: Freshness security requirements

- **Ensure Authorization:** This security requirement is specified to prevent the gathering of unauthorized access rights to the resources. During the firmware

update process, it is required that the system uses the access control rules to decide whether access requests from the (authenticated) service station, using the diagnosis tool, for an installation of the firmware shall be approved or disapproved. Moreover, it is also required to restrict the access rights for accessing the flash memory, reading from the flash, and also limiting the access rights to firmware update functions, or restricting access to install authority keys as shown in figure 6.4.

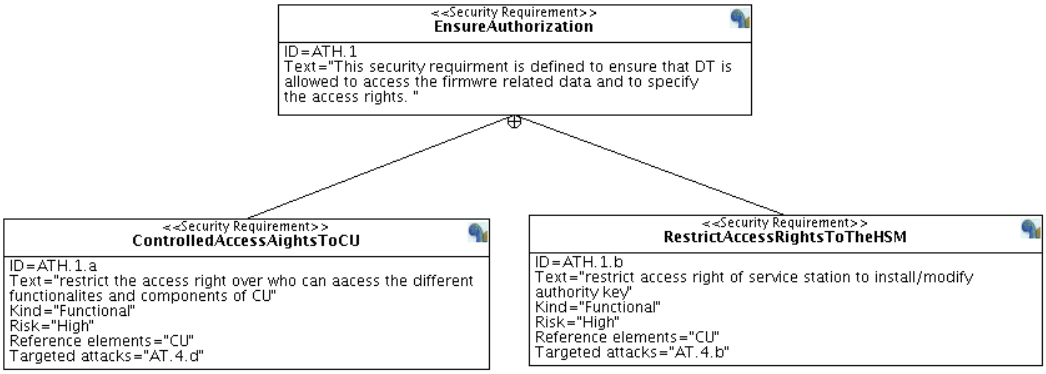


Figure 6.4: Authorization security requirements

- **Ensure Confidentiality**: This security requirement is defined to ensure that the authorized entities are the only ones that can know any secret of information (i.e., firmware data, firmware shared keys, etc.). In the firmware update scenario, this requirement is mainly specified in order to prevent an adversary from accessing the firmware code, analyzing its structure, function, and injecting his own code in the original firmware flashing code. Figure 6.5 demonstrates the SysML representation of Confidentiality requirement.

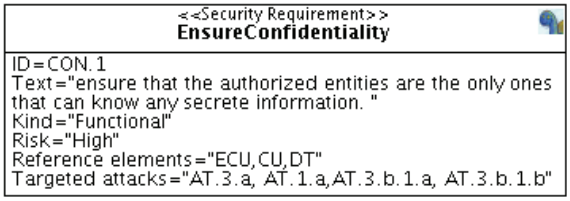


Figure 6.5: Confidentiality security requirements

- **Ensure Availability**: This security requirement is focusing on properties that should be maintained despite denial of service attacks, coming either under the form of computational resource oriented DoS, network DoS, or even degradation of real-time constraints. In this context, an availability requirement applies to a service provided by the ECU, or to platform running on the ECU, or to physical components of the ECU (i.e., CPU, RAM, or Bus) providing a service. This requirement is satisfied when some service is opera-

tional during operational periods. It is further detailed with the specification of the period during which the availability is required and of a set of entities requesting the availability. Figure 6.8 depicts the SysML representation of availability requirements mapping of ontological terms and concepts.

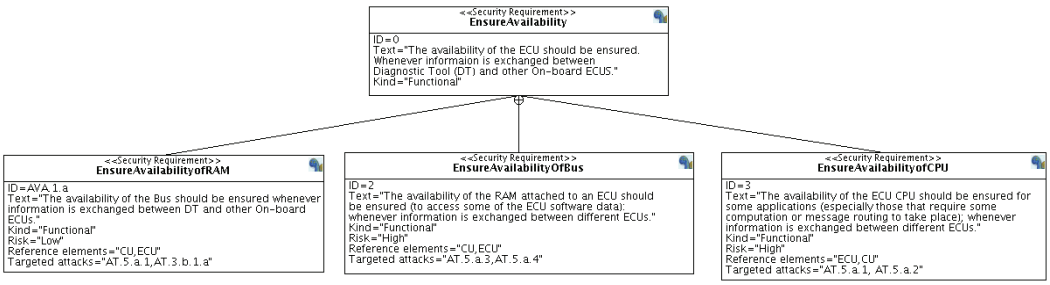


Figure 6.6: Availability security requirements

6.2.1.2 Non-Functional Security Requirements – NFSR

The previous section showed how to define functional security requirements, their classification. In this section, we present non-functional security requirements, which are also essential to ensure the security of the firmware update process.

- **Monitor the Network Traffic:** This security requirement is focusing on properties that the system should monitor about network or system activities, or policy violations. It should then trigger alerts on detecting unusual behavior, during the firmware update process.

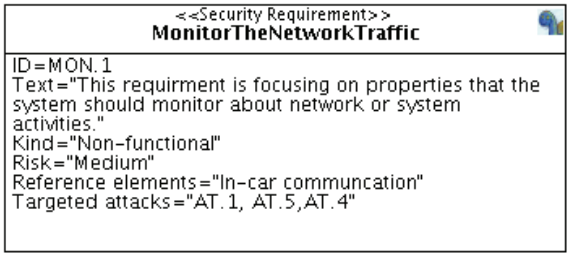


Figure 6.7: Monitor the network traffic security requirements

- **Prevent Structural Weakness of the Firmware Keys:** This security requirement is defined to ensure that the keys to be used within the firmware update process have a well-defined structure. They should therefore be large enough so that a brute force attack (possible against any encryption algorithm) is infeasible – i.e., would take too long to execute. As the security of the firmware code is solely based on the strength of cryptographic keys and of

the encryption algorithm, the difficulty for an adversary to obtain the key determines the security of the firmware update process.

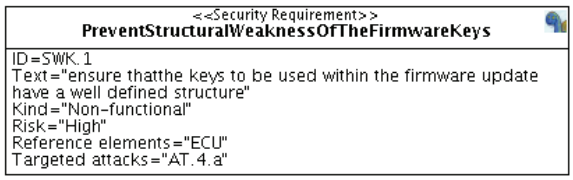


Figure 6.8: Prevent structural weakness of the firmware keys security requirements

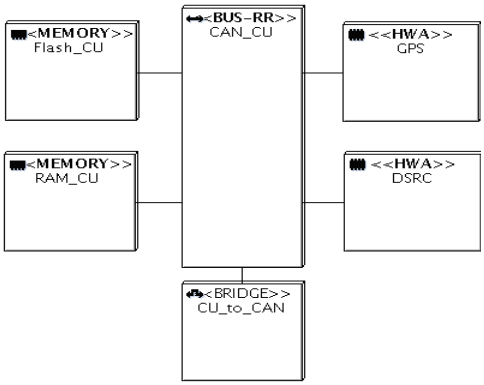
6.3 Security Requirements Refinement

Although the refinement relationship is already defined and explored in aforementioned approaches for an explicit iteration of security requirements, an important amount of diversification of this relationship still remains hidden and underspecified. For example, those approaches generally lack any basic support for understanding significant changes from the perspective of dependency relationships between different artifacts (or security classes in our definition). In this perspective of an early introduction of security concerns in the design of a system, security requirements and goals are often originated from the functional behavior of the system. More precisely, security requirements originate from the system functional specification, and in particular from the system architecture artifacts themselves, as well as from the threats identified on those components: in this respect, the SRE model should also take into account the relationships between SRs and the specific context such as attacks, goal, and architecture, that prompted their expression, which should become increasingly detailed through refinement. The latter point of view is for instance supported by approaches like ISO-15408 ([62], sec. 6.2), which links security requirements with system assets. The next section describes in more detail the different dimensions of the refinement of security requirements in the vision outlined above.

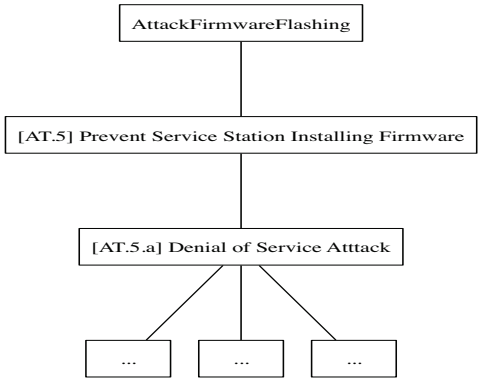
6.3.1 What a SR Refinement is not ...

In this section, we investigate how a fundamental flaw in state of the art approaches to SRE limits the fulfillment of the refinement vision sketched above. A first step towards this evaluation consists in analyzing the various relationships among different artifacts that are involved in the security requirement identification process. We can find various relationships and associations used in the literature, like the dependency relationship [88, 93, 87, 47, 57, 141] or the composition relationship [69, 153, 57]. However, as mentioned in the literature review section 2.2, the impact of these relationships on the iteration process is fairly neglected. With regards to iteration

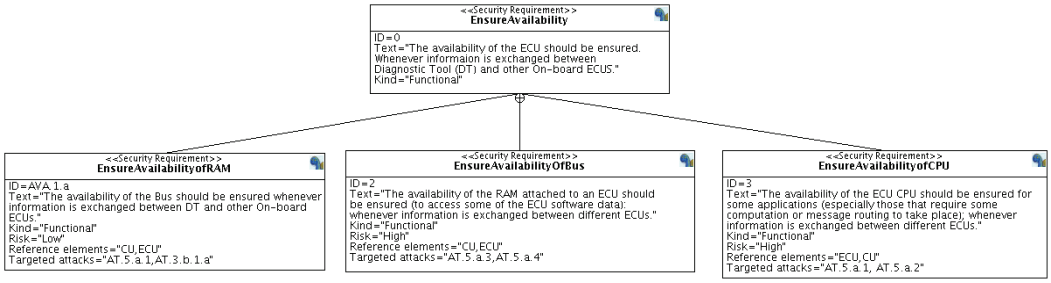
and relationship binding, each refinement step may have different implications on the security requirement (or set of SRs). As an illustration to SR refinement, we consider the dependency relationship and consider this relationship on our running example (section 6.2). Figure 6.9, recaps the initial results obtained from the SREP.



(a) Partial view of the system architecture



(b) Partial view of attack tree



(c) Availability security requirement

Figure 6.9: Security requirements, Attack Tree, and System architecture

From the refinement point of view, the dependency relationship means that a single change in any of the activity involved in the SREP (i.e., security attacks, goals, system architecture) may lead to changes in SRs. Consider for example a case where the security attack (i.e. DoS – attack tree node 1) would be prevented by modifying the system architecture: there is no longer a need for any availability SR in that case, or if the attack definition changes, the availability SR should meet the new attack contract. For instance, we can assume that a number of previously considered security attacks and vulnerabilities might not possible anymore, or refinement in the system design makes it difficult for an adversary to exploit this attacks (i.e., DOS attack). Nevertheless, we should not neglect the fact that sometimes such a refinement can also make it easy for an adversary to exploit other security weaknesses. In this regard, it is desirable that SRs are defined and refined in relation with the other system wide development activities, because it may otherwise lead to inconsistent and incomplete SR specifications. Inconsistencies often arise because multiple conflicting requirements are introduced into the SR, or because the system functional

specifications themselves are in a transient stage of evolutionary development.

Additionally, we have learned from our experience towards security requirement refinement [129], that sometimes a trivial adjustment in the functional specification of the system architecture design or a small adaptation in the assumption specification leads to a completely different realization or refinement of security concerns. For example, during the system architecture design, suppose that functional requirement about the message encoding scheme is changed from BER to DER [65], in order to be compliance with underlying layers such as low level drivers and Hardware Security Module (HSM) interfaces [136]. This variation in the system architecture design, provides an opportunity to the expert adversaries, who have knowledge about system architecture, to encode the messages with previously message encoding scheme (BER), and send to the vehicle to make system busy in analyzing and decoding unwanted message formats. This may lead to either system crash, or system remains too busy in decoding unnecessary messages that it is not able to provide services (i.e., DoS attacks) to other legitimate requests [58]. Furthermore, we have experienced with a system, which is a combination of a heterogeneous landscape of technologies (e.g., RPC) and includes various off-The-shelf components from third parties: they also need to be analyzed and coordinated to determine and refine security requirements. We have identified several security attacks and vulnerabilities (as mentioned in Chapter 5), due to the use of a middleware layer, which was introduced in later stages of the system development [58]. From this perspective, SRs and other system development activities have complementary relationships.

Walking through these different concerns, we note that we cannot simply rely on just parallel refinement models. Instead, there are strong relationships between SRs, system assets, and security attacks. There is a need for an integrated approach where these security classes can be linked together; starting from an initial high level goal specification and refining down to concrete security mechanisms that can be enforced by the system model. However, existing frameworks or methodologies [70, 47, 88] for SR refinement are falling short with respect to that objective as they consider refinement separately in these different dimensions (if they even consider more than one dimension). They also generally fail to link requirements together. The next section describes our approach towards the refinement of security requirements in relation to different security classes.

6.3.2 SR Refinement Process

Contrarily to what is often done in requirement engineering approaches, we develop an approach for SR refinement which not only follows the iterative and incremental development lifecycle but also deeply rely on the relationship and concepts defined in the SREP (cf. Section 2.4.1). More precisely, the relationships coming from the SREP are considered as the most important driving factor for the refinement of the SRs specification. We in particular believe this consideration is of utmost impor-

tance to realize the vision of security requirements engineering as the driving force behind the design and implementation of a secure system. This approach can be supported only by a constant dialog between the design of system functions, the requirements that are attached to them, the design and development of the system architecture, and the assessment of the threats to system assets, which would give momentum to the refinement of the security requirements, as well as to that of the system architecture and threat analysis. Thus, the focus is on refining the amount of detail, which facilitates both the selection of the right architectural solutions and the specification of a security rationale for architectural choices. For the purpose of clarity, rather than refining the whole system development process, we refine only that part having to do with security requirements refinement. The SR refinement process, in particular, follows the same steps and inference rules as defined in the SREP (cf. section 2.4.1), depending on the level of details and properties we are interested in. That is to say, the preliminary constructed specifications (i.e., system models, security attack specifications, risk metrics, and security requirements specifications) are refined to more detailed specification. As soon as the system models are evolved to new states, where all the new functions and mapping parameters (i.e., arbitration policies, priorities, etc.) are specified with their possible values, we insert this knowledge into the corresponding knowledge base of these security classes. For instance, consider the case of structural models in particular architecture description (cf. section 4.4.2.2), where more detail about the RPC layer is included in the middleware specification.

At the ontology level, we used the "hasSubclass" relationship (as described in section 3.4) to link the definition of RPC layer with the middleware view. In this case, we take advantage of new knowledge and analyze the evolved system models for the identification of security attacks and vulnerabilities. For instance, we can use the Equation 6.3 to extract the new knowledge about the system architecture. This query returns a list of architecture components and their subclasses as shown in figure 6.10. However, in order to limit the search space, we have used the concept of reference attribute, defined in section 3.3.3, which links the design artifacts to the security requirements as well as to security attacks. In this case, we only query for those particular concepts that are referenced in the design. For example, if we look at the "impersonating valid users" attack (see figure 6.11), a reference to system asset "middleware" is specified in its reference parameter. We use this value and query (Rule 6.3) the architecture ontology to detect and analyze the current state of this particular architecture component. This query returns a value RPC layer and its subclasses such as "FirmwareFlashing", "log_off", etc as shown with dotted lines in the Figure 6.10. Based on the result, we infer and refine the threats using rules specified in Section 5.2, and embed this knowledge into the attack tree diagram. Note that the refinement process goes on until reaching terminal conditions that are either realizable security attacks in view of the adversary's capabilities. This is illustrated in figure 6.11, which shows the refinement of an attack tree node (AT.4.b) into a "log_off" attack (AT.4.b.1). A detail description of this attack is presented in

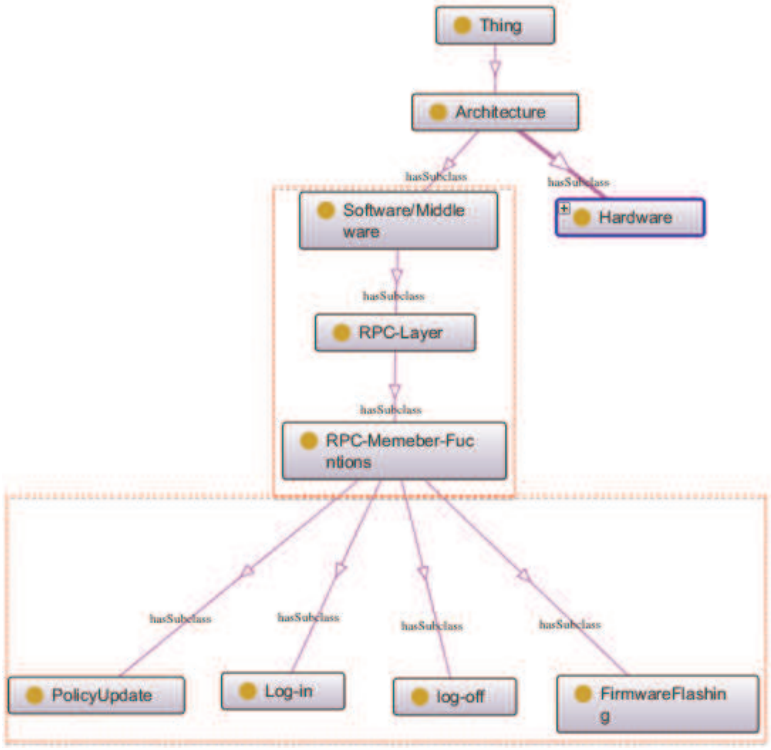


Figure 6.10: Ontological representation of the refined system architecture

section 5.4. In addition, while refining attack tree, we add/update the source value (i.e., source of the attack) in the reference attribute, which in this case is "RPC layer". In this way, it is possible to make sure that, at the given level of abstraction, for example, a particular security attack is discovered due to some specific system configuration. Using reference attributes in this way increase the maintainability of security attacks as well as security requirements and makes its unnecessary repeating the same rule for different ontology classes. On the one hand, the use of a reference attribute also solves the problem, which we exposed in Section 6.3.1. It provides us with a way to link with different system development activities as well as allows us to support step-by-step refinement to fulfill our refinement design principles.

In the next stage of the refinement process, we use this newly generated knowledge and decide if any refinement of SR is required, or whether SR still covers the desired goals, attacks, system requirements, etc. In particular, we use different inference rules (as described in section 6.2), and combination of different condition to refine the security requirements. For instance, we can consider the Rule 6.3 to extract the knowledge about security attacks and their targeted system assets. This rule for instance returns the "log_off" attack and the "RPC layer" as a result. Based on this result we refine the "ensure the identity of a service station" requirement into " RPC layer authentication" (see figure 6.12), which states "RPC member functions should only be executed by the entity that authenticated as that

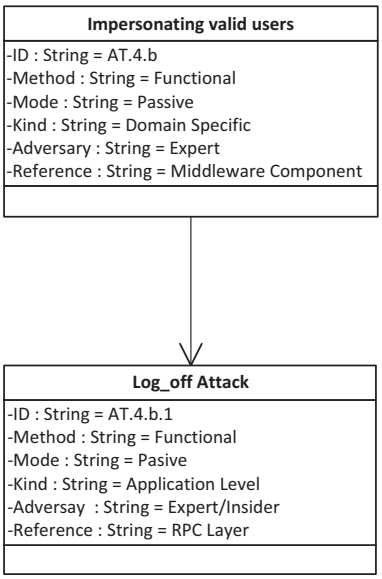


Figure 6.11: Refined Attack Tree

designated in the parameter". We use the SysML «deriveReq» relationship in order to specify the relationships between these requirements. The SRs that results from a refinement cycle are at a refinement level suitable to prevent security attacks and meet the requirement of refined system architecture. We may thus now apply our reference concept here again to relate security requirement with the security attack (i.e., AT1.1) and a reference to the system asset (i.e., RPC layer) is specified in the SR diagram in order to link and maintain the relationship between different models. The goal of this step is to refine the SR specification so that enhances and clarifies previously specified SRs in relation to other security classes (i.e., attacks, goals, etc.) and their constructs.

6.4 Security Requirements Traceability

In the previous chapters, we have defined the different processes to perform the SRE, through a conceptual construction of ontologies and associated security classes. The SREP describes an iterative and incremental construction of the SR specification whose focus is to provide a relationship between the SRs, the security attacks, and the other system development activities like the architecture design. However, as mentioned in the chapter 2, major concern in designing secure systems, especially the ones with evolving security requirements, is tracing the source of requirements and understanding why and how the system meets the current set of security requirements. In this context, it is essential to maintain the traceability link between security requirements and the other security classes during the system development process. However, as can be seen from the above SRE model, linking security re-

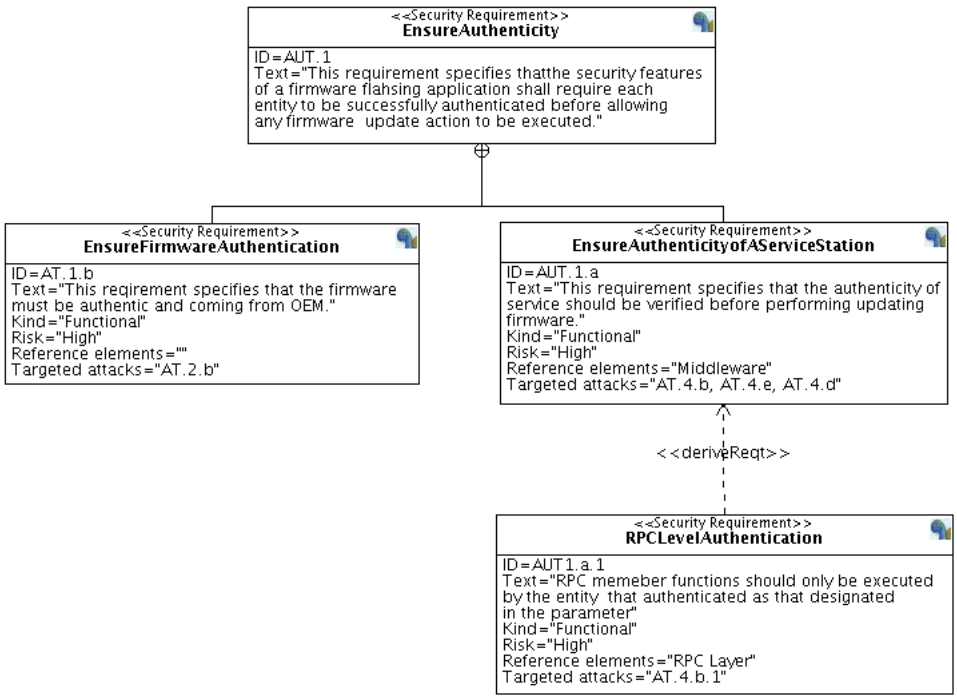


Figure 6.12: Derived security requirements

quirements to their sources, and providing traceability links during the SREP can be along several dimensions. In particular, different roles (i.e., business modelers, security engineers, test and verification teams, etc.) contribute to capturing and building the SR specification, often with divergent perspectives. For instance, the types of relationships of interest to a test engineer are tracing and verifying the relationship between security goals and enforced security requirements. In contract, a security engineer interests are in providing the link between security requirements and other security classes. To help in this alignment, we developed security requirements traceability metamodel (see figure 6.13). The purpose of this metamodel is at least two fold:

- Building the source – target relationship between different security classes to trace the origin of security requirements across different system development efforts.

- Allowing designers to link and show that the security classes, more specifically security requirements meets the system design at different development stages, and helping with the early recognition of those security requirements not satisfied by the system design.

In our context, as introduced in chapter 3 and exemplified in our various examples, we have used the reference attribute to link all different security classes and their generated artifacts. In the case of traceability relationship between security require-

ments and other security classes (i.e., system assets, security attack class, etc.), the reference element of the meta-model is defined by:

- A security class (Source), which references an element of the security class(es). More specifically, the source relation documents the essence, which the security requirement is based on. The source element can be, for example, a system asset, an attack tree node(s), a value from the risk matrices, or security requirements.
- A security class (Target), which references an element of the security requirements class.

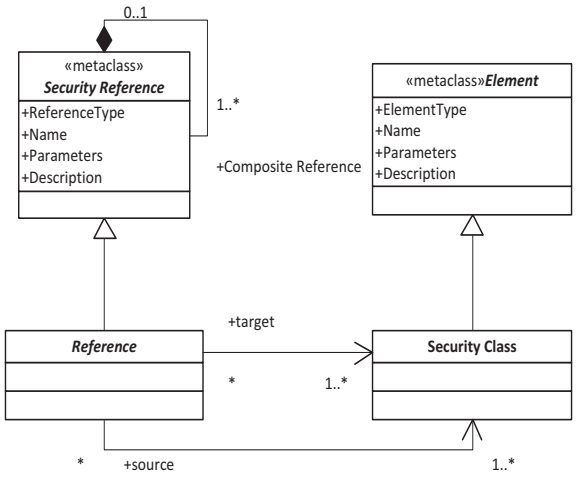


Figure 6.13: Security requiremnt traceability metamodel

As discussed earlier, the advantage of the source – target relationship as a seed for the subsequent analysis is that, we can directly relate different security classes (i.e., security attacks, security requirements, etc.) with their origin, as well as with the reason for their existence. However, one of the concerns in managing traceability links is that SRs originates from multiple sources and also have strong dependency relationships with one another. In this context, a composite reference is required to group and trace SRs though the lifecycle at the different level of granularity. By "composite reference", we mean aggregating individual references into one or more complex traceability links. To support these security requirements traceability links, we have made several extensions to SysML semantic (presented in section 3.3), in order to link and cross-reference SysML diagrams (i.e., SysML AT Diagrams, SysML SR Diagrams, etc.) with one another. We also used the SysML «trace» relationship to add a source reference to any SysML model item, thus indicating its origin. Once the reference attribute is specified, we can obtain a detailed report of the SRs traced, using the TTool report generation mechanism. A SR traceability table, shown in figure 6.14, illustrates the interest of this reporting feature. In this table, the ID column depicts the identifiers of SRs and also indicates the dependencies between different security requirements at the different levels of system abstraction such as,

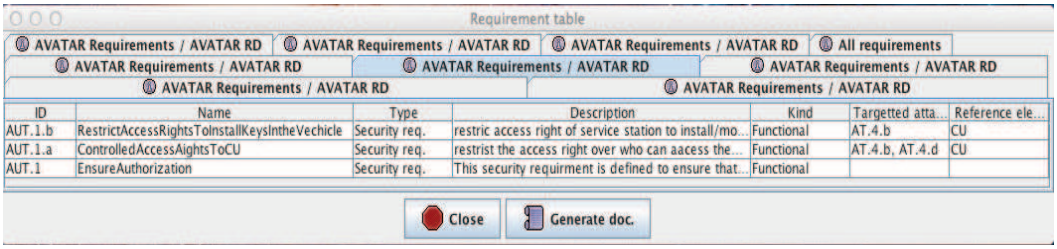


Figure 6.14: Security requirement traceability table

AUT-1 and AUT-1.a, AUA-1.b and so on (see figure 6.1). The following two columns (Name, Type) represent the SR and defied SR type. The Kind column specifies the more concrete category these requirements belong to. The columns Targeted Attack and Reference Element contain a reference of the attack tree node and a reference to system asset elements for which the security requirement is specified. Our summary table (figure 6.14) illustrate the capabilities of our tool in tracing SRs along with additional properties, which we have specified in SRO such as the type of SR, SR kind, as well as assumptions (if defined) during the security requirements engineering process, etc.

6.5 Where we Stand

The expression of security requirements is central when it comes to describing how to secure a system. We reviewed in chapter 2.2, SRE approaches and in particular how appropriate they are for the identification, the refinement, and the traceability capabilities to security requirements. Table 6.1, summarizes the core capabilities of our SRE methodology (SysMLsec) in comparison with other SRE approaches. The main observation concerning this comparison table is that currently there is no perfect match with respect to the capabilities that SRE should provide.

6.6 Conclusions

In this chapter, we have presented the SREP that help in capturing the characteristics of SRs and the way these characteristics can be specified using the SysML representation. The aim of specifying security requirements is to provide an input to the secure on-board architecture design, to the model-based verification, to the protocol specification, and to the security architecture implementation. We illustrated why security requirements can be refined with enough precision for supporting the design of security architecture provided if they are extensively linked with security attacks and the system architecture. Finally, we presented traceability metamodel to facilitate tracking, management and assurance of SRs to support our approach. We believe that, SREP is expressive enough to model and build fine-grained SR

	Conceptual Classification	Security Requirement Approaches	Core Representation	Capabilities	
				Refinement	Traceability
Unified SRE Methodology	Goal Based	KAOS	Goal	■	⊞
		Secure i*	Goal	⊞	□
	Model Base	Secure Tropos	TROPOS	⊞	□
		UMLsec	UML	⊞	⊞
		SecureUML	UML	□	□
	Problem Oriented	Abuse Frames	Problem Framework	□	□
		Misuse Cases	Use Case	□	□
	Process Oriented	SQUARE	No specific modeling diagram proposed	⊞	□
		SysMLsec	SysML	■	■

Table 6.1: Comparative analysis of security requirements approaches. The degree of fulfillment will be "■" for available properties, "□" for not available, and "⊞" for partly or optionally available properties.

specifications; at the same time, the complexity of adapting SREP to subsequent stages of the system development lifecycle is manageable.

Part III

Security Requirements Enforcement

Chapter 7

Constructing Security Specification of Cryptographic Protocol Design

7.1 Introduction

When developing cryptographic protocols, the most common design principle is specifying the behavior of protocols, security properties to be enforced, as well as discussing the context/environment in which they will be used. While this simple logic is not always explicit or evident when one designs cryptographic protocols, there are various dilemmas in describing knowledge of cryptographic protocols in this manner, in particular, two important issues. First, the notion of knowledge typically depends upon certain mixes of functional and non-functional constructs and the knowledge of a few system parameters: the sequences of states or events (protocol itself), security requirements, the operational environment and system boundaries, and security expertise about security mechanisms. Second, a more important problem relates to how this security knowledge and conceptual foundations for different security classes (i.e., security requirements, security algorithms, security mechanisms, etc.) is shared and used in the design of cryptographic protocols. We experienced during the design and verification of cryptographic protocols for automotive on-board network that, even though the protocols are based on cryptographic building blocks, and proven backed by hardware mechanisms, the adversary is still capable of performing considerably simple attacks (cf. logoff attack presented in section 5.4). In particular, such kind of attacks are possible due to the lack of relationship binding between different architecture layers and also due to the weak association with other security constructs (i.e., security requirements, system architecture, etc.). In a similar way, it is also challenging to decide about security mechanisms and security controls without having enough details about all involved artifacts, their properties, and how security related information is exchanged and used by different functions in the protocol specification. Given these constraints, constructing cryptographic protocols is a subtle and complex task, because it is difficult to speak of any logic for security protocols without relating to different security constructs and classes. In this perspective, we argue that the concept of ontologies make it clearer what are the relationship between security mechanism and security

requirements. In general, such ontology allows a cryptographic protocol, together with its knowledge representation, to be constructed in accordance with functional and non-functional security aspects of system architecture.
The focus of this chapter is consolidating and developing the relationship among different ontologies that can be used to design cryptographic protocols specification. In section 7.2, we propose ontology for cryptographic protocols design, which, in turn, facilitates the construction of cryptographic protocols. With an ability to refer to different security constructs, the proposed ontology provides modularization in such a way that cryptographic protocols can be combined and, when appropriate, interchanged while preventing that certain security functionality is implemented redundantly. In the sequel, in section 7.3, we present an OTA firmware update cryptographic protocol to show how this security protocol ontology is employed in order to secure firmware update process. As already highlighted in chapter 5, attacks on the in-vehicle network have serious consequences for the driver. If an adversary can downgrade the legitimate firmware with his malicious firmware, he can essentially control the functionalities of the vehicle and perform arbitrary actions on the in-vehicle network. The OTA firmware flashing cryptographic protocol has especially been designed with respect to these attacks as well as by considering the functional and non-functional requirements of such heterogeneous on-board architectures. Our approach provides the link between hardware security anchors (integrated security modules) and software security framework, which is necessary to achieve an enhanced trust level for safety-critical applications like influencing the vehicle's behavior. Section 7.4, reviews the capabilities of existing OTA firmware update protocols in comparison with our own protocol specification. In section 7.5, general conclusions concerning the functionality of security protocol ontology and the design rational of firmware update cryptographic protocol for such heterogeneous automotive networks are drawn.

7.2 Ontology for Cryptographic Protocols

The objective of ontology is to assist the protocol designer in reviewing and relating different security concepts when designing the cryptographic protocol specification. In this section, we highlight different security classes of the ontology (see Figure 7.1) and the relationships binding between their conceptual foundations.

- **Security Requirements** class describe the desired security behavior expected of a system. In particular, the purpose of security requirements is not only to specify per-assets security needs, but also to ease the selection of appropriate security measures and the enforcement of security constructs. We use the security requirement ontology (see Section 2.4.2.4) to extract the knowledge about different security requirements and their classification.

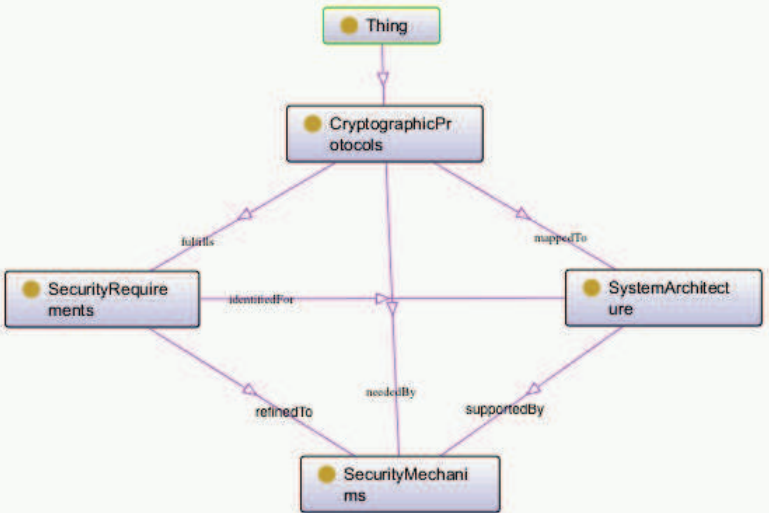


Figure 7.1: Cryptographic protocols ontology

- **Security Mechanisms** class describe methods and techniques that are used to implement and enforce security requirements. More precisely, this class serves as the "security building block" whose result is the basis for SRs enforcement. The organization of this class is depicted in Figure 7.2.

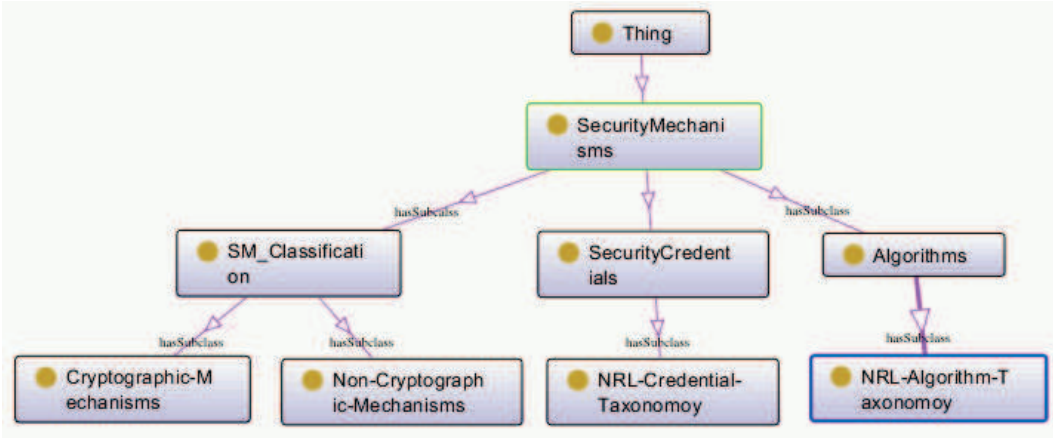


Figure 7.2: Core classes of security mechanisms ontology

  - Security mechanisms classification: This class is defined to distinguish between different security mechanisms used to enforce security objectives. We categorize security mechanism in to two categorizes: cryptography and non-cryptographic security mechanism. For example, applying either simple password based security solution can enforce authentication or approaches like authentication with symmetric challenge-response techniques or smart card based mechanisms like single sign-on (SSO), one time password (OPT), etc. can be used. Each of these security mecha-

nism represent different construct to achieve the same security objective; that is based on non-cryptography and cryptography security constructs, respectively.

– Security algorithms: This class is defined to categorize and systematically aggregate security algorithms into a set of well-defined classes that provide a comprehensive description of cryptographic algorithms and their objectives. Here, we can reuse the security algorithm taxonomies provided in [77, 157]. The organization of NRL algorithm taxonomy [77] is depicted in Figure 7.3.

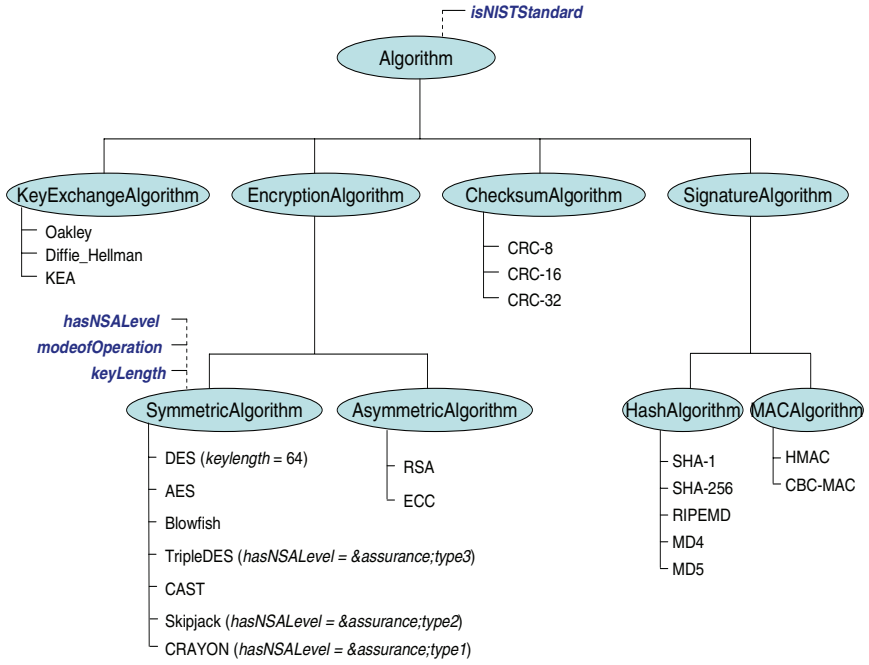


Figure 7.3: NRL security algorithm taxonomy [77]

– Security credentials: In this class, systems assets are classified in terms of their properties towards security such as smart card, passport, fingerprint, etc. To annotate specific system assets that support security credential class, we reuse the credential taxonomy (see Figure 7.4) developed by NRL [77].

- **System Architecture:** The cryptographic ontology is further enriched with knowledge about system architecture. The security architecture constitutes the framework that describes how system assets interact and work together to achieve global system objectives. In particular, it describes the behavioral and structural models of the system, what each asset of the system does, and what information is exchanged among system assets as well as security functionalities supported and provided by different components of system architecture.

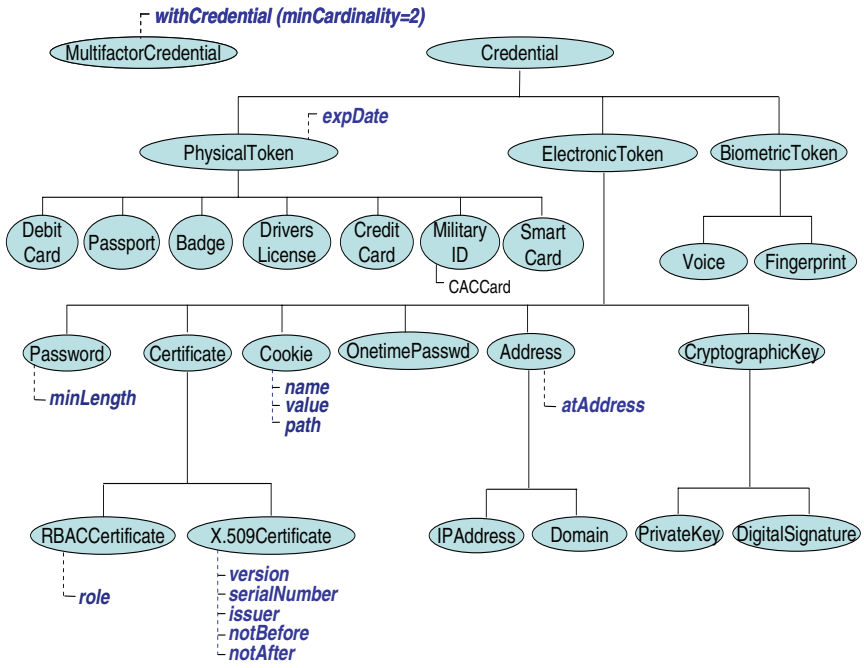


Figure 7.4: NRL security credentials taxonomy [77]

In this context, we can make a link to the previously defined system architecture ontology (se Section 2.4.2.2), which provides different architect views (i.e., application view, middleware view, infrastructure view), and logical and conceptual models of the system, and describes the relationships among different assets.

7.3 Firmware Flashing Cryptographic Protocol

In the following we present a cryptographic protocol for firmware flashing application and show how the different classes specified in the ontology can be adopted to build a cryptographic protocol specification. We start building cryptographic protocol specification by extracting and tracing back to different knowledge bases that we have developed so far. For instance, let us start from extracting the knowledge about functional security requirements (FSR) by querying (Rule 7.1) the security requirement knowledge base. This query returns all the FSRs that we have specified for firmware flashing application, during the SRE process.

$$SR(?r) \wedge hasFunctional(?r, ?FSR) \rightarrow select(?r, ?FSR) \tag{7.1}$$

Based on the results, we decided to select authenticity (Figure 6.1), integrity (Figure 6.2), freshness (Figure 6.3), authorization (Figure 6.4), and confidentiality (6.5) as the security criteria of interest regarding the firmware update process. In addition,

thanks to the knowledge specified in the SysML SR diagram, these SRs also provides us details about security attacks they cover, and their relationships with system assets. For instance, in order to prevent service station from `"installing bogus authority keys"` (see SR – AT.4.b in Figure 5.2) in the Hardware Security Module (HSM), we have specified the `"ensure authenticity of service station"` (see SR – AUT.1.a in Figure 6.2) and `"restrict access rights to the HSM"` (see SR – ATH.1.b in Figure 6.4) security requirements. Thus, facilitate us to decide appropriate security mechanisms as well as how to enforce them in order to prevent security attacks. For instance, we can enforce the `"ensure authenticity of service station"` security requirement by defining a `"digital signature"` security mechanism. In a similar way, we translate each security requirement into a specific security mechanism or set of security mechanisms. In order to do so, we first extract (Rule 7.2) the knowledge about system architecture such as type of an ECU, its properties, security services supported by a particular system assets, etc.

$$\begin{aligned} &Architecture(?a) \ \wedge \ hasFunctions(?a, \ ?Funtions) \\ \wedge\, &hasSequence(?Functions, ?Sequence) \wedge \ hasAssets(?a, \ ?Assets) \\ &\rightarrow select(?a, ?Functions, ?Assets, ?Sequence) \end{aligned} \tag{7.2}$$

As we previously mentioned, the reason to extract the knowledge about system architecture is to have enough details about all involved artifacts, their properties, and what kind of information is exchanged and used by different functions in the protocol specification. All these details will certainly help us to select appropriate security mechanisms. For example, in order to satisfy the performance requirements for signing and verifying messages for V2X communications, a very efficient asymmetric cryptographic engine is required, whereas, for in-vehicle communication, we make use of shared secrets (i.e., symmetric keys) due to cost and embedded constraints [161, 134]. Based on the analysis of the security requirements, the system architecture as well as extracting knowledge from the security mechanism class, we have specified the security primitives in section 7.3.1 and used these security primitives to develop the secure firmware flashing protocols in section 7.3.3.

7.3.1 Security Primitives

In this section, we briefly describe the security primitives that are required during firmware flashing process. In particular, the Hardware Security Module (HSM) provides these security primitives. A topology to enable ECUs to implement cryptography security primitives in a secure manner is shown in Figure 7.5. In this figure, the application CPU(s) of an ECU is equipped with a cryptography coprocessor HSM. This module is responsible for performing all cryptography applications including symmetric encryption/decryption, symmetric integrity checking, asymmetric encryption/decryption, digital signature creation/verification, and generation of

random numbers used for security applications.

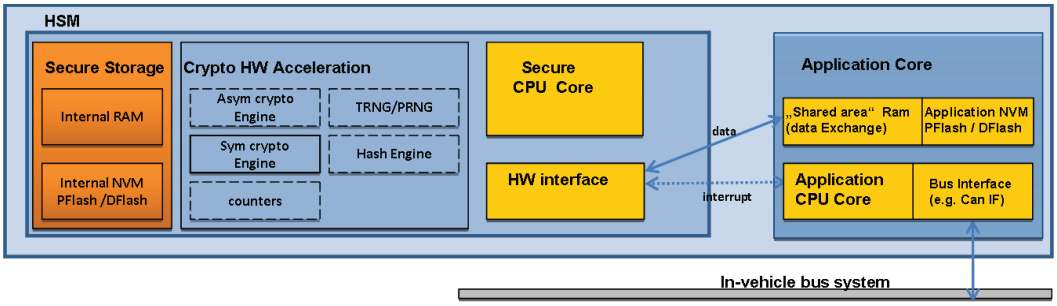


Figure 7.5: Hardware Security Module – HSM [161]

- **Signature:** This function is used for demonstrating the authenticity and integrity of a message. A valid signature gives a recipient reason to believe that a known sender created the message, and that it was not altered in transit. For signature generation, a signature generation scheme $sig(m)_k$ takes as input a key k, and message m, outputs a signature $\mathcal{S}$; we write $sig(\mathrm{m})_k = \{\mathcal{S}\}_k$. Where k is the security parameter, outputs a pair of keys $(s\ ;\ v)$. s is the signing key, which is kept secret, and v is the verification key which is made public. We also assume that a time stamp (UTC Time) is generated and then also covered by the signature calculation, and write $\mathfrak{m} = (m+t)$ to denote the message and a time stamp whose signature is $\mathcal{S}$. For the signature verification, $ver\_sig(\mathfrak{m}\ ,\ \mathcal{S})_v \rightarrow \mathcal{S}'$ function is defined, takes as input the signature $\mathcal{S}$, the signature verification public key part v, and outputs the answer $\mathcal{S}'$ which is either succeed (signature is valid) or fail (signature is invalid). As a precondition, the v must be loaded and enabled for verification.

- **Message Authentication Code – MAC:** This function is used to protect both the data integrity and the authenticity of a message, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. For generating a MAC as well as the message itself, the notation $MAC\,(m)_k = \{\mathcal{M}\}_k$ is used, so that it produces the message itself plus the cryptographic authentication code based on k and m. Here, k refers to a cryptographic key for MAC generation and m to the message to be authenticated. In the same way as for signatures, the use of the time stamp $\mathfrak{m} = (m+t)$ is covered by the MAC calculation. For the verification of a MAC, the notation $ver\_MAC(\mathfrak{m}, \mathcal{M})$ is used. Based on the k, it is verified whether $\mathcal{M}$ corresponds to the message $\mathfrak{m}$.

- **Key Management:** This module provides following functionalities for internal key creation (using the internal RNG), key import and export:

  – Key Creation is used for the creation of a key k on a hardware module, using HSM Create_Random_Key function. All properties of the key are

determined and fixed during creation. This includes the cryptographic algorithm to be used, the use and further property use flags indicating what actions may be done with this key (i.e., sign and verify) as well as the authorization data needed for key usage. Additionally, the creator of a key has the possibility to set individual usage authorizations (use flags) for each key usage. The use flag parameter indicates the operations that may be performed with the key. In particular, the following flags are present:

* `sign|verify`: Key can be used to generate and/or verify digital signatures or H/MACs of any data.
* `encrypt|decrypt`: Key can be used to encrypt and/or decrypt any data.
* `secureboot`: Can be used to create/verify secure boot references.
* `keycreation`: Can be used for creation of new keys, e.g. via key derivation functions (symmetric) or DH key agreement (asymmetric).
* `securestorage`: Can be used to realize (locally bound) secure storage
* `utcsync`: Can be used for synchronizing internal tick counter to UTC.
* `transport`: Can be used to protect transports of keys (i.e., migration, swapping, move) between locations, according to individual transport flags (i.e., 0 = INT, 1 = MIG ,2 = OEM, 3 = EXT).

Only the use flag may explicitly be set by the creator whereas further property flags are set inherently. Once created, the key properties are unchangeable. As output, the function delivers a key handle for later usage of the key.

– Key Export is used for moving keys between different HSMs, between HSMs and external (trusted) locations (if permitted). The HSM provides key_export functionality that ensures confidentiality by encrypting ($\mathcal{E}(k)_{\mathcal{T}_k}$) private key internals via a special transport key ($\mathcal{T}_k$) (symmetric or asymmetric) transport encryption as well as authenticity of all key data structures via (symmetric or asymmetric) so-called transport authenticity codes (i.e., a digital signature or a MAC). The key authenticity code can be an explicit symmetric key enabled with use flag = verify or an implicit symmetric/asymmetric key derived from a transport key. The use of this key authenticity code is mandatory. As output, the function delivers the encrypted key together with its authentication code; we write $\mathcal{E}(k)_{\mathcal{T}_k} = \{\mathcal{K}e\}_{\mathcal{T}_k}$. As an important precondition, the specified transport

key must be loaded and enabled to be used for transport. Furthermore, the transport flag of the key to be exported must be appropriately marked according to the type of module managing the transport key.

– Key Import is used for importing keys into HSM or to other trusted parties. In this way, the key_import function provides the counterpart to the previously described export function. The key k may be imported either into the non-volatile memory or into the main memory (RAM) of the HSM. In the same manner as for key export, the use of the key authenticity code is mandatory. As output, the function delivers a key handle to reference the key for later usage. As a precondition, the transport key must be loaded and enabled before. In addition, the authentication code verification key must be loaded if the key is protected by a signature.

- **Key Master – KM:** We introduce a new functional entity, which we call the key master. As there exist multiple variants of the HSM, that support different cryptographic keys (symmetric/asymmetric), we had to take this into account for key distribution. The KM is a central element in the establishment of a session between entities. It holds public key ($\mathcal{P}$) and pre-shared keys ($\mathcal{S}_k$) of the individual ECUs, which are used as transport keys, to establish a secure session. This functional entity resides on a dedicated ECU or is integrated into another ECU. There may be more than one KM node in a vehicle for replication purposes.

- **Monotonic Counter:** serves as a simple secure clock alternative while providing at least 16 monotonically increasing 64-bit counters together with corresponding access control similar to TCG's monotonic counters [42]. For handling these counters, the following HSM functions are provided: Create_counter; Read_Counter, Increment_Counter, and `Delete_Counter`. Access authorization data needs to be provided as input data, and is later necessary to create, increment or delete the counter.

- **Pseudo Random Number Generator:** creates pseudo random numbers with a PRNG algorithm specified on invocation that can be seeded internally from a physical true random number generator (TRNG) or from an external TRNG during production in a controlled environment of the chip man- ufacturer. The latter case additionally requires a proper seed update protocol. All prototype modules provide at least an officially evaluated PRNG according to E.4 [131] (e.g., AES- or hash-based).

7.3.2 Assumptions and Constraints

Before sketching the protocol, we describe some additional assumptions and constraints that have to be taken into account for secure firmware updates: including

the secure storage of key material in the diagnostic tool and secure transport of firmware data over the on-board bus system (i.e., CAN).

- **HSM in the Diagnostic Tool:** As mentioned already in Chapter 5, there are numerous scenarios, where an attacker targets the diagnostic tool (DT). For instance, the attacker might inject bogus authority keys into the ECU, through DT, which compromises the overall security of the vehicular on-board architecture In particular, this means that the DT stores challenges and public strings for key recovery (i.e., ECU unlock key) and is therefore responsible for the security of the subsystem. Therefore, this information needs to be stored securely on the DT-side. An additional advantage of HSM is the resistance against physical tampering of the DT. Any damage to the HSM changes the behavior and therefore prevents the extraction of secret key material.
- **Bandwidth Limitations of In-vehicle Networking Technologies:** Firmware update protocols comprise two parts: a V2I part, and an intra-vehicular part, the latter involving a large number of interconnected ECUs. Secure transport protocols are needed for the exchange of on-board messages. In the on-board bus systems used, a specific restriction lies in the limited size of data packets. For the CAN bus, for example, this means that only eight bytes of payload may be transmitted at a time. For this purpose, secure common transport protocols (S-CTP) [59], extensions of the CTP defined in [15] are applied to diagnosis jobs, where typically larger data chunks need to be transmitted.

7.3.3 Cryptographic Protocol Specification

To simplify the description of the protocol, we split the firmware update protocol in five sequential phases named (1) remote diagnosis, (2) ECU reprogramming mode, (3) firmware encryption key exchange, (4) firmware download, and (5) firmware installation and verification. In the next subsections, we describe the design of the firmware flashing protocol. The resulting protocol is shown in Figure 7.6.

7.3.3.1 Remote Diagnosis

In the firmware flashing process, a service station using a diagnostic tool (DT) connects remotely to a vehicle, using V2I communication channel, to assess the state of the vehicle. To know which version is installed, a diagnosis of the vehicle is required to have all necessary information such as ECU type, firmware version, and date of last update. An employee of the station using the DT establishes a secure connection with the vehicle, at the ECU level, in order to determine the current state of the vehicle.

To do so, DT creates a session key k_s (exportable), by sending a HSM command create random key and specifies the set of allowed key properties as

$$k_s : create\_random\_key(\texttt{target\_algorithm\_identifier}, \texttt{key\_size}, \texttt{valid\_until} \\ \texttt{memory\_target\{nv|ram\}}, \texttt{key\_usage\_size}, \texttt{key\_usage\_data}) \tag{7.3}$$

It then calls export function (see Equation 7.4) of the HSM to encrypt the k_s, using $\mathcal{P}_\mathfrak{b}$[1] (Public key of the central communication unit – CU) as a transport key ($\mathcal{T}_k$), and transmits a freshly created session key along with its signature to the vehicle, gives us message exchange as shown in Equation 7.5.

$$\{\mathcal{K}s\}_{\mathcal{P}_\mathfrak{b}} : key\_export(\texttt{key\_handle}, \texttt{use\_flags transport\_key\_handle} \\ \texttt{transport\_key\_authorization\_size}, \texttt{authenticity\_key\_handle} \\ \texttt{authenticity\_key\_authorization\_size}, \texttt{authenticity\_key\_authorization}) \tag{7.4}$$

$$\mathcal{A} \rightarrow \mathcal{B} : m, \{\mathcal{S}\}_{s_\mathfrak{a}} \quad here, \mathfrak{m} = (\{\mathcal{K}s\}_{\mathcal{P}_\mathfrak{b}} + t) \tag{7.5}$$

Here, the CU is the first receiving entity in the vehicle, responsible for receiving and distributing messages to the on-board network. In the vehicle, the CU, equipped with the HSM and acting as a key master (KM) node, receives the connection request. The authorization for the connection is verified in the CU, by calling the PDM. The message is checked for freshness, integrity and authentication of the service station is also verified by calling the EAM. If the check succeeds, CU imports the key k_s into the HSM (see Equation 7.6). It then exports the received $\{\mathcal{K}s\}_{\mathcal{P}_\mathfrak{c}}$ with the corresponding $\mathcal{P}_\mathfrak{c}$ and distributes it to the target ECU (see Equation 7.7) in order to enable end-to-end communication. This message includes all information that is necessary to deliver this message to the correct ECU.

$$\mathcal{K}i : key\_import(\texttt{transport\_key\_handle}, \texttt{transport\_key\_authorization\_size}, \\ \texttt{transport\_key\_authorization}, \texttt{authenticity\_key\_handle} \\ \texttt{authenticity\_key\_authorization\_size}, \texttt{authenticity\_key\_authorization}, \texttt{memory\_target}, \\ \texttt{import\_key\_size}, \texttt{import\_key}, \texttt{key\_authenticity\_code\_size}, \texttt{key\_authenticity\_code}) \tag{7.6}$$

$$\mathcal{B} \rightarrow \mathcal{C} : m, \{\mathcal{S}\}_{s_\mathfrak{b}} \quad here, \mathfrak{m} = (\{\mathcal{K}s\}_{\mathcal{P}_\mathfrak{c}} + t) \tag{7.7}$$

On the receiving side, ECU verifies the integrity, authenticity and authorizations of CU as well as for DT, based on the policies specified in PDM as to whether DT

[1]For clarity reasons of our OTA firmware update protocol description, we denote the diagnostic tool (DT) as "$\mathcal{A}$", the communication unit (CU) as "$\mathcal{B}$", ECU as "$\mathcal{C}$", and the OEM as "$\mathcal{D}$". Furthermore, all cryptographic operations such as generation or import/export, signature verification, integrity checks, etc. take place inside the respective HSMs.

is allowed to deliver a message or not. If this is true, and the message is fresh, ECU imports the k_s in the HSM, using the same equation defined in 7.6. Once key k_s have imported, an acknowledgment is sent back to DT. After this acknowledgment frame, the DT sends, depending on the option chosen by the employee of the service station, requests to read out diagnosis information (State/Log information) from the ECU it wants to check.

Advance Notification: Due to legal reasons and to allow for flexible deployment, we consider that service station will send an advance notification of possible firmware updates, if the type is the expected one. This advance notification is intended to help customers plan for the effective deployment of updates, and includes information about the number of new updates being released. These updates still need to be approved for install before downloading. The customer receives this information on the vehicle human-machine interface (HMI) and can decide about possible deployment (i.e., Install, Decline, Decide later, etc.). Only updates that have the approval status Install will be downloaded to the vehicle. Disabling any ECU while vehicle is running may cause safety critical problems, depending on the function ECU is responsible for. We thus assume that additional checks will be performed by the on-board system, to ensure that the vehicle is stopped and has access to the infrastructure, before switching the ECU into the re-programming mode. Furthermore, we assume that the V2I communication is available throughout the OTA firmware update process.

7.3.3.2 ECU Re-Programming Mode

If the type is the expected one, the DT forces the ECU to switch from an application mode into a re-programming mode by requesting a seed (see Equation 7.8). This seed is required to calculate an ECU specific key value to unlock the ECU for re-programming. The ECU verifies desired security properties. ECU verifies whether $\mathfrak{m}$ is authentic and fresh by verifying the $\{\mathcal{M}\}_{k_s}$. If it is true, ECU sends a HSM command SecM_Generate(seed) to generate a seed Na as shown in Equation 7.9. It then encrypts the seed $\mathcal{E}(Na)_{k_s}$ for confidentiality enforcement, compute a $\mathcal{M}$ using key k_s and transmits it to the DT (see Equation 7.10).

$$\mathcal{A} \rightarrow \mathcal{C} : \mathfrak{m}, \{\mathcal{M}\}_{k_s}, \;\; here, \mathfrak{m} = (request\_seed + t) \tag{7.8}$$

$$Na : rng\_get\_random(\texttt{algorithm\_identifier}, \texttt{random\_byte\_request\_size}) \tag{7.9}$$

$$\mathcal{A} \leftarrow \mathcal{C} : \mathfrak{m}, \{\mathcal{M}\}_{k_s} \;\; here, \mathfrak{m} = (\mathcal{E}(Na)_{k_s} + t) \tag{7.10}$$

At the same time, the ECU sends a HSM command to compute the key on the HSM using Na. As output, the function delivers a $\mathcal{K}_u$ key, that is used to unlock

the ECU as

$$\mathcal{K}_u : secm\_compute\_key(\texttt{secm\_seed\_type}, \texttt{secm\_word\_type}, \texttt{secm\_key\_type}) \tag{7.11}$$

On the DT side, it verifies $\{\mathcal{M}\}_{k_s}$, decrypts the received seed $(\mathcal{E}(Na)_{k_s})$, and computes the $\mathcal{K}_u$ with the aid of the received Na, using the same key computation fuction as used by the ECU (see Equation 7.11). Once the $\mathcal{K}_u$ key value is computed, it is exported $\{\mathcal{K}_u\}_{k_s}$, using session key k_s as a transport key, and transmitted to the target ECU as

$$\mathcal{A} \rightarrow \mathcal{C} : \mathfrak{m}, \{\mathcal{M}\}_{k_s} \quad here, \mathfrak{m} = (\{\mathcal{K}_u\}_{k_s} + t) \tag{7.12}$$

The ECU verifies the $\mathcal{M}$, and compares the received $\mathcal{K}_u$ with the self-generated $\mathcal{K}_u$. If the two values are identical, the ECU is switched into unlock state (from application mode to the re-programming mode) and sends an acknowledgement message to the DT. This message is sent after the ECU is switched into the unlock state to make sure the switch has been performed. The information whether a re-programming request has been received or not shall be stored in non-volatile memory, e.g. EEPROM. Since switching from the application to the re-programming mode shall be done via a hardware reset, all contents of volatile memory will be lost [92]. If the comparison failed, the flashloader [92] holds the ECU in locked state. ECU re-programming is possible only in the unlocked state.

7.3.3.3 Firmware Encryption Key Exchange

In this phase we are considering two possible scenarios for exchanging firmware encryption keys: (1) on-line solution and (2) off-line solution. In the on-line solution: the service station has access to an online infrastructure of the manufacturer, it can request the firmware and as well as the firmware encryption key – $\{\mathcal{K}\}_{ssk}$. Here, the SSK is a stakeholder symmetric key pair [59], created externally, with use flag = decrypt only, key for stakeholder individual usage e.g., software update. Instead, in the case of off-line firmware is encrypted with the pre-installed SSK.
Considering current trends and advancements in the automotive industry, on-line solutions provide more reliability, flexibility and will eventually increase the security of the on-board network. Sharing the firmware encryption key only with specific ECUs makes an on-line solution more robust and generic compared with of-line approaches, where all vehicles share unique symmetric keys that are pre-installed in the vehicles. In addition, the existence of various security levels in the architecture [129], pleads for the specification of a validity period of the SSK (short term or long term keys), for an individual ECU.
Following our results from security requirements engineering (cf. Non-Functional Security Requirements – NFSR presented in section 6.2.1.2), we suggest to use short term keys for firmware encryption. Short terms keys will expire after a short amount of time and thus, as there is no need for instant revocation if keys are compromised.

This has the advantage that OEMs do not have to go through another key migration (installing new keys) process if keys are compromised. As such, the following section only details the on-line solution. In this context, the DT sends a request to the OEM server to get the firmware $\mathcal{F}$ encryption key (see Equation 7.13). This message $\mathfrak{m}$ includes information about the ECU (i.e., ECU type, ECU identification number, firmware version, etc.).

$$\mathcal{A} \rightarrow \mathcal{D} : \mathfrak{m}, \{\mathcal{S}\}_{P_{\mathfrak{a}}} \quad here, \mathfrak{m} = (request\_firmware\_key + t) \tag{7.13}$$

The OEM verifies whether $\mathfrak{m}$ is authentic, fresh, and integrity protected by verifying the signature $\mathcal{S}$ of the received message. If verified, OEM server retrieve the $\mathcal{P}_{\mathfrak{c}}$ from the Public Key Infrastructure (PKI), (possibly) maintained by an individual OEM or by third parties, and exports SSK using $\mathcal{P}_{\mathfrak{c}}$ as a transport key. As the SSK key blob is encrypted with the ECU key, It is not possible for the DT to retrieve the firmware encryption key. The OEM server sends a message $\mathfrak{m}$ with an encrypted and signed firmware encryption key to the service station as

$$\mathcal{A} \leftarrow \mathcal{D} : \mathfrak{m}, \{\mathcal{S}\}_{P_{\mathfrak{d}}} \quad here, \mathfrak{m} = (\{\mathcal{K}_{ssk}\}_{\mathcal{P}_c} + t) \tag{7.14}$$

Next, the DT transmits the received firmware encryption key to the ECU as

$$\mathcal{A} \rightarrow \mathcal{C} : \mathfrak{m}, \{\mathcal{M}\}_{k_s} \quad here, \mathfrak{m} = (\{\mathcal{K}_{ssk}\}_{\mathcal{P}_c} + t) \tag{7.15}$$

The ECU imports the SSK in the HSM using the key import function (see Equation 7.6). The key import function provides the assurance to the ECU that the key is generated by the OEM, by verifying the authentication code send along with the encrypted key, and can only be decrypted by the specific ECU key. After importing the SSK in the HSM, the ECU sends an acknowledgment about the successful import of the SSK.

7.3.3.4 Firmware Download

Once the SSK is successfully imported into the HSM, the DT sends the received signed and encrypted firmware $\mathcal{F}$ along with its ECU Configuration Register (ECR) reference to the Random-Access Memory (RAM) of the ECU. Following the HSM use flag approach, where multiple key-properties areq set, only the OEM server can sign and encrypt the firmware, whereas the receiving ECU can decrypt and verify the received firmware, using the same key material, shared before. The encrypted firmware is downloaded block by block (logical block). Each of those blocks is divided into segments, which are a set of bytes containing a start address and a length. The start address and the length of each segment is sent to the HSM during the segment initialization. For one block, a download request is sent from the DT to the ECU. The ECU initializes the decryption service and sends an answer to the DT. The download then starts segment by segment. After sending the last firmware segment, the DT sends a transfer exit message to the ECU as

$$\mathcal{A} \rightarrow \mathcal{C} : \mathfrak{m}, \{\mathcal{M}\}_{k_s} \quad here, \mathfrak{m} = (transfer\_exit + t) \tag{7.16}$$

7.3.3.5 Firmware Installation and Verification

For an installation of the firmware, we consider the standard firmware installation procedure defined in [92], where each logical block is erased and re-programmed. However, before the flash driver can be used to re-program an ECU, its compatibility with the underlying hardware, the calling software environment and with prior versions of the firmware has to be checked. This compatibility check is performed by means of version information stored in the HSM monotonic counters. The HSM read counter function is used to read out the value of a counter. A counter identifier previously increased after every authentic and successful installation of the firmware. These monotonic counters are defined to perform such a checking of its current version against the new firmware version in order to prevent the downgrading attacks meant to install older firmware.

For the verification, we defined a two-step verification process: In the first step, before re-programming, the ECU verifies the signature of the firmware data. This is verified by using the pre-installed Manufacturer Verification Key MVK. It proves that the software was indeed released from the OEM. In the second step: we construct a tiny trusted computing base (TCB) during the installation phase. We compute an ECR trusted chain at each step of the firmware installation. The ECR reference is needed to ascertain the integrity/authenticity of the firmware data. An extend ECR function is defined to build the ECR trusted chain. This function is used for updating the ECR with a new hash value. The new value is provided as input and chained with the existing value stored in the ECR, using a hash update function. As output, the function delivers the updated ECR value.

After a successful installation of the new firmware data, software consistence check is performed. The check for software dependencies shall be done by means of a callback routine provided by the ECU supplier. This check is done after re-programming and before setting the new ECR reference. Next, the compare ECR function is called. This comparison can only be performed after all writing procedures for the logical block have been finished. This function allows the direct comparison of the current ECR with a reference ECR value received with the firmware. It is also possible that the ECR reference may be contained inside the firmware itself. In this case the flashloader shall call a routine provided by the ECU supplier to obtain the ECR reference. If the check succeeds, the HSM preset ECR function is called. This function is used to manage references to ECR values by ECR indices in the context of a secure boot. After successfully setting the ECR value, the HSM increment counter function is called to increment the monotonic counter with the new value. At the last step, the actual hardware reset is executed, the flash-loader deletes (i.e. overwrites) the routines for erasing and/or programming the flash memory from the ECU's RAM [92], thereby making sure those routines are not present on the ECU in application mode. After the reset, the application is started.

Error Handling: Each function of the HSM returns a status after its successful or unsuccessful execution. Some functions may deliver further function specific error codes. The value of the status shows the positive execution of the function or the reason for the failure. In case of a failure, the flash process must stop with an error code and the ECU enters the locked state.

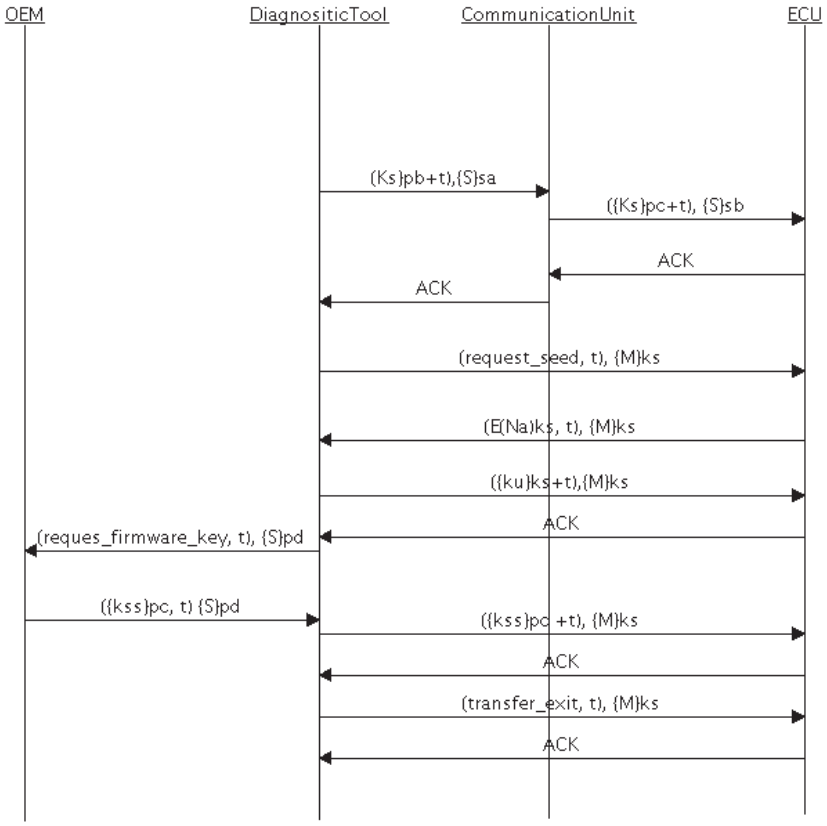


Figure 7.6: Firmware flashing cryptographic protocol

In this section, we have presented a security protocol to show how hardware, software security mechanisms can be used to achieve secure firmware updates process. In particular, by using secure in-vehicle communication and a trusted platform model, we showed how to establish a secure end-to-end link between the manufacturer, the service station and the vehicle. Despite the fact that a trusted platform model entails certain constraints, such as the obligation to bind cryptographic keys to a given boot configuration, we showed how the protocols we presented deal with the update of the platform reference registers during the boot phase of an ECU.

7.4 The State of the Art: Firmware Update

The past decade has seen a tremendous growth in the vehicular communication domain, yet no comprehensive security architecture solution has been defined that covers all aspects of on-board communication (data protection, secure communication, secure and tamper proof execution platform for applications). On the other hand, several projects, namely GST [118], C2C-CC [16], IEEE Wave [160] and SeVeCOM [120] have been concerned with inter-vehicular communication and have come up with security architectures for protecting V2X communications.

These proposals essentially aim at communication specific security requirements in a host-based security architecture style, as attackers are assumed to be within a network where no security perimeter can be defined (ad-hoc communication). These proposals consider the car mostly as a single entity, communicating with other cars using secure protocols. Mahmud et al. [89] present a security architecture and discuss secure firmware upload, which depends however on a number of prerequisites and assumptions (i.e., sending multiple copies to ensure firmware updates) in order to make secure firmware update. However, sending multiple copies is not realistic and imposes several constraints on the infrastructure. This proposal does not consider automotive on-board networks, where domains are traditionally separated, and due to functional and non-functional requirements. Furthermore, on-board key management issues are not mentioned in their approach. Kim et al. [78] present remote progressive updates for flash-based networked embedded systems. In their solution a link-time technique is proposed which reduces the energy consumption during installation. However, no security concern is addressed in this proposal.

Nilsson et al. discuss in [96, 98] provide a lightweight protocol and verification for secure firmware updates over the air (SFOTA). In the SFOTA protocol, different properties are ensured during firmware update protocol (i.e., data integrity, data confidentiality, and data freshness). However, this approach also relies on strong imposed assumptions in order to ensure the secure software upload: the authentication of the vehicle is not considered, keys are assumed to be stored securely and the authors use a single encryption key for all the ECUs in a car. Furthermore, no specific execution platform requirements are put forward by this proposal. In [97], key management issues are discussed in relation with software updates. A rekeying protocol is defined in order to distribute keys with only specific nodes in the group. It also uses a multicast approach to update the software on a group of node. However, we consider that different firmware's are installed on different ECUs, depending on the ECU functionalities, which makes multicast approach not useful. Furthermore, as mentioned above, this approach also does not consider execution platform requirements. It does not discuss about computation attacks, where the attacker can learn and modify the firmware, during the installation phase or simply prevent to update the counter, for later replay attacks.

Hagai [135] presents an approach that takes hardware into account by providing a secured runtime environment with a so-called Trust Zone on an ARM processor. In contrast the solutions of [9, 48] are software based. The so called *tools* and *enablers*, which are low-level and application-level security functions in [9] also cover a number of on-board automotive use-cases, while leaving the essential link to the external communication domain uncovered. The approach most closely related to our work is that of the Herstelle-Initiative Software – HIS [92]. The flashing process defined by the HIS provides a good basis for the OEMs, but the recommended protocol does not provide all the necessary security functionalities (i.e., freshness). Furthermore, this process only addresses hardwired firmware updates and does not provide any information about which key is used for firmware encryption, in a heterogeneous landscape of communication network technologies.

7.5 Conclusion

In this chapter, we have introduced the concept of ontology for cryptographic protocols in order to build the foundations for designing secure protocols in accordance with available security constructs and security classes. In particular, ontology makes it possible to analyze and design the cryptographic protocols by combining and binding system architecture, its security requirements, relationship with its security mechanisms, and available security services provided by different system assets. We have exemplified how the ontology can be used for building dedicated distributed embedded system protocols like a vehicular on-board firmware flashing cryptographic protocol specification. We showed how a root of trust in hardware could sensibly be combined with software modules such as PDM, EAM, and KMM. These modules and primitives have been applied to show how firmware flashing can be done securely. In contrast to existing approaches, the protocols presented in this chapter, describe a complete process, which involves the service provider, the vehicle infrastructure as well as the manufacturer and the service station. By using secure in-vehicle communication and a trusted platform model, we show how to establish a secure end-to-end link between the manufacturer, the workshop and the vehicle.

Chapter 8

Towards the Enforcement of Access Control Security Requirements

8.1 Introduction

The design and enforcement of security requirements, and in particular the access control related requirements, is central to securing automotive on-board networks from various attacks (i.e., RPC log-off function, intercept key data, etc.), but also a relatively very complex task. There already exist a few automotive-capable security solutions (detailed in Section 8.5.3): these solutions have traditionally been developed using standard security solutions in mind, where the vehicle is mostly considered as a single entity or only concerned with only the enforcement of access control are for protecting V2X communications. However, automotive on-board architectures do not only rely on the simple enforcement of security rules but also involve multiple enforcement points, especially when the underlying platforms and infrastructures are providing services themselves, like HSM, or middleware layers. Guided by our requirements engineering approach that specifies the security relevant automotive requirements to mitigate several attacks and vulnerabilities on these systems [80, 128, 52, 163] as well as by the specific needs and constraints, we extracted various types of access control requirements to analyze what kind of security policy is required as well as to decide about the appropriate enforcement points. We have identified the following set of security requirements whose expression must notably be enforced by the access control architecture:

- Authorization requirements (see Figure 6.4) that specify to which extent a certain entity is allowed to access and use a specific resource under a certain condition
- Authentication related access control requirements (see Figure 6.1) that define what level of authentication is required for corresponding role authorizations

The above requirements have to be coordinated, which should be reflected by a comprehensive security policy. For instance, the security policy to be applied in a vehicle is the combination of an invariant policy for the usage control of cryptographic credentials of electronic control units, and a flexible networking security

policy. The credential usage control policy is enforced by the HSM and possibly through the virtualization of the ECUs if applications on the same ECU have to be segregated. In contrast, the networking security policy is enforced by all network elements. Moreover, the access control architecture must also allow enforcing the rule to limit the traffic on the buses under consideration, based on trusted authentication or other security mechanisms like traffic filtering or secure logging. To simplify the authorization steps and to enforce these different sets of security policies in an on-board architecture, we need a system in which access control decisions are based on authenticated attributes of the subjects, and when the authorization authority is decentralized, in order to attain more fine grained access rights.

Unfortunately, there is currently no automotive-capable access control architecture, which not only needs to be extremely reliable and defect, but also extremely efficient for the enforcement of these security policies. To this end, we have defined and prototyped a security policy enforcement architecture for automotive on-board networks where security rules are enforced and handled at different layers of the architecture. However, the prerequisites of such an enforcement architecture are the knowledge of the communication buses and of the available computational capabilities of the on-board networks, more specifically, their ability to transmit, process, or store complex security policies. We show how the ASN.1 [67] specification that has been used for other purposes in the vehicle (i.e., low level drivers, RPC definition, HSM interfaces, etc.) can be employed in order to solve this particular issue.

In this chapter, we discuss how to design security policy engines that implement an effective enforcement in such heterogeneous automotive on-board networks. The novelty of our approach is in its enforcement of various access control rules, by deploying multiple policy enforcement points at the different levels of system abstraction. In particular, such a system must configure on one hand the mechanisms for handling all kind of authorization requests (i.e., management of keys for communication between ECUs, RPC level, and application level access control, etc.) and, on the other hand, the filtering mechanisms are enforced at the inter-domain gateway level in order to limit the unnecessary traffic. The configuration of security mechanisms (cf. deployment architecture presented in Chapter 7) in automotive on-board networks makes it necessary to define and deploy adapted security policies. In this context, we decided on building upon the XACML access control language. This language is suitable for the high level description of subjects, objects (or resources) and permissions on these. This language provides a flexible and modular way to define and enforce policies in distributed environments. By using XACML, we provide a reasonable interface towards new infrastructure and application services with W3C compliance. Although, XACML defines the policy language and a "request–response" message format, the use of the complete XACML policy specification is neither necessary nor even desirable for an automotive environment. Similarly, we could not afford the XML based security policy for reasons of message size constraints as well as footprint of an additional XML parser in the on-board system.

We thus evaluate how policies expressed in XACML can be adapted to the automotive environment efficiency requirements despite the limited computational power of those units and network bandwidth limitations. To this end, we propose an alternative interchangeable format, PDM Native Language (PNL) for XACML based security policies that is designed and implemented for compatibility with today's vehicular functional and non-functional needs. We further look at the performance analysis of our proposed PNL encoded policies with security policies specified in the XML format. We show that PNL encoded policy is lighter and enables a much faster parsing and configuration of security policies.

The remainder of this chapter is organized as follows: Section 8.2 outlines the security policy enforcement architecture, and in what way the security policy supports the flexible deployment and enforcement of the networking security policy. Section 8.3 discusses how XACML is being used to define the flexible part of such a security policy. In section 8.4, we present the security policy configuration process in order to install and configure access control policies with in the vehicle. Section 8.5 presents and analyzes performance figures for our policy engine. Finally, the chapter summarizes the results that we achieved regarding enforcement of access control security requirements.

8.2 Security Policy Enforcement Architecture

We first describe the structure of Policy Decision Module (PDM) that we adopted in chapter 7 in order to enforce various access control related security requirements. The PDM, which can be flexibly updated, is deployed within the automotive system at different levels of architecture layers. Considering the automotive networks and design specifications, we propose modular access control deployment architecture. A PDM which serves as a security policy engine is flexible deployable within the automotive system environment, which means, that a PDM could be used as a centralized module accessible from different domains and application within the automotive on-board network, or it could be deployed based on a multi-centered approach, or the module could be completely distributed within the system environment. Therefore, the PDM is applicable to the different requirements and constraints given by a specific on-board architecture and thus, can be deployed in different vehicle series over changing in-vehicular IT infrastructures. The PDM is composed of two main components (1) Policy Decision Point (PDP), following the terminology used in XACML, and (2) Policy Enforcement Points (PEPs). As a proof of concept, we have implemented PDM within the EMVY framework as a centralized module accessible from different domains and application. EMVY uses distributed master-client architecture and provide component-based templates for introducing security relevant mechanisms for securing communication between ECUs within a vehicle. All clients request services such as authorization request, entity authentication or key distribution, from the master in a "thin client" fashion, i.e., the service only

needs to be implemented on the master as shown in figure 8.1.

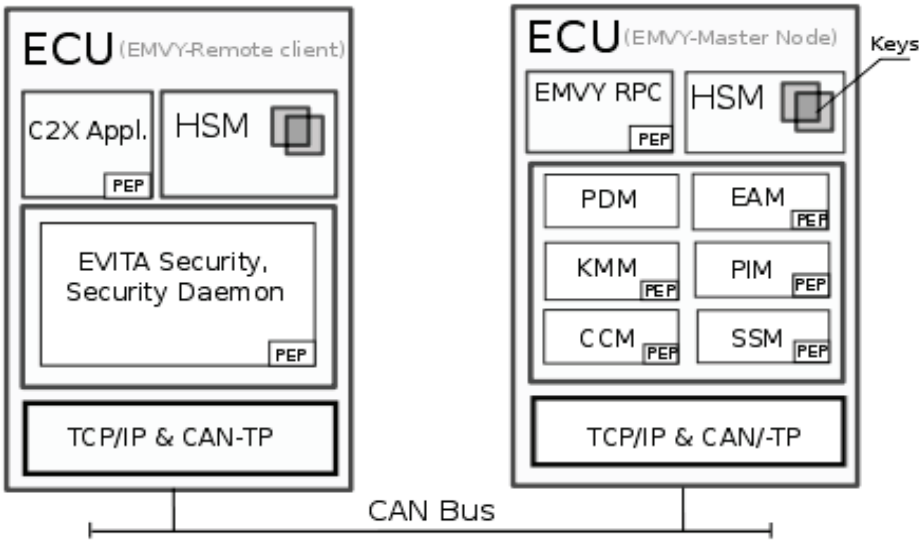


Figure 8.1: PDM and PEP deployment

8.2.1 Policy Enforcement Points

The PEP is the point, which receives the request from the requester and forwards it to the PDP. The PDP makes the decisions based on the policy that is a set for accessing that resource, while the PEP enforces the access control decision. Thus, PEPs handle access, or communication, or boot sequence validation requests for instance and have to enforce policy-based decisions. In order to enable a maximum level of flexibility and adaptability of the security functionality, we configure the PEP in a multi-layer fashion as a part of many security modules or programs that effectively enforce access rights of security-relevant resources, as shown in figure 8.1. In particular, this multi-layer enforcement architecture enforces security policies and rules in a distributed fashion, even though enforcement mechanisms used at each layer are different. All this means that we need some data structure to store security attributes computed by an enforcement point along the control flow, and that may be used to check the satisfaction of further policy rules. In particular, such a binding allow us to share security related information at the different architecture layers as well as to mitigate several attacks on these layers (cf. MLAM presented in Chapter 5). For instance, deciding on access control for some RPC request requires first allowing network traffic, at the CCM level, between the communicating parties, then the key credential usage control policy is enforced by the PEP deployed at the HSM in order to determine if the client allowed to communicate with other on-board clients and which ECUs the KMM component will distribute keys to in the

vehicle, then authorizing the communication channel using these keys, then finally making sure that the RPC request can be performed by the entity authenticated at the transport layer. Moreover, the filtering policies are enforced by the PEP defined at the CCM layer. The aim of those policies is to decide whether to forward messages or to drop them based on the authorization rules defined. Those rules will typically define patterns that have to be matched by the transport layer for the message to be forwarded (positive rules) or dropped (negative rules). Every CCM will contain a list of such rules that will be screened in an orderly fashion (so as to solve conflicts between positive and negative rules), looking for the first match. Rules apply to the incoming interface. The case where no match applies will result in the message being dropped, and a notification being sent to the intrusion detection system. Rules of such policies will be based mainly on transport-layer parameters (source and destination addresses, domain of origin or destination, etc.) but also on application-layer information, resulting for instance not only from the observation of a message (e.g., sending of a message from a sensor), but also on that part of its content (depending on the processing capabilities of the gateway), e.g. contained security features on application level.

Filtering may also be stateful, for instance relying on the past observation of a message, and even on contextual information, i.e., information about the vehicle and its environment that parameterize the policy but may be updated separately (e.g., vehicle speed, nearby vehicles, attack detected on another gateway, etc.). Filtering may therefore rely on plausibility checks. Contrary to filtering performed for traffic coming from some domains (e.g., the Head Unit (HU) domain), filtering on emergency messages should never completely prevent the transmission of potentially e-safety related messages, and only endpoints, i.e., applications, should ultimately decide about the validity of some information.

In addition, endpoint access control policies, enforced by PEP at the application level (i.e., C2X App shown in figure 8.1), on the other hand determine precisely whether an application can process certain messages based on its origin and on the definition of the authorizations of stakeholders. These are again positive rules expressing authorization granted to subjects. These policies are rather fine-grained. Most of these policies are about application permissions, that is, operations that can be performed by ECUs internally. However, some permission relate to stakeholder's rights, in particular with respect to the right to update firmware or parameters of ECUs. A detail description of rule enforcement used at each layer is discussed in Table 8.1.

8.2.2 Handling Policy Decisions

The PDP decides based on security policies whether or not access to a particular resource is granted. The decision of the PDP is then enforced by the policy PEP that drops a message, forwards a message or modifies a message (e.g., encrypts the

Security Module	Policy Enforcement Point (PEP)
Communication Control Module– CCM	enforce the filtering policies which act as firewall. Rule enforcement will be based mainly on transport-layer parameters but also on application-layer information.
Entity Authentication Module –EAM	enforce the policy based login and authentication services (e.g., password, smartcard).
Key Management Module –KMM	enforce the key generation and group communication security rules.
Platform Integrity Module – PIM	rule enforcement regarding valid boot integrity measurements.
Secure Storage Module – SSM	enforce the secure storage/access of data security rules (i.e. encrypt the storage device or, can encrypt data objects individually).
Security Watchdog Module – SWD	policy enforcement rely on a set of rules consists of a attack pattern and an action.

Table 8.1: Software security modules and Policy Enforcement Points (PEPs).

message with the cryptographic key of the destination ECU) according to the security policy. In our deployment architecture, a PDP usually acts autonomously in its domain where he is assigned and makes decisions in response to every authorization request. By default, PEP actively queries the PDP for every decision. However, PEP could be pre-configured by the PDM (autonomous PEP). Such an autonomous PEP would act as an ancillary PDP, mostly based on static or cached security decisions, and possibly parameterized decisions depending on security information that are available locally. For instance, during the secure bootstrapping phase, the SSM-PEP acts as an autonomous PEP and decides based on its local policy whether the PDP is allowed to load security policies.

8.3 Security Policy Expression

As we explained, the on-board network policy has to describe how to configure very different security mechanisms. After exploring several alternatives including drafting our own policy language, we decided on building upon the XACML [101] access control language, obviously for expressing the access control rules of our security policy, such as the definition of secure communication groups and related authorizations at the RPC level, but as well as a more general policy language. XACML provides a flexible and modular way to define and enforce policies, and its decision/enforcement model fits well in distributed environments, even for the on-board embedded system of a vehicle. XACML provides an interchangeable policy format, support for the fine-grained description of resources, can describe conditional rights, supports policy combination and conflict resolution. Another important aspect regarding the choice of XACML as our policy language was its independence from a specific implementation and the large number of tools for writing and analyzing any policy. In the case of specific configurations, XACML is flexible enough to represent

different security profiles. For instance, the XACML `Policy` element can be used in order to encapsulate complex firewall rules comprising multiple attributes, the source IP address/port numbers being specified in the XACML `Subject` element for instance, and the destination IP address/port number being mapped to the XACML `Resource` element. In a similar way, the XACML `Rule` element can be used to represent distinct firewall rules. However, due to the embedded nature of the on-board system and its functional and non-functional constraints [94], [74], it is not feasible to transmit, process, or store XML-based policies in the car. For instance, for the ubiquitous CAN bus, which is operated at around 500kbit/s and offers 8 bytes of data payload per packet, verbose formats like XML would constitute a hardly justifiable increase of the bus load.

To cope with the above-mentioned limitations, we defined a binary-based security policy language that consumes less bandwidth, is fast to process, and requires less memory (see section 8.5 for a performance analysis). We called this representation the Policy decision module Native Language (PNL). The purpose of the PNL is not to define yet another access control policy language but rather to provide an alternative interchangeable format for XACML policies, that can be used where performance is an issue. We built PNL on ASN.1 standards [67]. These standards are adopted in a wide range of application domains, as in aviation systems for traffic control, mobile networks, network management, secure emails, fast web services, etc., [67]. PNL makes use of these standards, describes a serialized representation of XACML policies in binary format, and ensures that the XACML structure is preserved during serialization. In order to do so, a XACML schema is mapped into a corresponding ASN.1 definition (see listing D.1). This mapping is based on the ITU-T X.694 standard (Mapping from XML Schemas to ASN.1 modules) [66].

```
1  XACML DEFINITIONS AUTOMATIC TAGS ::= BEGIN
2  /* XACML Policy Defintion */
3  PolicyType ::= SEQUENCE {
4    ...
5    ruleCombiningAlgId UTF8String,
6    target Target,
7    choice-list SEQUENCE OF CHOICE {
8    ...
9    rule Rule
10  } OPTIONAL,
11  obligations Obligations OPTIONAL
12 }
13 Policy ::= PolicyType
14 /* XACML Target Definition */
15 TargetType ::= SEQUENCE {
16   subjects  Subjects OPTIONAL,
17   resources  Resources OPTIONAL,
18   actions   Actions OPTIONAL,
19   environments  Environments OPTIONAL
20 }
21 Target ::= TargetType
22 ...
23 /* XACML Rule Defintion */
24 RuleType ::= SEQUENCE {
```

```
25    effect  EffectType,
26    ruleId  UTF8String,
27    description  Description OPTIONAL,
28    target  Target OPTIONAL,
29    condition Condition OPTIONAL
30 }
31 ...
32 Rule ::= RuleType
33 AttributeAssignmentType ::= SEQUENCE {
34     attributeId [0] UTF8String,
35     dataType [1] UTF8String,
36     attr [2] SEQUENCE OF UTF8String
37 }
38
39 AttributeAssignment ::= AttributeAssignmentType
40
41 ObligationType ::= SEQUENCE {
42     fulfillOn [0] EffectType,
43     obligationId [1] UTF8String,
44     attributeAssignment-list [2] SEQUENCE OF attributeAssignment
45        AttributeAssignment
46 }
47 ...
48 END
```

Listing 8.1: Excerpt of a Policy decision module Native Language (PNL) based on ASN.1 Defintion.

We have implemented the XACML to PNL mapping engine as shown in figure 8.2. In our architecture, this encoder resides at the OEM's backend system. The mapping engine is responsible for serializing security policies into the PNL format and then transmitting it to the vehicle. During the serialization process each security policy is verified against a XACML schema. Upon a successful validation, the security policy is serialized into a specific ASN.1 encoding scheme. Due to constraints from other components (e.g., low level drivers, HSM interfaces, etc.), which employ the ASN.1 DER encoding [65], the security policies are also serialized using DER encoding rules. However, a more efficient binary encoding such as Packed Encoding Rules (PER) can be used if needed. We anticipate that using PER encoding scheme would further enhance performance and latency results.

8.4 Security Policy Configuration

Security policies are the basic for all authorizations within the vehicular system. Hence, mechanisms for creating, updating, and configuring policies are very important. We are considering security policy update protocols [59], follows the similar steps like defined in the OTA firmware update protocol, to be used for securely transmitting new or updated security policies in the vehicle. Whereas, from the on-board PNL configuration perspective, our high-level goal is to auto-configure security policies and appoint access rules from the vehicle ignition stage. This implies that security policy are deserialized during secure bootstrapping phase, while also

recognizing an explicit configuration procedures, after secure bootstrapping phase, in order to load new/update policies (i.e., installation of new application with its policy set or update existing policy). Note that the new policy set or security rules, which may have an impact on the basic safety of the vehicle, are always configured during next vehicle start cycle. Adding/updating safety critical rules while vehicle is running may cause safety critical problems, depending on the security policy responsible for.

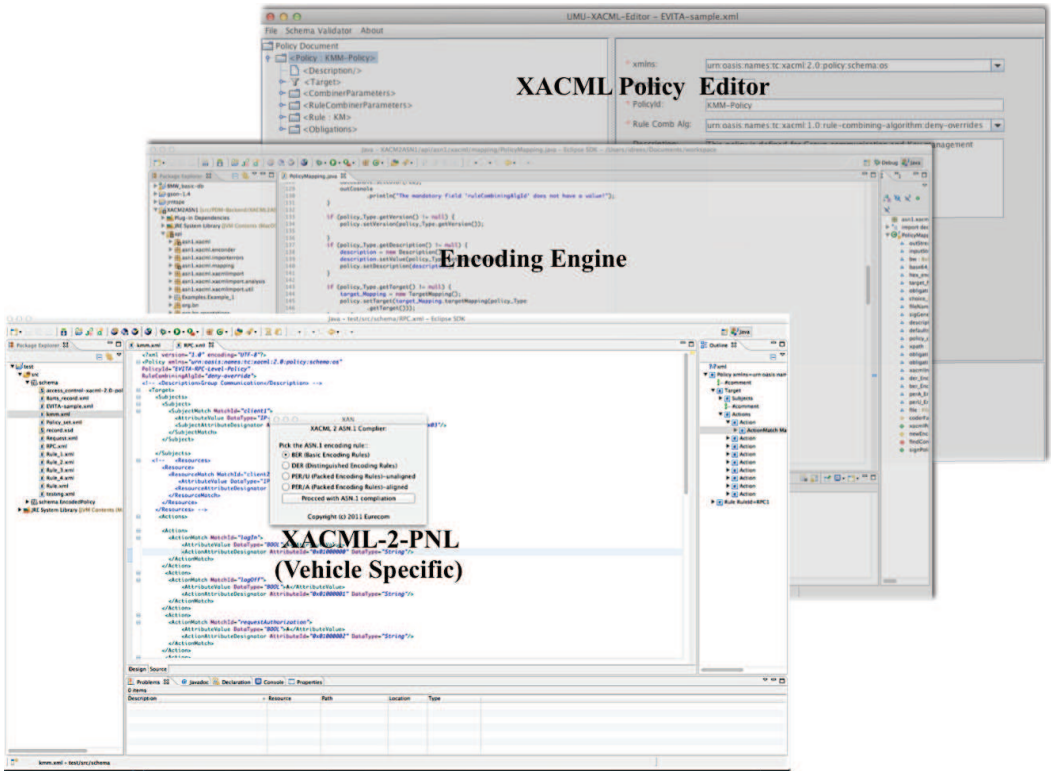


Figure 8.2: XACML to PNL mapping engine

In our implementation stack, the bootstrapping protocols are defined to ensure the secure initialization of all security modules and components (see figure 8.1). On a certain point of the boot strapping procedure, the boot chain send an initialization call to the PDP(s) to load all PNL based security policies (i.e., group communication policy) from its policy database. Note that these security policies are stored in the Secure Storage Module (SSM), which enforces confidentiality, integrity, authenticity, and freshness mechanisms. Thus, require proper authentication and access rights to access these policies. Since during boot strapping process, SSM cannot ask PDP for a decision when PDP is opens/reads from its policy database (which is also stored via SSM), it has to make autonomous decisions. A detail description of autonomous decisions (autonomous PEP) is discussed in section 8.2.1.

On successful verification of access rights, the SSM allow the PDP(s) to read policy set from its policy database. Regarding the integrity of policies, we rely on the security solutions enforced by SSM. However for the policy validation attempt in our prototype implementation, we enforced, that a vehicle will only be started in case of successful configuration of all PDP/autonomous PEP(s) with respective security policies and the vehicle will shutdown momentarily otherwise.

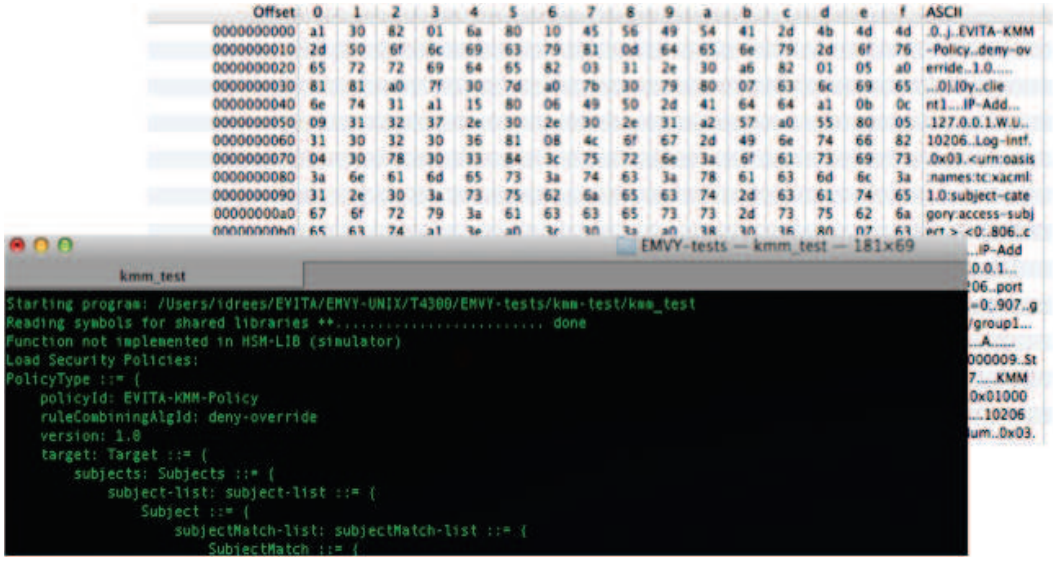


Figure 8.3: PDM: On-board policy deserialization and configuration

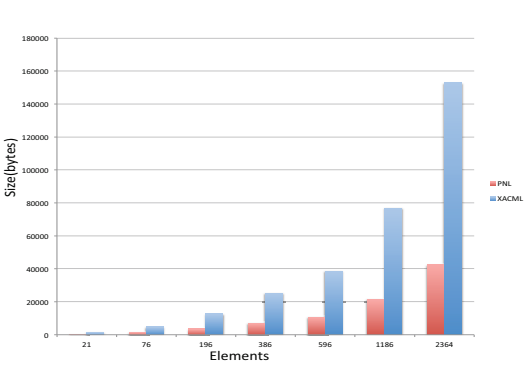
8.5 Performance Analysis

We have analyzed the performance of the PNL as well as the memory consumption of different policies by varying the number of elements and attributes used. Performance has a special importance with respect to the user experience in automotive environments, where the time to load and configure security policies, and to assess authorization request and response time is a critical issue when the driver waits for his vehicle to start. The deserialization and configuration of authorization/security policies must be performed before receiving any request from PEP. Hence, the configuration of these security policies is significantly contributing to the overall responsiveness of the policy decision and enforcement at startup time.

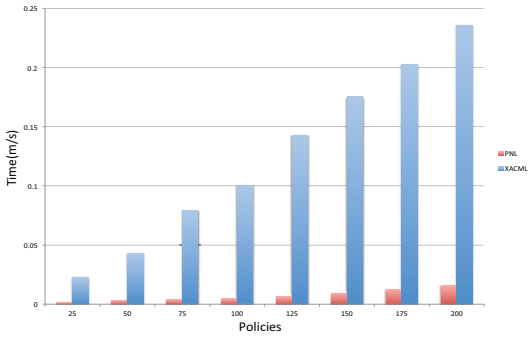
8.5.1 Performance Analysis: Technical Approach

In order to evaluate our results against other XML based access control policies, we are comparing our results with different XML parsers. There have been numerous benchmark studies towards the evaluation of XML parsers [72, 19, 46, 75, 165, 132].

These benchmarks cover several aspects of XML parsing such as performance, schema validation, DOM manipulation, XML security, etc. We used these benchmarks in order to select an agile XML parser implemented in the same language as our own (C/C++), and to run a comparison with our ASN.1 encoded policies. Several lightweight C/C++ XML libraries [72, 75, 49] have been developed for low power devices, with fast parsing capabilities. For instance, the pugixml library enables extremely fast, convenient, and memory-efficient XML document processing. It consists of a DOM-like interface with rich traversal and modification capabilities. However, since pugixml has a DOM parser, it cannot process XML documents that do not fit in memory; also the parser is not a schema validating parser [72], which is a mandatory requirement in our case, for obvious security as well as safety reasons, such as to prevent software compromises.



(a). Comparison of XML and PNL security policies, with different elements and attributes, in terms of size (bytes) of policy data



(b). Speed up analysis of about 25-200 XML and PNL security policies, where single security policy is constitute of 21 XACML Policy Elements.

Figure 8.4: Size of data and increase in speed up factor.

8.5.2 Experimental Setup and Results

Based on results of these benchmarks and these specific constraints, we decided to evaluate our results with the Gnome XML toolkit (libxml) [154]. According to the XMLBench Project [19], libxml is the fastest toolkit that has a rich enough set of features. We have generated several XACML policies with various sizes using the UMU XACML editor. The policy tickets are than transformed into a corresponding PNL description, as described in section 8.3. We have set up a testing environment in order to compare the scalability of our parser with libxml. All presented performance results were obtained using a 64 bit Mac OS X 10.7.2 on a MacBook Pro with 8GB of RAM and a 2.8GHz Intel Core i7 processor. All tests were run in a single user mode without any system services running. We followed the assumptions outlined in [19]: for instance, the time spent to initialize the toolkits is not counted in these results, in order to compare our results with this benchmark. The measurements

consist in several latency results which show that the parsing of a PNL encoded policy is lighter and enables a much faster parsing and configuration of security policies. Figure 8.4.a compares the parsing time with different policy sizes, in which the PNL encoded policy parsing is approximately a third of the XML policy. For 2364 elements (2898 attributes) in a single XACML policy, the PNL encoded policy amounts to 42,368 bytes and the XML encoded one to 152,817 bytes. In the eight tests presented on Figure 8.4.b, we parsed a single XML policy (21 Elements, 22 Attributes) up to 200 times to understand the scalability. This resulted in parsing times of 0.25 ms. In contrast, deserializing using the PNL encoded security policy 200 times takes approximately 0.015 ms. Parsing with the PNL encoding is thus about 10 times faster that with an XML based encoding.

8.5.3 The State of the Art: Automotive Access Control Architecture

There has been a quite remarkable progress in the area of access control architecture for automotive networks [32, 16, 116], but they appear to have succeeded mostly in terms of requirement specification or have been only concerned with security policies for protecting V2X communications. Gerlach et al. [37] present a C2C communication solution integrating several previous proposals [124, 123, 109] for secure vehicular communications [16]. These proposals consider the car mostly as a single entity, communicating with other cars using secure protocols and thus essentially aim at communication specific security policies enforced at the Communication Unit of the vehicle. Our approach in contrast treats both the expression of V2X and intra-vehicle security policies uniformly. The EASIS project [32] defines a central gateway, a sort of firewall, that is configured so that it denies all data traffic from the external interfaces (e.g. C2C/C2I or Telematics) as a default. Unfortunately, like [16] this proposal is also limited to V2X security policy enforcement and not accompanied by any further analysis of the particular requirements/limitation of an in-vehicle architecture with respect to security policies. Zrelli et al. [168] proposed a security framework for the vehicular communication infrastructure implementing access control at both the data link layer and the network layer. However, the proposed solution is solely based on a central policy decision and enforcement module. A single failure in this module may compromise the overall security of the on-board network. Furthermore, this solution is obviously only handling V2X security policy enforcement at the gateway level.

For in-vehicle architectures, numerous authors mention the need for on-board access control architecture [80, 119, 9]. However, very few solutions have been proposed. A recent security analysis [80] has shown that the risk of attacks on vehicle on-board systems is not anymore of theoretic nature. It depicts several scenarios where access control is either weak or simply not considered, like the firmware update process, which may compromise the overall security of the on-board network, yet

no security architecture is described in this work. In [9], a set of cryptographic protocols is discussed to support vehicular use-cases. However, regarding access control, this intra-vehicle security toolbox is also limited to the only specification of an API, without any detail about the policy decision engine, practical matters regarding the enforcement architecture, nor implementation perspectives.

Chutorash et al. [22] propose an approach for integrating firewalls in a vehicle communication bus. Firewalls are integrated between application software and between vehicle components. In their approach, filtering rules are applied only on user's request, and commands sent from the HMI, preventing unauthorized access to vehicle components. We see that the practicality of this approach is largely limited by the fact that: (1) rule enforcement is limited to firewall rules and more specifically only to user commands. However, in an automotive system, different entities (i.e., security modules as discussed in chapter 7) are themselves requesters (2) rules are statically defined and remain the same over the vehicle lifetime, and (3) the constraints of embedded vehicular networks regarding notably the policy transmission, processing, or storage and the practical implementation of the proposed approach, are again left out. The OVERSEE Project [119] aims among other objectives at in-vehicle firewall configuration and application level access control using XML based configuration rules. As of now, this project has just begun and no result is available, thus we cannot evaluate the practicality of this approach. However, according to our experience, the performance of verbose formats like XML in a constrained environment has to be closely watched.

8.6 Conclusion

We have exposed in this chapter in what respect the complexity of automotive on-board network architectures and their evolution involve a complex expression of the access control security requirements and their enforcement. Today's automobiles are a perfect example of a system whose security relies on the combination of many different enforcement points in the automotive on-board network and even in every electronic control unit's communication stack. The role of access control security requirements in this context is to link the trusted computing base and the trusted credentials it stores with enforcement mechanisms, as well as to connect enforcement mechanisms together. We are taking advantage of the extensibility of XACML subjects to associate attributes, and in particular the means to perform a trusted authentication of electronic control units: this mechanism is at the core of the EVITA approach. The performance of the policy parsing is also very important in a vehicle. We described how the XACML policy could be encoded in ASN.1 in order to make it fit better the resource-constrained environment of a vehicle. The policy engine described in this chapter was finally deployed in the EVITA project demonstrator - two cars equipped with the EVITA HSMs and software framework - and was successfully used for network filtering, configuring secure group communication, and RPC level access control.

Chapter 9

Conclusions and Future Perspectives

This dissertation deals about requirements engineering driven approach to security architecture design for distributed embedded systems. Various approaches and techniques are tackled with the wish to design the secure system from the early stages of system conceptualization.

The first part of this study introduced the notion of knowledge centric security requirement engineering methodology. An overview of existing security requirement engineering approaches and methodologies is provided and their techniques are compared to describe their strengths and weaknesses with respect to security requirement elicitation process. As a solution to overcome the shortcomings and limitations in the state of the art approaches to SRE, we have defined an unified methodology based on the concepts of ontologies which offers means to combine different capabilities of these models in a single unified security requirement engineering process. Different concepts and terms identified during the analysis were then adapted to various security ontology classes such as security requirements, security goals, security attacks, etc. In order to well integrate our approach with standard system engineering activities, we decided to use SysML as an underlying modeling language to model our different security requirement engineering concepts. In this regard, we have proposed several extensions to the SysML semantic to integrate our security concepts and developed a SysMLsec profile. The objective of this profile is to combine different modeling diagrams that help in building security solutions. The novelty of this extension is in its integration of concepts and terms from security ontologies in the SysML diagrams as a controlled vocabulary. Thus, gives the security engineers the freedom to choose and reason about appropriate ontological concepts, provided that the SysML diagrams have the sufficient semantics to support the detailed ontological concepts.

The second part of this thesis describes the different solutions proposed to build the security architecture design for embedded systems. The first solution relies on the usage of knowledge based security analysis to identify the security attacks and vulnerabilities in the context of multilayered embedded system architecture. Furthermore, the concept of knowledge oriented attack trees, parameterized by the

ontological concepts, was brought in as the foundational graphical representation security attacks and vulnerabilities. Which makes it possible to graphically represent various types of attack related metadata such as attack type, attack method, adversary capabilities, etc. The second solution exposes a security requirement engineering based solution for the identification, the refinement, and the traceability of system wide security requirements that tend to overcome shortcoming in state of the art security requirement engineering approaches. This solution introduces the new concept of dependent requirement engineering that is used for an identification and refinement of security requirements in relation to their source(s). Furthermore, we also discussed the traceability property of security requirements and proposed a very simple solution to link different models by using the reference principle.

In the third and last part of this thesis focused on the enforcement of security requirements by developing cryptographic protocols and development of access control architecture for automotive embedded system. We proposed a firmware flashing cryptographic protocol for securely updating firmware in the vehicle. We showed how a root of trust in hardware could sensibly be combined with software modules. These modules and primitives have been applied to show how firmware updates can be done securely and over-the-air, while respecting existing standards and infrastructure. In contrast to existing approaches, the firmware flashing protocol describe a complete process, which involves the service provider, the vehicle infrastructure as well as the manufacturer and the workshop. By using secure in-vehicle communication and a trusted platform model, we showed how to establish a secure end-to-end link between the manufacturer, the workshop and the vehicle. We finally presented our solution to enforce access control related security requirements in the complex automotive architecture. We proposed how to design policy engines that implement an effective enforcement in such architectures despite the complexity of the protocol stacks of on-board electronic control units.

We reckon that this thesis is only an extensive snapshot of our work. We plan to continue with some of the threads of future work that we identified in the different chapters. Specifically, an interesting issue that needs further research is embedding the SysMLsec within a formal logic, formal methods can be further facilitate us to formally verify security aspects defined in these models as well as to maintain consistency as a design evolves. To do so, we need to enlarge the semantics and the expression of the security ontologies as well as SysMLsec.

Further, we plan to evaluate the knowledge-centric security requirement engineering methodology with further case studies. Most importantly, we are interested in case studies from other application domains, and also at different stages of systems development. Based on these, we plan to iteratively refine the concepts of the security ontologies, with the prospect of developing security architecture.

Bibliography

[1] ISO/IEC 15408-2:2008. Information technology – Security techniques – Evaluation criteria for IT security – Part 2: Security Functional Requirements. Available at `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=46414`, 2009. 27, 38

[2] ISO/IEC 27002:2005. Information technology – Security techniques – Code of practice for information security management. Available at `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=50297`, 2005. 25, 33

[3] ISO/IEC/IEEE 42010:2011. Systems and Software Engineering – Architecture Description. Available at `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=50508`, 2007. xi, 29, 31

[4] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008. 9

[5] L. Apvrille. TTool for DIPLODOCUS: An Environment for Design Space Exploration. *In Proc. of the 8th International Conference on New Technologies in Distributed Systems*, 2008. 81, 83

[6] L. Apvrille, J-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. *IEEE Transactions on Software Engineering*, pages 473–487, 2004. 46

[7] L. Apvrille, A. Mifdaoui, and P. de Saqui-Sannes. Real-Time Distributed Systems Dimensioning and Validation: The TURTLE Method. *Studia Informatica Universalis*, pages 47–69, 2010. 46

[8] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. A UML-based Environment for System Design Space Exploration. In *In 13th IEEE International Conference on Electronics, Circuits and Systems*, ICECS'06, pages 1272–1275, 2006. 46

[9] H. Bar-El. Intra-Vehicle Information Security Framework. In *7th Workshop on Embedded Security in Cars*, ESCAR'09, 2009. 1, 136, 148, 149, 193

[10] C. Basile, J. Silvestro, A. Lioy, D. Canavese, M. Arrigoni Neri, S. Paraboschi amd M. Verdicchio, M. Casalino, and T. Scholte. Security Ontology Definition. Technical Report D3.2, PoSecCO Project, 2011. 21

[11] D. Berardi, A. Cali, D. Calvanese, and G. Di Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence*, 168, 2003. 56

[12] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge Representation as the Basis for Requirements Specifications. *IEEE Computer Society Press - Computer*, pages 82–91, April 1985. 20

[13] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agents and Multi-Agents System*, 2004. 12

[14] R. R. Brooks, S. Sander, J. Deng, and J. Taiber. Automobile Security Concerns. *IEEE Vehicular Technology Magazine*, pages 52–64, June 2009. 2, 194

[15] M. Busse and M. Pleil. Data Exchange Concepts for Gateways. Technical Report D1.2-10, EASIS Project, 2006. 128

[16] C2C-CC. Car2Car Communication Consortium. `http://www.car-to-car.org/`. 135, 148

[17] S. Ahmet Camtepe and B. Yener. Modeling and Detection of Complex Attacks. *In 3rd International Conference on Security and Privacy in Communications Networks, SecureComm*, September 2007. 54

[18] CAPEC. Common Attack Pattren Enumeration and Classification. `http://capec.mitre.org/`. 33

[19] S. Chilingaryan. The XMLBench Project: Comparison of Fast, Multi-platform XML libraries. In *Database Systems for Advanced Applications*, pages 21–34. Springer-Verlag, Berlin, Heidelberg, 2009. 146, 147

[20] T. Christian. Security Requirements Reusability and the SQUARE Methodology. Technical Report CMU/SEI-2010-TN-027, Software Engineering Institute, Carnegie Mellon University, September 2010. 23, 24

[21] L. Chung, P. Leite, and J. Cesar. On Non-Functional Requirements in Software Engineering. In *Conceptual Modeling: Foundations and Applications*, pages 363–379, 2009. 29

[22] R . J. Chutorash. Firewall for Vehicle Communication Bus, WO/2000/009363, PCT/US1999/017852. *International Patent Classification 7 - European Patent Office*, Feb 2000. 149

[23] S. Cranefield. UML and the Semantic Web. *In Proc. of the International Semantic Web Working Symposium*, 2001. 20

[24] CVE. Common Vulnerabilities and Exposures. `http://cve.mitre.org/`. 33

[25] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed Requirements Acquisition. *Selected Papers of the Sixth International Workshop on Software Specification and Design*, pages 3–50, 1993. 24

[26] R. Darimont and A. van Lamsweerde. Formal Refinement Patterns for Goal-driven Requirements Elaboration. In *Proc. of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT'96, pages 179–190, 1996. 11

[27] M. Dean and G. Schreiber. OWL Web Ontology Language Reference. W3C Recommendation, W3C, Feb 2004. 28

[28] B. Decker, E. Ras, J. Rech, B. Klein, and C. Hoecht. Self-organized Reuse of Software Engineering Knowledge supported by Semantic Wikis. In *Workshop on Semantic Web Enabled Software Engineering*, SWESE'05, 2005. 21

[29] D. Dolev and A. C. Yao. On the Security of Public key Protocols. Technical report, Stanford University, Available at `ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/81/854/CS-TR-81-854.pdf`, 1981. 26

[30] A. Ekelhart, S. Fenz, M. Klemen, and E. Weippl. Security Ontologies: Improving Quantitative Risk Analysis. In *40th Annual Hawaii International Conference on System Sciences*, HICSS'07, page 156a, Jan 2007. 21, 63

[31] R. Ernst. Hardware/Software Co-Design of Embedded Systems. In *Asia Pacific Conference on Computer Hardware Description Languages*, APCHDL'97, August 1997. 79

[32] T. Eymann and M. Busse. Security and Firewall concepts for Gateways. Technical Report Deliverable D1.2-12, EASIS Project, 2006. 148

[33] S. Fenz and A. Ekelhart. Formalizing Information Security Knowledge. In *Proc. of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS'09, pages 183–194, 2009. 21

[34] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL- A language for deductive query answering on the Semantic Web. *Journal Web Semantics: Science, Services and Agents on the World Wide Web*, pages 19–29, December 2004. 63

[35] D. George Firesmith. A Taxonomy of Security-Related Requirements. In *International Workshop on High Assurance Systems*, RHAS'05, 2005. 21

[36] D. Gaševic, N. Kaviani, and M. Milanovic. Ontologies and Software Engineering. In *Handbook on Ontologies*, pages 593–615. Springer, 2009. 20

[37] M. Gerlach, A. Festag, T. Leinmüller, G. Goldacker, and C. Harsch. Security Architecture for Vehicular Communication. In *2nd International Workshop on Intelligent Transportation*, WIT'05, 2005. 148

[38] P. Giorgini, H. Mouratidis, and N. Zannone. Modelling Security and Trust with Secure Tropos. In *Integrating Security and Software Engineering: Advances and Future Visions*, 2006. 101

[39] H. Graves. Integrating SysML and OWL. In *OWL Experiences and Directions October Workshop*, OWLED'09, October 2009. 56

[40] H. Graves. Ontological foundations for SysML. In *Proc. of 3rd International Conference on Model-Based Systems Engineering*, MBSE'10, September 2010. 56

[41] H. Graves and A. Associates. Integrating Reasoning with SysML. *A Journal On The Theory Of Ordered Sets And Its Applications*, 2012. 56, 57

[42] Trusted Computing Group. Trusted Platform Module Specifications. Available at `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2007. 127

[43] M. Grüninger and M. S. Fox. The Design and Evaluation of Ontologies for Enterprise Engineering. *Workshop on Implemented Ontologies, European Conference on Artificial Intelligence*, 1994. 20

[44] V. Haarslev and R. Moller. RACER: An OWL Reasoning Agent for the Semantic Web. In *1st International Workkshop on Applications, Products and Services of Web-based Support Systems*, WCC'03, pages 91–95, 2003. 63

[45] C. B. Haley, R. Laney, Jonathan D. Moffett, and B. Nuseibeh. Security Requirements Engineering: A Framework for Representation and Analysis. *IEEE Transactions on Software Engineering*, pages 133–153, January 2008. 9, 12

[46] S. Cheng Haw and G. S. V. Radha Krishna Rao. A Comparative Study and Benchmarking on XML Parsers. In *9th International Conference on Advanced Communication Technology*, pages 321–325, February 2007. 146

[47] W. Heaven and A. Finkelstein. A UML profile to support requirements engineering with KAOS. In *IEE Proceedings - Software*, pages 10–27, February 2004. 107, 109

[48] A. Hergenhan and G. Heiser. Operating Systems Technology for Converged ECUs. In *6th Workshop on Embedded Security in Cars*, ESCAR'08, November 2008. 136

[49] J Higgins. Arabica XML and HTML Processing Toolkit. Available at `http://www.jezuk.co.uk/cgi-bin/view/arabica`. 147

[50] J. Holt and S. Perry. *SysML for System Engineering (Professional Applications of Computing)*, volume 7. IET, 2007. 23, 44

[51] T. Hoppe, S. Kiltz, and J. Dittmann. Security Threats to Automotive CAN Networks - Practical Examples and Selected Short-Term Countermeasures. In *Proc. of the 27th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'08, pages 235–248, 2008. 2, 194

[52] T. Hoppe, S. Kiltz, and J. Dittmann. Automotive IT-Security as a Challenge: Basic Attacks from the Black Box Perspective on the Example of Privacy Threats. In *Proc. of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'09, pages 145–158, 2009. 137

[53] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Available at `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`, May 2004. 63

[54] S. Hilde Houmb, S. Islam, E. Knauss, J. Jürjens, and K. Schneider. Eliciting Security Requirements and Tracing them to Design: An Integration of Common Criteria, Heuristics, and UMLsec. *Journal of Requirements Engineering - Special Issue on RE'09: Security Requirements Engineering*, pages 63–93, March 2010. xi, 14, 101

[55] D. D. Hwang, P. Schaumont, K. Tiri, and I. Verbauwhede. Securing Embedded Systems. *IEEE Security & Privacy*, pages 40–49, April 2006. 3, 194

[56] IBM. Rational Rhapsody Developer. Available at `http://www-01.ibm.com/software/rational/products/rhapsody/developer/`. 46

[57] M. Sabir Idrees, Y. Roudier, and L. Apvrille. A Framework Towards the Efficient Identification and Modeling of Security Requirements. In *5th Conf. on Network Architectures and Information Systems Security*, SAR-SSI'10, May 2010. 107

[58] M. Sabir Idrees, Y. Roudier, L. Apvrille, and G. Pedroza. Test Results. Technical Report D4.4.2, EVITA Project, 2011. 87, 109

[59] M. Sabir Idrees, Y. Roudier, H. Schweppe, B. Weyl, R. E. Khayari, O. Henniger, D. Scheuermann, G. Pedroza, L. Apvrille, H. Seudie, H. Platzdasch, and M. Sall. Secure On-Board Protocols Specification. Technical Report D3.3, EVITA Project, 2010. 128, 131, 144

[60] M. Sabir Idrees, G. Serme, Y. Roudier, A. Santana De Oliveira, H. Grall, and M. Sudholt. Evolving Security Requirements in Multi-layered Service-Oriented-Architectures. In *4th International Workshop on Autonomous and Spontaneous Security*, SETOP'11, September 2011. 4, 87, 196

[61] IEEE. *IEEE Guide to Software Requirements Specifications, ANSI/IEEE Standard 830-1998*, 1998. 27

[62] ISO/IEC-15408:2009. Information technology – Security techniques – Evaluation Criteria for IT security – Part 1: Introduction and General Model. Available at `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=50341`, 2009. 9, 24, 29, 33, 34, 36, 107

[63] ISO/IEC-21827:2008. Information technology – Security techniques – Systems Security Engineering – Capability Maturity Model SSE-CMM. Available at `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=44716`, 2008. 25, 33

[64] ISO/IEC-27000:2012. Information technology – Security techniques – Information security management systems – Overview and vocabulary. Available at `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=56891`, 2009. 24, 32, 33

[65] ITU-T. Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), ITU-T Recommendation X.690. Available at `http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf`, 2002. 109, 144

[66] ITU-T. Information Technology - ASN.1 encoding rules: Mapping W3C XML schema definitions into ASN.1, ITU-T Recommendation X.694. Available at `http://www.itu.int/ITU-T/studygroups/com17/languages/X694.pdf`, 2004. 143

[67] ITU-T. Information Technology - ASN.1 encoding rules: Abstract Syntax Notation one (ASN.1): Specification of basic notation, ITU-T Recommendation X.680. Available at `http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf`, 2008. 138, 143

[68] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Professional, 2001. 16, 18

[69] J. Jüerjens. *Secure Systems Development with UML*. Springer, 2003. 2, 9, 13, 41, 107, 194

[70] J. Jürjens. Towards development of secure systems using UMLSec. *4th International Conference on Fundamental Approaches to Software Enineering*, pages 32–42, 2001. 25, 109

[71] J. Jürjens. Using UMLsec and Goal Trees for Secure Systems Development. In *Proc. of the 2002 ACM symposium on Applied computing*, SAC'02, pages 1026–1030, 2002. 13

[72] A. Kapoulkine. Pugixml Benchmark. Available at `http://pugixml.org/benchmark/`. 146, 147

[73] M. Karyda, T. Balopoulos, L. Gymnopoulos, S. Kokolakis, C. Lambrinoudakis, S. Gritzalis, and S. Dritsas. An Ontology for Secure e-Government Applications. In *Proc. of the First International Conference on Availability, Reliability and Security*, ARES'06, pages 1033–1037, 2006. 21

[74] E. Kelling, M. Friedewald, T. Leimbach, M. Menzel, P.Säger, H. Seudié, and B. Weyl. Specification and Evaluation of e-Security Relevant Use cases. Technical Report D2.1, EVITA Project, 2009. xi, 69, 70, 143

[75] M. Kerbiquet. Asm-XML Benchmark. Available at `http://tibleiz.net/asm-xml/benchmark.html`. 146, 147

[76] B. Kienhuis, Ed F. Deprettere, P. van Der Wolf, and K. Vissers. A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*, SAMOS, pages 18–37, 2002. 79, 80

[77] A. Kim, J. Luo, and M. Kang. Security Ontology for Annotating Resources. In *Proc. of the 2005 OTM Confederated international conference on the Move to Meaningful Internet Systems: CoopIS, COA, and ODBASE*, OTM'05, pages 1483–1499, 2005. xii, 21, 63, 122, 123

[78] J. Kim and P. H. Chou. Remote Progressive Firmware Update for Flash-based Networked Embedded Systems. In *Proc. of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED'09, pages 407–412, 2009. 135

[79] P. Kocher, R. Lee, G. Mcgraw, and S. Ravi. Security as a New Dimension in Embedded System Design. In *Proc. of the 41st Design Automation Conference*, DAC'04, pages 753–760, 2004. 3, 194

[80] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *IEEE Symposium on Security & Privacy*, pages 447–462, may 2010. 1, 2, 137, 148, 193, 194

[81] A. Kott and J. Peasant. Representation And Management Of Requirements: The Rapid-Ws Project. In *Concurrent Engineering Research and Applications*, pages 93–106, June 1995. 20

[82] TELECOM ParisTech LabSoc. The TURTLE Toolkit - TTool. Available at `http://labsoc.comelec.enst.fr/turtle/ttool.html`. 4, 46, 196

[83] R. Laleau, F. Semmak, A. Matoussi, D. Petit, A. Hammad, and B. Tatibouet. A First attempt to combine SysML Requirements Diagrams and B. *Innovations in Systems and Software Engineering*, pages 47–54, 2010. 56

[84] J. Lin, M. S. Fox, and T. Bilgic. A Requirement Ontology for Engineering Design. *Concurrent Engineering: Research and Applications*, pages 279–291, September 1996. 20

[85] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Analysing Security Threats and Vulnerabilities Using Abuse Frames. Technical Report 2003/10, Department of Computing, The Open University, October 2003. xi, 16

[86] L. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Introducing Abuse Frames for Analysing Security Requirements. In *Proc. of the 11th IEEE International Conference on Requirements Engineering*, RE'03, pages 371–372, September 2003. 9, 16, 41

[87] L. Liu, E. Yu, and J. Mylopoulos. Security and Privacy Requirements Analysis within a Social Setting. In *Proc. of the 11th IEEE International Requirements Engineering Conference*, RE'03, pages 151–161, September 2003. 107

[88] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proc. of the 5th International Conference on The Unified Modeling Language*, UML'02, pages 426–441, 2002. xi, 9, 14, 15, 25, 41, 101, 107, 109

[89] S. Masud Mahmud, S. Shanker, and I. Hossain. Secure Software Upload in an Intelligent Vehicle via Wireless Communication Links. In *IEEE Intelligent Vehicles Symposium*, pages 588–593, 2005. 135

[90] N. Mayer, A. Rifaut, and E. Dubois. Towards a Risk-Based Security Requirements Engineering Framework. In *Proc. of the 19th International Working Conference on Requirements Engineering: Foundation for Software Quality*, REFSQ'05, June 2005. 12

[91] N. R. Mead and T. Stehney. Security Quality Eequirements Engineering (SQUARE) Methodology. *ACM SIGSOFT Software Engineering Notes*, pages 1–7, May 2005. xiii, 18

[92] T. Miehling, P. Vondracek, M. Huber, H. Chodura, and G. Bauersachs. HIS Flashloader Specification Version 1.1, 2006. 131, 133, 136

[93] H. Mouratidis, P. Giorgini, G. Manson, and I. Philp. A Natural Extension of Tropos Methodology for Modelling Security. In *Proc. of the Agent Oriented Methodologies Workshop*, OOPSLA'02, 2002. 9, 12, 41, 107

[94] N. Navet. Automotive Communication Systems: From Dependability to Security. In *1st Seminar on Vehicular Communications and Applications*, VCA'11, May 2011. 143

[95] A. Nhlabatsi, B. Nuseibeh, and Y. Yu. Security Requirements Engineering for Evolving Software Systems: A Survey. *Journal of Secure Software Engineering*, pages 54–73, 2009. 18

[96] D. K. Nilsson and U. E. Larson. Secure Firmware Updates Over the Air in Intelligent Vehicles. In *IEEE International Conference on Communications Workshops*, ICC Workshops'08, pages 380–384, May 2008. 135

[97] D. K. Nilsson, T. Roosta, U. Lindqvist, and A. Valdes. Key Management and Secure Software Updates in Wireless Process Control Environments. In *Proc. of the 1st ACM conference on Wireless network security*, WiSec'08, pages 100–108, 2008. 135

[98] D. K. Nilsson, L. Sun, and T. Nakajima. A Framework for Self-Verification of Firmware Updates Over the Air in Vehicle ECUs. In *IEEE Global Communication Conference*, GLOBECOM Workshops'08, pages 1–5, December 2008. 135

[99] NIST-SP-800:30. Risk Management Guide for Information Technology Systems. Available at `http://csrc.nist.gov/publications/nistpubs/800-30-rev1/sp800_30_r1.pdf`, September 2012. 33

[100] T. Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Embedded Technology. Elsevier Science, 2005. 2, 194

[101] OASIS. XACML: eXtensible Access Control Markup Language TC v2.0. Available at `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cos01-en.pdf`, September 2012. 142

[102] I. Ober and I. Dragomir. OMEGA2: A New Version of the Profile and the Tools. In *14th IEEE International Conference on Engineering of Complex Computer Systems*, UML&AADL'09, pages 373–378, June 2009. 46

[103] Objectiver. A KAOS Tutorial. Available at `http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf`, 2007. xi, 11

[104] M. J. O'Connor and A. K. Das. SQWRL: A Query Language for OWL. In *Proc. of the 5th International Workshop on OWL: Experiences and Directions*, OWLED'09, October 2009. 63

[105] Ministry of Defense. DoD Architecture Framework Volume II: Product Descriptions. Available at `http://dodcio.defense.gov/Portals/0/Documents/DODAF/DoDAF_Volume_II.pdf`, April 2007. 25

[106] OMG. Object Management Group. `http://www.omg.org`. 4, 196

[107] OMG. SysML - The Systems Modeling Language Specification, (ptc/02-06-12), OMG final adopted specification. Available at `http://www.omgsysml.org/`, 2012. xi, 4, 23, 39, 42, 43, 44, 45, 46, 48, 49, 50, 51, 196

[108] OWASP. Open Web Application Security Project. `https://www.owasp.org`. 33

[109] P. Papadimitratos, V. Gligor, and J-P. Hubaux. Securing Vehicular Communications - Assumptions, Requirements, and Principles. In *4th Workshop on Embedded Security in Cars*, ESCAR'06, November 2006. 148

[110] Papyrus. Papyrus for SysML. Available at `http://www.papyrusuml.org/`. 46

[111] F. Silva Parreiras and S. Staab. Using Ontologies with UML Class-based Modeling: The TwoUse Approach. *Data & Knowledge Engineering*, pages 1194–1207, 2010. 56

[112] F. Silva Parreiras, S. Staab, and A. Winter. On Marrying Ontological and Metamodeling Technical Spaces. In *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE'07, September 2007. 56

[113] M. Pavlidis and S. Islam. SecTro: A CASE Tool for Modelling Security in Requirements Engineering using Secure Tropos. In *23rd International Conference on Advanced Information Systems Engineering*, CAiSE'11, pages 89–96, June 2011. xi, 13

[114] G. Pedroza, D. Knorreck, and L. Apvrille. AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties. In *11th IEEE Conference on Distributed Systems and New Technologies*, NOTERE'11, may 2011. 46

[115] CESSA Project. CESSA - Compositional Evolution of Secure Services using Aspects. `http://cessa.gforge.inria.fr/`. 4, 195

[116] CVIS Project. CVIS - Cooperative Vehicle Infrastructure Systems. `http://www.cvisproject.org/`. 148

[117] EVITA Project. E-safety Vehicle InTrusion protected Applications. `http://www.evita-project.org`. 4, 100, 196

[118] GST Project. Global Systems for Telematics. `http://www.gst-forum.org/`. 135

[119] OVESEE Project. Open Vehicular Secure Platform. `https://www.oversee-project.com/`. 148, 149

[120] SeVeCOM Project. Secure Vehicle Communication. `http://www.sevecom.org/`. 135

[121] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Available at `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`, January 2008. 63

[122] R. Oberhauser R. Schmidt, C. Bartsch. Ontology-based Representation of Compliance Requirements for Service Processes. In *Workshop on Semantic Business Process and Product Lifecycle Management*, SBPM'07, June 2007. 20

[123] M. Raya, D. Jungels, P. Papadimitratos, I. Aad, and J-P. Hubaux. Certificate Revocation in Vehicular Networks. Technical Report LCAREPORT-2006-006, Laboratory for Computer Communications and Applications (LCA), EPFL, 2006. 148

[124] M. Raya, P. Papadimitratos, and Jean-Pierre Hubaux. Securing Vehicular Communications. *IEEE Wireless Communications Magazine*, Vol 13:8–15, 2006. 148

[125] BMW Group Research and Technology. EMVY: The Embedded Vehicular IT Security Construction Kit. Basic Concept, June 2009. 96

[126] D. Richards, A. Stuart, and M. Hause. Testing Solutions through UML/SysML. In *19th Annual INCOSE International Symposium*, INCOSE'09, 2009. 46

[127] G-C. Roman. A Taxonomy of Current Issues in Requirements Engineering. *IEEE Computer Society - Computer*, pages 14–23, April 1985. 20

[128] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar. Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study. In *Proc. of the 19th USENIX Security Symposium*, USENIX Security'10, August 2010. 137

[129] A. Ruddle, D.Ward, B. Weyl, M. Sabir Idrees, Y.Roudier, M.Friedewald, T.Leimbach, A.Fuchs, S.Gürgens, O.Henniger, R. Rieke, M.Ritscher, H.Broberg, L. Apvrille, R. Pacalet, and G.Pedroza. Security Requirements for Automotive On-Board Networks based on Dark-side Scenarios. Technical Report 2.3, EVITA Project, 2010. 1, 54, 87, 89, 100, 109, 131, 167, 193

[130] M. Saeki and H. Kaiya. Security Requirements Elicitation Using Method Weaving and Common Criteria. In *ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'08 Workshops, pages 185–196, 2008. 25

[131] W. Schindler. Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators. Available at `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/ais20e_pdf.pdf?__blob=publicationFile`, 2007. 127

[132] A. Schmidt, F. Waas, M. Kersten, Michael J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of 28th International Conference on Very Large Data Bases*, VLDB'02, pages 974–985, August 2002. 146

[133] B. Schneier. Attack Trees: Modeling Security Threats. Available at `http://www.schneier.com/paper-attacktrees-ddj-ft.html`, 1999. 53, 54, 55

[134] H. Schweppe, Y. Roudier, B. Weyl, L. Apvrille, and D. Scheuermann. Car2X communication : Securing the Last Meter - A Cost-Effective Approach for Ensuring Trust in Car2X Applications using In-Vehicle Symmetric Cryptography. In *4th IEEE International Symposium on Wireless Vehicular Communications*, WIVEC'11, September 2011. 124

[135] secunet. Towards a Secure Automotive Platform (White Paper). Available at `http://www.secunet.com/fileadmin/user_upload/Download/Printmaterial/englisch/sn_Whitepaper_Secure_Automotive_Platform_E.pdf`, 2009. 136

[136] H. Seudie, E. Akcabelen, I. Ipli, H. Schweppe, Y. Roudier, and M. Sabir Idrees. Security Architecture Implementation. Technical Report D4.3, EVITA Project, 2011. 109

[137] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. Google Android: A State-of-the-Art Review of Security Mechanisms. Technical Report CoRR abs/0912.5101, Department of Information Systems Engineering, Ben-Gurion University, Israel, 2009. 67

[138] J. Sherwood, A. Clark, and D. Lynas. *Enterprise Security Architecture: A Business-Driven Approach.* CRC Press, 2005. 1, 193

[139] H. Shrobe. Computational Vulnerability Analysis for Information Survivability. *18th National Conference on Artificial Intelligence*, pages 919–926, 2002. 26

[140] K. Siegemund, E. J Thomas, Y. Zhao, J. Pan, and U. Assmann. Towards Ontology-driven Requirements Engineering. *7th International Workshop on Semantic Web Enabled Software Engineering*, October 2011. 20

[141] G. Sindre and A. L. Opdahl. Eliciting Security Requirements by Misuse Cases. In *Proc. of the 37th International Conference on Technology of Object-Oriented Languages and Systems*, TOOLS-Pacific'00, pages 120–131, 2000. xi, 9, 11, 16, 17, 26, 41, 90, 107

[142] A. SOUAG, I. Comyn-Wattiau, and C. Salinesi. Ontologies for Security Requirements: A Literature Survey and Classification. In *2nd International Workshop on Information Systems Security Engineering*, WISSE'12, pages 1–8, June 2012. 21

[143] J. Stefan and M. Schumacher. Collaborative Attack Modeling. In *Proc. of the ACM Symposium on Applied Computing*, SAC'02, pages 253–259, March 2002. 54, 87

[144] Sparx Systems. Enterprise Architect for SysML. Available at `http://www.sparxsystems.com/`. 46

[145] A. Thorn, T. Christen, B. Gruber, R. Portman, and L. Ruf. What is a Security Architecture? Available at `http://www.isss.ch/fileadmin/publ/agsa/Security_Architecture.pdf`. 1, 193

[146] B. Tsoumas and D. Gritzalis. Towards an Ontology-based Security Management. In *Proc. of the 20th International Conference on Advanced Information Networking and Applications*, AINA'06, pages 985–992, 2006. 21

[147] Stanford University. Protégé Ontology Editor and Knowledge-base Framework. Available at `http://protege.stanford.edu/`. 63

[148] P. Henricus Antonius van der Putten. *Specification of Reactive Hardware/software Systems: The Method Software/Hardware Engineering (SHE).* Ph.D. Thesis, Eindhoven University of Technology, Department of Electrical Engineering, 1997. 79

[149] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using Description Logic to Maintain Consistency between UML Models. In *6th International Conference on The Unified Modeling Language, Modeling Languages and Applications*, UML'03, pages 326–340, 2003. 56

[150] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proc. of the Fifth IEEE International Symposium on Requirements Engineering*, RE'01, pages 249–262, 2001. 24

[151] A. van Lamsweerde. Engineering Requirements for System Reliability and Security. In *Software System Reliability and Security*, NATO Security through Science Series - Information and Communicarion Security, pages 196–238, 2007. 3, 9, 11, 16, 24, 25, 28, 41, 55, 56, 101, 195

[152] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley Publishing, March 2009. 2, 194

[153] A. van Lamsweerde and E. Letier. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In *Radical Innovations of Software and System Engineering*, pages 4–8, 2003. 9, 10, 107

[154] D. Veillard. Libxml - The XML C Parser and toolkit of Gnome. Available at `http://www.xmlsoft.org`, 2009. 147

[155] C. Veres, J. Sampson, S.J. Bleistein, K. Cox, and J. Verner. Using Semantic Technologies to Enhance a Requirements Engineering Approach for Alignment of IT with Business Strategy. In *International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS'09, pages 469–474, March 2009. 21

[156] G. Vigna, S. Eckmann, and R. Kemmerer. Attack Languages. In *Proc. of the IEEE Information Survivability Workshop*, ISW'00, pages 163–166, October 2000. 54

[157] A. Vorobiev and N. Bekmamedova. An Ontology-Driven Approach Applied to Information Security. *Journal Of Research And Practice In Information Technology*, pages 61–76, 2010. 122

[158] D. A. Wagner, M. B. Bennett, R. Karban, N. Rouquette, S. Jenkins, and M. Ingham. An ontology for State Analysis: Formalizing the mapping to SysML. In *IEEE Aerospace Conference*, pages 1–16, March 2012. 56

[159] WAND, Inc. WAND Automotive Taxonomy. Available at `http://www.wandinc.com/`. 29

[160] IEEE WAVE. Wireless Access in Vehicular Environments, IEEE standard 1609.2. 135

[161] B. Weyl, M. Wolf, F. Zweers, T. Gendrullis, M. Sabir Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. E. Khayari, O. Henniger, D. Scheuermann, A. Fuchsa, L. Apvrille, G. Pedroza, H. Seudie, J. Shokrollahi, and A. Keil. Secure On-board Architecture Specification. Technical Report D3.2, EVITA Project, 2010. xii, 3, 124, 125, 194

[162] J. Whittle, D. Wijesekera, and M. Hartong. Executable misuse cases for modeling security concerns. In *Proc. of the 30th international conference on Software engineering*, ICSE'08, pages 121–130, 2008. 16

[163] M. Wolf, A. Weimerskirch, C. Paar, and M. Bluetooth. Security in Automotive Bus Systems. In *2nd Workshop on Embedded Security in Cars*, ESCAR'04, 2004. 2, 137, 194

[164] M. Wolf, A. Weimerskirch, and T. J. Wollinger. State of the Art: Embedding Security in Vehicles. *Journal on Embedded Systems, Special Issue: Embedded Systems for Intelligent Vehicles*, 2007. 2, 3, 194

[165] Y. Wu, Q. Zhang, Z. Yu, and J. Li. A Hybrid Parallel Processing for XML Parsing and Schema Validation. In *Proc. of Balisage: The Markup Conference*, August 2008. 146

[166] Y. Ying-ying, L. Zong-yong, and W. Zhi-xue. Domain Knowledge Consistency Checking for Ontology-Based Requirement Engineering. In *International Conference on Computer Science and Software Engineering*, volume 2, pages 302–305, December 2008. 20

[167] X. Zhu. Inconsistency Measurement of Software Requirements Specifications: An Ontology-Based Approach. In *Proc. of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS'05, pages 402–410, 2005. 20

[168] S. Zrelli, A. Miyaji, Y. Shinoda, and T. Ernst. Security and Access Control for Vehicular Communications. In *Proc.of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication*, WIMOB'08, pages 561–566, October 2008. 148

Appendix A

Security Properties

We follow a revised definition of security properties specified in the EVITA project [129], that is relevant for heterogeneous networks and embedded systems. The informal explanations below reflect how these concepts are generally understood.

A.1 Data origin authenticity

A data origin authenticity property applies to a quantum of information and a claimed author. The property is satisfied when the quantum of information truly originates from the author. The property can be made more specific by providing an observation of the quantum of information (defined, e.g., by a time and a location in the system). The author can also be constrained by adding a time and/or a place of creation of the quantum of information. Note that in most security oriented frameworks, data origin authenticity implies integrity.

A.2 Integrity

An integrity property applies to a quantum of information between two observations (defined, e.g., by a time and a location in the system). The property is satisfied when the quantum of information has not been modified between the two observations. It guarantees for instance that the content of a storage facility has not been modified between two given read operations, or that a message sent on a communication channel has not been altered during its journey.

A.3 Authorization

A controlled access property or requirement applies to a set of actions and/or information and a set of authorized entities. The property is guaranteed if the specified entities are the only entities that can perform the actions or access the information. The property can be further detailed with time constraints on the period of authorization. Controlled access is needed to ensure that stakeholders only have access to information and functions that they are authorized to access as appropriate to their expected activities.

A.4 Freshness

A freshness property applies to a quantum of information, a receiving entity, and a given time. The property is satisfied if the quantum of information received by the entity at the given time is not a copy of the same information received by the same or another entity in the past. Ensuring freshness can be used to prevent replay attacks.

A.5 Non-Repudiation

A non-repudiation property or requirement applies to an action and an entity performing the action. The non-repudiation of the action is guaranteed if it is impossible for the entity that performed the action to claim that it did not perform it. This property can be further detailed with a set of entities for which the action needs to be undeniable, with a time limit, etc. There may be specific legal requirements for non-repudiation. However, non-repudiation may also be introduced for convenience, for example, as an aid in providing evidence or proving liability.

A.6 Privacy

A privacy property or requirement applies to an entity and a set of information. qPrivacy is guaranteed if the relation between the entity and the set of information is confidential. This relation can however significantly vary. For instance, one generally distinguishes different types of privacy, typically anonymity, unlinkability, and pseudonymity.

A.6.1 Anonymity

This is the property that the relation between an entity and its identity is strictly confidential. Privacy property must be made consistent with potentially conflicting requirements for identification, auditing, non-repudiation and jurisdictional access, which may require users to be identified and information about their interactions to be stored.

A.6.2 Unlinkability

The unlinkability of two or more Item of Interest (abbreviated IOIs, e.g., subject, messages, actions, ... sent, received, or performed by the principal) from an attacker′s perspective means that within the system (comprising these and possibly other items), the attacker cannot sufficiently distinguish whether these IOIs are related or not.

A.6.3 Pseudonymity

Pseudonymity refers to the capability of recognizing the same subject without being able to relate him to his identity [1].

A.7 Confidentiality

A confidentiality property applies to a quantum of information and a set of authorized entities. The property is satisfied when the authorized entities are the only ones that can know the quantum of information. Privacy relies on confidentiality and can be considered as a special case of confidentiality.

A.8 Availability

An availability property applies to a service or a physical device providing a service. The property is satisfied when some service is operational. Denial of service attacks aim at compromising the availability of their target. The property can be further detailed with the specification of a period during which the availability is required and of a set of client entities requesting the availability.

[1] A pseudonym is an identifier, that is, a name or another bit string, generated in a fully independent manner from the subject and related attributes values, do not contain side information on the subject they are attached to.

Appendix B

Risk Model

B.1 Introduction

The provision of security measures, like any other feature of a product or service, is inevitably accompanied by development and implementation costs. Consequently, protection against every conceivable security threat would be too costly for the development of complex systems, so resources need to be targeted on the most significant threats. The countermeasures that are selected for inclusion in the design must therefore be based on an objective assessment of potential threats and their anticipated implications. Thus, in order to identify the most important security requirements that are needed to secure the system it is necessary to assess the level of risk that may be posed by potential attacks. This provides a convenient basis for systematically identifying and prioritizing threats that need to be mitigated as follows:

- Where a number of possible attack objectives may achieve the attack goal, the attack objective with the highest perceived risk level is the priority for countermeasures to reduce the risk level for the attack objective.
- Where a number of possible attack methods may lead to the same attack objective, the attack method with the highest perceived attack probability (i.e. lowest attack potential) is the priority for countermeasures to reduce the risk level for the attack objective.
- Were a number of asset attacks may lead to the same attack objective, the asset attack with the highest perceived attack probability (i.e. lowest attack potential) is the priority for countermeasures to reduce the risk level for the attack objective.

The results of the risk analysis is collated over all of the attack scenarios that are considered and summarized in terms of the number of instances of particular risk levels found for each of the attacks that were envisaged against the various system assets (see Table B.1). This therefore gives an indication of the relative importance of protecting the system by providing countermeasures for specific assets (i.e. what to protect) against particular types of attacks. Since all the investigated functions assume a common basic architecture, it is likely that common patterns will arise in the attack trees. Consequently, the repeated occurrence of particular attack patterns in attack trees is a further indicator for prioritizing countermeasures that are likely to provide favorable cost-benefit properties. Furthermore, the security requirements are mapped onto the functions and assets, providing a way to evaluate the system performance (latency, throughput, and resource utilization) and allow the designer to further prioritize the security requirements according to system needs. However, the expected cost of the proposed countermeasures also needs to be taken into account in selecting specific security requirements.

Identified Threats		**Risk Analysis results**		**Security**
Assets	*Attack*	*Risk Level*	*Instances*	**Requirements**
software/ hardware	$AT_{(1-n)}$	$R_{(0-6)}$	*1 - n*	$SR_{(1-n)}$

Table B.1: Security Requirements Prioritizstion

In order to identify the most important security requirements that are needed to prevent (or at least detect and contain) key threats, it is necessary to assess the level of risk that may be posed by potential attacks. The risk associated with an attack is a function of the possible *severity* of the attack for the stakeholders and the estimated *probability* of occurrence of a successful attack of this nature. The severity of an attack is assessed using the attack tree, by considering the potential implications of the attack objectives for the stakeholders. The probability of a successful attack is also derived from the attack tree, by identifying combinations of possible attacks on the system assets that could contribute to an attack method.

Attack Potential		**Attack Probability**	
Rating	*Description*	*Likelihood*	*Ranking*
0 - 9	Basic	Highly Likely	5
10 - 13	Enhanced-Basic	Likely	4
14 - 19	Moderate	Possible	3
20 - 24	High	Unlikely	2
≥ 25	Beyond-High	Remote	1

Table B.2: Relating attack potential to attack probability

To determine for each path in an attack tree the attack potentials of the contributing asset attacks are defined and classified, as shown in Table B.2. Note that once an attack scenario has been identified and been exploited, it may be exploited repeatedly with less effort than for the first time. Both phases, identification and exploitation, are considered in conjunction. In this context the term attack potential is really describing the difficulty of mounting a successful attack, while for risk analysis purposes a probability measure is required. A high probability of successful attack is assumed to correspond to the basic attack potential, since many possible attackers will have the necessary attack potential. Conversely, a high attack potential suggests a lower probability of successful attacks, since the number of attackers with the necessary attack potential is expected to be comparatively small. Consequently, Table B.2 also proposes an associated numerical scale that reflects the relative probability of success associated with the attack potential in a more intuitive manner. The attack probability measure (P) is higher for easier attacks that are associated with lower attack potentials, and lower for more difficult attacks associated with the higher attack potentials. Where the severity vector includes a non-zero safety component, the risk

Severity (Si)	**Combined attack probability** (A)				
	A=1	A=2	A=3	A=4	A=5
S_i	$R_{(0\text{-}6)}$	$R_{(0\text{-}6)}$	$R_{(0\text{-}6)}$	$R_{(0\text{-}6)}$	$R_{(0\text{-}6)}$

Table B.3: Proposed security risk levels mapped to severity and probability

assessment may include an additional probability parameter that represents the potential to influence the severity of the outcome. The probability and severity combinations are mapped to a series of risk levels ranging from **0** (lowest) to **6** (highest) in order to rank relative risks (see Table B.3). The risk level *(**R**, a vector)* is determined from the severity (**S**) associated with the attack objective and the combined attack probability (A) associated with a particular attack method. This is achieved by mapping the severity and attack probability to the risk using a risk graph approach.

In this scheme, attacks that are judged to be of high probability and high severity are considered to represent high risks, and therefore definitely require countermeasures to mitigate their potential impact, while other attacks that are judged to be of low probability and low severity are considered to be low risk, and may not require specific countermeasures. However, more careful consideration may be required for those threats (likely to be in the majority) that are less readily ranked in terms of relative risk, such as those judged to be either fairly likely but not particularly severe, or relatively unlikely but fairly severe. Risk management is an iterative process that can be performed during each major phase of the SDLC. However, the risk management methodology is the same regardless of the SDLC phase for which the assessment is being conducted. We may use several methodologies and standards (i.e., ISO/IEC 13335, NIST SP-800:30, ISO/IEC ISO 31000:2009, ISO/IEC 73:2002, or ISO 14971:2000 etc.) to carry out the risk assessment. These standards provide guidance and describe the characteristics of each SDLC phase and indicate how risk management can be performed in support of each phase.

B.1.1 Risk Analysis

Once security needs of firmware update application are defined for each security criterion, we analyzed the risks. We first started by analyzing the attacker capabilities relevant to each system asset (i.e., functions, processes, middleware, or hardware components, etc.) involved in the firmware update process. Table B.4 summarizes estimates for the "attack potential[1]", together with the underlying estimates for the influencing factors, for various attacks identified for firmware update scenario. Consequently, it also indicates an associated numerical scale that reflects the relative value and ranking/rating of attack potential in a more intuitive manner. The estimates are based on as-is automotive on-board networks, prior to the introduction of security measures. The results of the risk analysis are summarized in terms of the frequency of the risk levels found for each threat. This gives an indication of the relative importance of protecting against specific attacks: While a low maximum risk suggests a low priority, a high maximum risk suggests a higher priority for protection. A lower risk that appears in many attack trees, however, might be as important to tackle than a higher risk tears only once.

[1]For estimating risk in a non-ambiguous manner, an expert adversary is chosen

Asset (attack)	Elaspsed Time	Expertise	Knowledge of System	Window of Opportunity	Equipment	Attack Potential	
						Value	Ranking
Communication Unit (exploit vulnerability or implementation error)	10	6	3	4	7	33	Beyond High
Keys (illegal acquisition, modification, etc.)	17	6	7	10	7	35	Beyond High
In-Car Communication (corrupt or fake messages)	10	6	3	0	4	23	High
CU (denial of service)	4	6	3	1	4	15	Moderate

Table B.4: Evaluation of required attack potential for asset attacks identified from attack trees

Appendix C

SysMLsec-to-Ontology Translation Engine

C.1 SysMLsec Knowledge Extraction

```
1  package sysmlsec.ontology;
2
3  import javax.xml.parsers.DocumentBuilderFactory;
4  import javax.xml.parsers.DocumentBuilder;
5  import org.w3c.dom.Document;
6  import org.w3c.dom.NodeList;
7  import org.w3c.dom.Node;
8  import org.w3c.dom.Element;
9
10 import com.hp.hpl.jena.vocabulary.RDF.Nodes;
11
12 import antlr.debug.NewLineEvent;
13
14 import java.io.File;
15 import java.util.ArrayList;
16 import java.util.Arrays;
17 import java.util.HashMap;
18 import java.util.Iterator;
19 import java.util.Set;
20
21 public class TtoOntology {
22
23   /**
24    * for one subTab, Store all of the instances and its parameters, the
         format is
25    * TabName:
26    *     classA : Id : 0 kind: functional ....
27    */
28   private HashMap<String, HashMap<String, HashMap<String, String>>>
       allTabsInstances = new HashMap<String, HashMap<String,HashMap<String
       ,String>>>();
29
30   /**
31    * Store all of the relationships between the classes. the format is :
32    * {tabname:[classA, [classB, relationship]]}
33    */
34   private HashMap<String, HashMap<String, HashMap<String, ArrayList<String
       >>>> allTabsProperty = new HashMap<String, HashMap<String,HashMap<
       String,ArrayList<String>>>>();
35   private String filename;
36   private String labelName;
37
38   public TtoOntology(String filename,String labelString) {
39     this.filename = filename;
```

```
40      this.labelName = labelString;
41    }
42
43    public static void main(String[] args) {
44      TtoOntology tt = new TtoOntology("thesis-ontology.xml","SysMLsec
            Requriement");
45      HashMap<String, HashMap<String, HashMap<String, String>>>
            allTabsInstances = tt.getInstanceHashmap();
46      HashMap<String, HashMap<String, HashMap<String, ArrayList<String>>>>
            allTabsProperty = tt.getRelationshipHashMap();
47      Iterator<String> propertyIterator = allTabsProperty.keySet().iterator
            ();
48      while (propertyIterator.hasNext()) {
49        String tabName = propertyIterator.next();
50        System.out.println("Tab Name is " + tabName+"--------");
51        HashMap<String, HashMap<String, ArrayList<String>>>
              ontologyPropertyHashMap = allTabsProperty.get(tabName);
52        Iterator<String> ontologyproIterator = ontologyPropertyHashMap.
              keySet().iterator();
53        while (ontologyproIterator.hasNext()) {
54          String className = ontologyproIterator.next();
55          System.out.println("class "+ className + " has properties:");
56          HashMap<String, ArrayList<String>> proObj =
                ontologyPropertyHashMap.get(className);
57          Iterator<String> proObjiIterator = proObj.keySet().iterator();
58          while(proObjiIterator.hasNext()){
59            String propertyString = proObjiIterator.next();
60            System.out.println("----"+propertyString);
61            ArrayList<String> classesArrayList = proObj.get(propertyString);
62            for (int i = 0; i < classesArrayList.size(); i++) {
63              System.out.println("--------"+classesArrayList.get(i));
64            }
65          }
66        }
67      }
68    }
69
70    /**
71     * get the hashmap who contains all of the instances and their
            properties.
72     *
73     * @return the hashmap
74     */
75    public HashMap<String, HashMap<String, HashMap<String, String>>>
          getInstanceHashmap() {
76      Element modeling = readFile();
77      NodeList subTabs = modeling.getElementsByTagName(this.labelName);
78      for (int i = 0; i < subTabs.getLength(); i++) {
79        Element subtab= (Element)subTabs.item(i);
80        initialAllInstance(subtab);
81      }
82      return this.allTabsInstances;
83    }
84
85    /**
86     * get the hashmap who contains all of the instances and their
            relationships
87     *
88     * @return the relationship hashmap
89     */
90    public HashMap<String, HashMap<String, HashMap<String, ArrayList<String
          >>>> getRelationshipHashMap() {
```

```
91      Element document = readFile();
92      NodeList subTabs = document.getElementsByTagName(this.labelName);
93
94      for (int i = 0; i < subTabs.getLength(); i++) {
95        Element subtab= (Element)subTabs.item(i);
96        parseRelationShip(subtab);
97      }
98      return this.allTabsProperty;
99    }
100
101   /**
102    * read the given file and return the first Modeling element
103    *
104    * @return the first Modeling Element
105    */
106   private Element readFile() {
107     try {
108       File fXmlFile = new File(filename);
109       DocumentBuilderFactory dbFactory = DocumentBuilderFactory.
              newInstance();
110       DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
111       Document doc = dBuilder.parse(fXmlFile);
112       doc.getDocumentElement().normalize();
113       Element firstPart = (Element) doc.getElementsByTagName("Modeling").
              item(0);
114       System.out.println("first part property is "+firstPart.getAttribute(
              "nameTab"));
115       return firstPart;
116     } catch (Exception e) {
117       e.printStackTrace();
118       return null;
119     }
120   }
121    // public
122
123   /**
124    * this function read all of the relationships between the classes, and
125    * store them in the hashmap.
126    *
127    * @param doc the "Modeling Element" want to be parsed.
128    *
129    */
130   private void parseRelationShip(Element doc) {
131     HashMap<String, HashMap<String, ArrayList<String>>> allRelationShipMap
            = new HashMap<String, HashMap<String, ArrayList<String>>>();
132     try {
133       // all relationships are represented by the tag "connector"
134       NodeList nList = doc.getElementsByTagName("CONNECTOR");
135       for (int temp = 0; temp < nList.getLength(); temp++) {
136
137         Node nNode = nList.item(temp);
138         if (nNode.getNodeType() == Node.ELEMENT_NODE) {
139
140           Element eElement = (Element) nNode;
141           /*
142            * for one relationship, get the related two component name
143            */
144           String p1idString = getTagValue("id", "P1", eElement);
145           String p2idString = getTagValue("id", "P2", eElement);
146           String className1 = getTheNameById(doc, p1idString);
147           String className2 = getTheNameById(doc, p2idString);
148           className1.replaceAll(" ", "_");
```

```
149           className2.replaceAll(" ", "_");
150           /**
151            *  get the relation type, like "subclass of", "equivalent" or "
                   containment"
152            */
153           Element infoparamElement = (Element) eElement.
                  getElementsByTagName("infoparam").item(0);
154           String type = infoparamElement.getAttribute("value");
155           if (type.contains("deriveReqt")) {
156             this.addProperty(allRelationShipMap,className1, className2, "
                    superClassOf");
157           } else if (type.contains("composition")) {
158             this.addProperty(allRelationShipMap,className1, className2, "
                    contains");
159           } else if (type.contains("copy")) {
160             this.addProperty(allRelationShipMap,className1, className2, "
                    equivalent");
161           }
162         }
163       }
164     } catch (Exception e) {
165       e.printStackTrace();
166     }
167     String name = doc.getAttribute("name");
168     name = name.replaceAll(" ", "_");
169     this.allTabsProperty.put(name, allRelationShipMap);
170   }
171   private void addProperty(HashMap<String, HashMap<String, ArrayList<
        String>>> allRelationShipMap,String className1, String className2,
        String objpro) {
172     HashMap<String, ArrayList<String>> hash1 ;
173     ArrayList<String> nameList;
174     if(allRelationShipMap.containsKey(className2)) {
175       hash1 = allRelationShipMap.get(className2);
176       nameList = hash1.get(objpro);
177       if(nameList == null) {
178         nameList = new ArrayList<String>();
179         nameList.add(className1);
180         hash1.put(objpro, nameList);
181       } else {
182         nameList.add(className1);
183       }
184     }
185     else {
186       hash1 = new HashMap<String, ArrayList<String>>();
187       nameList = new ArrayList<String>();
188       nameList.add(className1);
189       hash1.put(objpro, nameList);
190       allRelationShipMap.put(className2, hash1);
191     }
192   }
193   /**
194    * read all of the instances and their properties in the file, and store
           them in the hashmap.
195    * @param doc
196    */
197   private void initialAllInstance(Element doc) {
198       HashMap<String, HashMap<String, String>> allInstancesHashMap = new
            HashMap<String, HashMap<String, String>>();
199     NodeList components = doc.getElementsByTagName("COMPONENT");
200     for (int i = 0; i < components.getLength(); i++) {
201       Element component = (Element) components.item(i);
```

```
202       Element infoparamElement = (Element) component.getElementsByTagName(
              "infoparam").item(0);
203       String name = infoparamElement.getAttribute("value");
204       name = name.replaceAll(" ", "_");
205       HashMap<String, String> parameters = new HashMap<String, String>();
206       Element extrparam = (Element) component.getElementsByTagName("
              extraparam").item(0);
207       NodeList paralist = extrparam.getChildNodes();
208       for (int j = 0; j < paralist.getLength(); j++) {
209         Node tmp = paralist.item(j);
210         if (tmp.getNodeType() == Element.ELEMENT_NODE) {
211           Element para = (Element) tmp;
212           parameters.put(para.getNodeName(), para.getAttribute("data"));
213         }
214       }
215       allInstancesHashMap.put(name, parameters);
216     }
217     String name = doc.getAttribute("name");
218     name = name.replaceAll(" ", "_");
219     this.allTabsInstances.put(name, allInstancesHashMap);
220
221   }
222
223   private String getTheNameById(Element doc, String id) {
224     NodeList components = doc.getElementsByTagName("COMPONENT");
225     for (int i = 0; i < components.getLength(); i++) {
226       Element component = (Element) components.item(i);
227       NodeList points = component.getElementsByTagName("TGConnectingPoint"
              );
228       for (int j = 0; j < points.getLength(); j++) {
229         Element pointElement = (Element) points.item(j);
230         String pointID = pointElement.getAttribute("id");
231         if (pointID.equals(id)) {
232           Element infoparamElement = (Element) component.
                  getElementsByTagName("infoparam").item(0);
233           String name = infoparamElement.getAttribute("value");
234           return name;
235         }
236
237       }
238     }
239     return null;
240   }
241
242   private String getTagValue(String attrName, String sTag, Element
          eElement) {
243     Element subNode = (Element) (eElement.getElementsByTagName(sTag).item
            (0));
244
245     return subNode.getAttribute(attrName);
246   }
247 }
```

Listing C.1: SysMLsec Knowledge Extraction

C.2 Building OWL Ontological Instance

```
1 package sysmlsec.ontology;
2
3 import java.io.File;
```

```
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.HashMap;
7  import java.util.Iterator;
8
9  import org.apache.log4j.lf5.PassingLogRecordFilter;
10 import org.openjena.atlas.iterator.Iter;
11
12 import edu.stanford.smi.protege.exception.OntologyLoadException;
13 import edu.stanford.smi.protegex.owl.ProtegeOWL;
14 import edu.stanford.smi.protegex.owl.jena.JenaOWLModel;
15 import edu.stanford.smi.protegex.owl.model.NamespaceManager;
16 import edu.stanford.smi.protegex.owl.model.OWLDatatypeProperty;
17 import edu.stanford.smi.protegex.owl.model.OWLIndividual;
18 import edu.stanford.smi.protegex.owl.model.OWLMaxCardinality;
19 import edu.stanford.smi.protegex.owl.model.OWLModel;
20 import edu.stanford.smi.protegex.owl.model.OWLNamedClass;
21 import edu.stanford.smi.protegex.owl.model.OWLObjectProperty;
22 import edu.stanford.smi.protegex.owl.model.RDFIndividual;
23
24 public class Ontology {
25
26   /**
27    * store the instances and its dataproperties.
28    */
29   private HashMap<String, HashMap<String, HashMap<String, String>>>
         allInstancesHashMap;
30   /**
31    * store the object properties between the instances.
32    */
33   private HashMap<String, HashMap<String, HashMap<String, ArrayList<String
         >>>> allRelationShipMap;
34
35   public Ontology(HashMap<String, HashMap<String, HashMap<String, String
         >>> instances,
36       HashMap<String, HashMap<String, HashMap<String, ArrayList<String>>>>
              allRelationShipMap) {
37     this.allInstancesHashMap = instances;
38     this.allRelationShipMap = allRelationShipMap;
39   }
40
41   /**
42    * generate the classes and the data type corresponding to these classes
         .
43    *
44    * @param prefix
45    *             the prefix of the ontology
46    * @param ontologyName
47    *             ontology uri
48    * @param filename
49    *             the file to store
50    */
51   public void generateOntology(String prefix, String ontologyName, String
         filename) {
52     try {
53
54       // create a new ontology
55       JenaOWLModel owlModel = ProtegeOWL.createJenaOWLModel();
56       // set the namespace
57       NamespaceManager nsmanager = owlModel.getNamespaceManager();
58       nsmanager.setDefaultNamespace(ontologyName);
```

```
59        owlModel.getTripleStoreModel().getTopTripleStore().
              setOriginalXMLBase(ontologyName);
60        owlModel.getTripleStoreModel().getTopTripleStore().setPrefix(
              ontologyName, prefix);
61        // get a subtab element
62        Iterator<String> tabNameIterator = this.allInstancesHashMap.keySet()
              .iterator();
63        while (tabNameIterator.hasNext()) {
64          // get the name and create a superclass
65          String tabName = tabNameIterator.next();
66          OWLNamedClass tabClass = owlModel.createOWLNamedClass(tabName);
67          // get all of the subclasses of the subtab class
68          HashMap<String, HashMap<String, String>> allIntances = this.
                allInstancesHashMap.get(tabName);
69          Iterator<String> classnameIterator = allIntances.keySet().iterator
                ();
70          while (classnameIterator.hasNext()) {
71            String className = classnameIterator.next();
72            // create one class and add set its supperclass to the
73            // subtab
74            OWLNamedClass ontoClass = owlModel.createOWLNamedClass(className
                  );
75            ontoClass.removeSuperclass(owlModel.getOWLThingClass());
76            ontoClass.addSuperclass(tabClass);
77
78            // read all of the data properties it has.
79            Iterator<String> parasIterator = allIntances.get(className).
                  keySet().iterator();
80            while (parasIterator.hasNext()) {
81              String paraString = parasIterator.next();
82              try {
83                OWLDatatypeProperty property = owlModel.
                      createOWLDatatypeProperty(paraString);
84                property.setDomain(ontoClass);
85
86                property.setRange(owlModel.getXSDstring());
87              } catch (Exception e) {
88                // if we have already created the data property, we
89                // add union domain class.
90                OWLDatatypeProperty property = owlModel.
                      getOWLDatatypeProperty(paraString);
91                property.addUnionDomainClass(ontoClass);
92              }
93            }
94          }
95        }
96        // set the relationship between the classes.
97        Iterator<String> tabRelations = this.allRelationShipMap.keySet().
              iterator();
98        while (tabRelations.hasNext()) {
99          String tabName = tabRelations.next();
100         OWLNamedClass tabClass = owlModel.getOWLNamedClass(tabName);
101         HashMap<String, HashMap<String, ArrayList<String>>>
                relationshiphaHashMap = this.allRelationShipMap
                .get(tabName);
102         Iterator<String> relations = relationshiphaHashMap.keySet().
                iterator();
103         while (relations.hasNext()) {
104           String class1String = relations.next();
105           OWLNamedClass class1 = owlModel.getOWLNamedClass(class1String);
106           HashMap<String, ArrayList<String>> relationshipHashMap2 =
                  relationshiphaHashMap.get(class1String);
107
```

```
108           Iterator<String> relationIterator = relationshipHashMap2.keySet
                  ().iterator();
109           while (relationIterator.hasNext()) {
110             String relation = relationIterator.next();
111             ArrayList<String> classes = relationshipHashMap2.get(relation)
                    ;
112             for (int i = 0; i < classes.size(); i++) {
113               String class2String = classes.get(i);
114               OWLNamedClass class2 = owlModel.getOWLNamedClass(
                      class2String);
115               if (relation.equals("superClassOf")) {
116
117                 class2.removeSuperclass(tabClass);
118                 class2.addSuperclass(class1);
119               } else if (relation.equals("contains")) {
120
121                 OWLObjectProperty containsProperty;
122                 try {
123                   containsProperty = owlModel.createOWLObjectProperty("
                          contains");
124                   containsProperty.setDomain(class1);
125                   containsProperty.setRange(class2);
126                 } catch (Exception e) {
127                   // TODO: handle exception
128                   containsProperty = owlModel.getOWLObjectProperty("
                          contains");
129                   containsProperty.addUnionDomainClass(class1);
130
131                   containsProperty.addUnionRangeClass(class2);
132                 }
133               } else if (relation.equals("equivalent")) {
134                 class1.addEquivalentClass(class2);
135               }
136             }
137
138           }
139         }
140       }
141
142       owlModel.save(new File(filename).toURI());
143     } catch (Exception e) {
144       // TODO Auto-generated catch block
145       e.printStackTrace();
146     }
147   }
148
149   /**
150    * generate the instances
151    *
152    * @param ontologyname
153    * @param namespaceString
154    * @param filename
155    */
156   public void generateInstance(String ontologyname, String namespaceString
          , String filename) {
157     try {
158       JenaOWLModel owlModel = ProtegeOWL.createJenaOWLModelFromURI(
              ontologyname);
159       owlModel.getNamespaceManager().setDefaultNamespace(namespaceString);
160       Iterator<String> tabNameIterator = this.allInstancesHashMap.keySet()
              .iterator();
161       while (tabNameIterator.hasNext()) {
```

```
162         String tabName = tabNameIterator.next();
163         HashMap<String, HashMap<String, String>> instanceHashMap = this.
                allInstancesHashMap.get(tabName);
164         Iterator<String> classnameIterator = instanceHashMap.keySet().
                iterator();
165         while (classnameIterator.hasNext()) {
166           String className = classnameIterator.next();
167 
168           OWLNamedClass ontoClass = owlModel.getOWLNamedClass(className);
169           RDFIndividual instance = ontoClass.createRDFIndividual(className
                    + "Instance");
170           // instance.setPropertyValue(ageProperty, new Integer(0));
171           Iterator<String> parasIterator = instanceHashMap.get(className).
                  keySet().iterator();
172           while (parasIterator.hasNext()) {
173             String paraString = parasIterator.next();
174             String valueString = instanceHashMap.get(className).get(
                    paraString);
175             OWLDatatypeProperty property = owlModel.getOWLDatatypeProperty
                    (paraString);
176             instance.setPropertyValue(property, valueString);
177           }
178         }
179       }
180       owlModel.save(new File(filename).toURI());
181     } catch (Exception e) {
182       // TODO Auto-generated catch block
183       e.printStackTrace();
184     }
185   }
186 
187   /**
188    * generate the properties between the instances
189    * @param ontologyname
190    * @param namespaceString
191    * @param filename
192    */
193   public void generateObjectPropertyBetweenInstance(String ontologyname,
          String namespaceString, String filename) {
194     try {
195       JenaOWLModel owlModel = ProtegeOWL.createJenaOWLModelFromURI(
              ontologyname);
196       owlModel.getNamespaceManager().setDefaultNamespace(namespaceString);
197       Iterator<String> tabRelations = this.allRelationShipMap.keySet().
              iterator();
198       while (tabRelations.hasNext()) {
199         String tabName = tabRelations.next();
200         HashMap<String, HashMap<String, ArrayList<String>>>
                relationshiphaHashMap = this.allRelationShipMap
201                 .get(tabName);
202         Iterator<String> relations = relationshiphaHashMap.keySet().
                iterator();
203         while (relations.hasNext()) {
204           String class1String = relations.next();
205           RDFIndividual instanceOfClass1 = owlModel.getRDFIndividual(
                    class1String + "Instance");
206           // System.out.print(class1String+" contains ");
207           HashMap<String, ArrayList<String>> relationshipHashMap2 =
                    relationshiphaHashMap.get(class1String);
208           Iterator<String> relationIterator = relationshipHashMap2.keySet
                    ().iterator();
209           while (relationIterator.hasNext()) {
```

```
210             String relation = relationIterator.next();
211             ArrayList<String> classes = relationshipHashMap2.get(relation)
                    ;
212             for (int i = 0; i < classes.size(); i++) {
213               String class2String = classes.get(i);
214               RDFIndividual instanceOfClass2 = owlModel.getRDFIndividual(
                      class2String + "Instance");
215               // we just contains the "contains" property
216               if (relation.equals("contains")) {
217                 OWLObjectProperty containsProperty;
218                 containsProperty = owlModel.getOWLObjectProperty("contains
                        ");
219                 instanceOfClass1.addPropertyValue(containsProperty,
                        instanceOfClass2);
220               }
221             }
222
223           }
224         }
225       }
226       owlModel.save(new File(filename).toURI());
227     } catch (Exception e) {
228       // TODO: handle exception
229     }
230   }
231
232 }
```

Listing C.2: Building OWL Ontological Instance

Appendix D

XACML to ANS.1 Defintion

```
Access-control-xacml-2-0-policy-schema DEFINITIONS AUTOMATIC TAGS
::= BEGIN

VersionType ::= UTF8String

Description ::= UTF8String

XPathVersion ::= UTF8String

DefaultsType ::= CHOICE {
   xPathVersion [0] XPathVersion
}

PolicySetDefaults ::= DefaultsType

AttributeValueType ::= SEQUENCE {
   dataType [0] UTF8String,
   attr [1] SEQUENCE OF UTF8String
}

AttributeValue ::= AttributeValueType

SubjectAttributeDesignatorType ::= SEQUENCE {
   attributeId [0] UTF8String,
   dataType [1] UTF8String,
   issuer [2] UTF8String OPTIONAL,
   mustBePresent [3] BOOLEAN DEFAULT FALSE,
   subjectCategory [4] UTF8String
}

SubjectAttributeDesignator ::= SubjectAttributeDesignatorType

AttributeSelectorType ::= SEQUENCE {
   dataType [0] UTF8String,
   mustBePresent [1] BOOLEAN DEFAULT FALSE,
   requestContextPath [2] UTF8String
}

AttributeSelector ::= AttributeSelectorType

SubjectMatchType ::= SEQUENCE {
   matchId [0] UTF8String,
   attributeValue [1] AttributeValue,
   choice [2] CHOICE {
      subjectAttributeDesignator [0] SubjectAttributeDesignator,
      attributeSelector [1] AttributeSelector
   }
}

SubjectMatch ::= SubjectMatchType

```

```
52 SubjectType ::= SEQUENCE {
53    subjectMatch-list [0] SEQUENCE (SIZE (1..MAX)) OF subjectMatch
          SubjectMatch
54 }
55
56 Subject ::= SubjectType
57
58 SubjectsType ::= SEQUENCE {
59    subject-list [0] SEQUENCE (SIZE (1..MAX)) OF subject Subject
60 }
61
62 Subjects ::= SubjectsType
63
64 AttributeDesignatorType ::= SEQUENCE {
65    attributeId [0] UTF8String,
66    dataType [1] UTF8String,
67    issuer [2] UTF8String OPTIONAL,
68    mustBePresent [3] BOOLEAN DEFAULT FALSE
69 }
70
71 AttributeDesignatorType-derivations ::= CHOICE {
72    attributeDesignatorType [0] AttributeDesignatorType,
73    subjectAttributeDesignatorType [1] SubjectAttributeDesignatorType
74 }
75
76 ResourceAttributeDesignator ::= AttributeDesignatorType-derivations
77
78 ResourceMatchType ::= SEQUENCE {
79    matchId [0] UTF8String,
80    attributeValue [1] AttributeValue,
81    choice [2] CHOICE {
82       resourceAttributeDesignator [0] ResourceAttributeDesignator,
83       attributeSelector [1] AttributeSelector
84    }
85 }
86
87 ResourceMatch ::= ResourceMatchType
88
89 ResourceType ::= SEQUENCE {
90    resourceMatch-list [0] SEQUENCE (SIZE (1..MAX)) OF resourceMatch
91       ResourceMatch
92 }
93
94 Resource ::= ResourceType
95
96 ResourcesType ::= SEQUENCE {
97    resource-list [0] SEQUENCE (SIZE (1..MAX)) OF resource Resource
98 }
99
100 Resources ::= ResourcesType
101
102 ActionAttributeDesignator ::= AttributeDesignatorType-derivations
103
104 ActionMatchType ::= SEQUENCE {
105    matchId [0] UTF8String,
106    attributeValue [1] AttributeValue,
107    choice [2] CHOICE {
108       actionAttributeDesignator [0] ActionAttributeDesignator,
109       attributeSelector [1] AttributeSelector
110    }
111 }
112
```

```
113 ActionMatch ::= ActionMatchType
114
115 ActionType ::= SEQUENCE {
116    actionMatch-list [0] SEQUENCE (SIZE (1..MAX)) OF actionMatch
           ActionMatch
117 }
118
119 Action ::= ActionType
120
121 ActionsType ::= SEQUENCE {
122    action-list [0] SEQUENCE (SIZE (1..MAX)) OF action Action
123 }
124
125 Actions ::= ActionsType
126
127 EnvironmentAttributeDesignator ::= AttributeDesignatorType-derivations
128
129 EnvironmentMatchType ::= SEQUENCE {
130    matchId [0] UTF8String,
131    attributeValue [1] AttributeValue,
132    choice [2] CHOICE {
133       environmentAttributeDesignator [0] EnvironmentAttributeDesignator,
134       attributeSelector [1] AttributeSelector
135    }
136 }
137
138 EnvironmentMatch ::= EnvironmentMatchType
139
140 EnvironmentType ::= SEQUENCE {
141    environmentMatch-list [0] SEQUENCE (SIZE (1..MAX)) OF environmentMatch
142       EnvironmentMatch
143 }
144
145 Environment ::= EnvironmentType
146
147 EnvironmentsType ::= SEQUENCE {
148    environment-list [0] SEQUENCE (SIZE (1..MAX)) OF environment
           Environment
149 }
150
151 Environments ::= EnvironmentsType
152
153 TargetType ::= SEQUENCE {
154    subjects [0] Subjects OPTIONAL,
155    resources [1] Resources OPTIONAL,
156    actions [2] Actions OPTIONAL,
157    environments [3] Environments OPTIONAL
158 }
159
160 Target ::= TargetType
161
162 PolicyDefaults ::= DefaultsType
163
164 CombinerParameterType ::= SEQUENCE {
165    parameterName [0] UTF8String,
166    attributeValue [1] AttributeValue
167 }
168
169 CombinerParameter ::= CombinerParameterType
170
171 CombinerParametersType ::= SEQUENCE {
```

```
172     combinerParameter-list [0] SEQUENCE OF combinerParameter
            CombinerParameter
173 }
174
175 RuleCombinerParametersType ::= SEQUENCE {
176     ruleIdRef [0] UTF8String,
177     combinerParameter-list [1] SEQUENCE OF combinerParameter
            CombinerParameter
178 }
179
180 PolicyCombinerParametersType ::= SEQUENCE {
181     policyIdRef [0] UTF8String,
182     combinerParameter-list [1] SEQUENCE OF combinerParameter
            CombinerParameter
183 }
184
185 PolicySetCombinerParametersType ::= SEQUENCE {
186     policySetIdRef [0] UTF8String,
187     combinerParameter-list [1] SEQUENCE OF combinerParameter
            CombinerParameter
188 }
189
190 CombinerParametersType-derivations ::= CHOICE {
191     combinerParametersType [0] CombinerParametersType,
192     ruleCombinerParametersType [1] RuleCombinerParametersType,
193     policyCombinerParametersType [2] PolicyCombinerParametersType,
194     policySetCombinerParametersType [3] PolicySetCombinerParametersType
195 }
196
197 CombinerParameters ::= CombinerParametersType-derivations
198
199 RuleCombinerParameters ::= RuleCombinerParametersType
200
201 VariableReferenceType ::= SEQUENCE {
202     variableId [0] UTF8String
203 }
204
205 VariableReference ::= VariableReferenceType
206
207 FunctionType ::= SEQUENCE {
208     functionId [0] UTF8String
209 }
210
211 Function ::= FunctionType
212
213 ApplyType ::= SEQUENCE {
214     functionId [0] UTF8String,
215     expression-list [1] SEQUENCE OF expression Expression-group
216 }
217
218 Apply ::= ApplyType
219
220 Expression-group ::= CHOICE {
221     variableReference [0] VariableReference,
222     attributeSelector [1] AttributeSelector,
223     resourceAttributeDesignator [2] ResourceAttributeDesignator,
224     actionAttributeDesignator [3] ActionAttributeDesignator,
225     environmentAttributeDesignator [4] EnvironmentAttributeDesignator,
226     subjectAttributeDesignator [5] SubjectAttributeDesignator,
227     attributeValue [6] AttributeValue,
228     function [7] Function,
229     apply [8] Apply
```

```
}

VariableDefinitionType ::= SEQUENCE {
   variableId [0] UTF8String,
   expression [1] Expression-group
}

VariableDefinition ::= VariableDefinitionType

EffectType ::= ENUMERATED { deny(0), permit(1) }
ConditionType ::= SEQUENCE {
   expression [0] Expression-group
}

Condition ::= ConditionType

RuleType ::= SEQUENCE {
   effect [0] EffectType,
   ruleId [1] UTF8String,
   description [2] Description OPTIONAL,
   target [3] Target OPTIONAL,
   condition [4] Condition OPTIONAL
}

Rule ::= RuleType

AttributeAssignmentType ::= SEQUENCE {
   attributeId [0] UTF8String,
   dataType [1] UTF8String,
   attr [2] SEQUENCE OF UTF8String
}

AttributeAssignment ::= AttributeAssignmentType
ObligationType ::= SEQUENCE {
   fulfillOn [0] EffectType,
   obligationId [1] UTF8String,
   attributeAssignment-list [2] SEQUENCE OF attributeAssignment
      AttributeAssignment
}

Obligation ::= ObligationType
ObligationsType ::= SEQUENCE {
   obligation-list [0] SEQUENCE (SIZE (1..MAX)) OF obligation Obligation
}

Obligations ::= ObligationsType

PolicyType ::= SEQUENCE {
   policyId [0] UTF8String,
   ruleCombiningAlgId [1] UTF8String,
   version [2] UTF8String OPTIONAL,
   description [3] Description OPTIONAL,
   policyDefaults [4] PolicyDefaults OPTIONAL,
   combinerParameters [5] CombinerParameters OPTIONAL,
   target [6] Target,
   choice-list [7] SEQUENCE OF CHOICE {
      combinerParameters [0] CombinerParameters OPTIONAL,
      ruleCombinerParameters [1] RuleCombinerParameters OPTIONAL,
      variableDefinition [2] VariableDefinition,
      rule [3] Rule
   } OPTIONAL,
   obligations [8] Obligations OPTIONAL
```

```
}

Policy ::= PolicyType

VersionMatchType ::= UTF8String

IdReferenceType ::= SEQUENCE {
   earliestVersion [0] UTF8String OPTIONAL,
   latestVersion [1] UTF8String OPTIONAL,
   version [2] UTF8String OPTIONAL,
   base [3] UTF8String
}

PolicySetIdReference ::= IdReferenceType

PolicyIdReference ::= IdReferenceType

PolicyCombinerParameters ::= PolicyCombinerParametersType

PolicySetCombinerParameters ::= PolicySetCombinerParametersType

PolicySetType ::= SEQUENCE {
   policyCombiningAlgId [0] UTF8String,
   policySetId [1] UTF8String,
   version [2] UTF8String OPTIONAL,
   description [3] Description OPTIONAL,
   policySetDefaults [4] PolicySetDefaults OPTIONAL,
   target [5] Target,
   choice-list [6] SEQUENCE OF CHOICE {
      policySet [0] PolicySet,
      policy [1] Policy,
      policySetIdReference [2] PolicySetIdReference,
      policyIdReference [3] PolicyIdReference,
      combinerParameters [4] CombinerParameters,
      policyCombinerParameters [5] PolicyCombinerParameters,
      policySetCombinerParameters [6] PolicySetCombinerParameters
   },
   obligations [7] Obligations OPTIONAL
}

PolicySet ::= PolicySetType

ExpressionType ::= SEQUENCE {

}

Expression ::= ExpressionType

ExpressionType-derivations ::= CHOICE {
   attributeDesignatorType [0] AttributeDesignatorType,
   subjectAttributeDesignatorType [1] SubjectAttributeDesignatorType,
   attributeSelectorType [2] AttributeSelectorType,
   variableReferenceType [3] VariableReferenceType,
   functionType [4] FunctionType,
   applyType [5] ApplyType
}

END
```

Listing D.1: Policy decision module Native Language (PNL) based on ASN.1 Defintion

Appendix E

Résumé en Français

E.1 Contexte

La conception des systèmes sécurisés a été toujours une tâche complexe. En pratique, beaucoup d'effort a été fourni par les concepteurs et les développeurs afin de définir et délivrer un système de travail. Concernant la sécurité de ces systèmes, l'approche considérée était toujours rétroactive qui s'applique qu'après la détection d'un ensemble de lacunes. Les experts en sécurité sont donc généralement confrontés à un système existant, dont l'architecture pourrait entraver le déploiement de mécanismes de sécurité, ce qui empêcherait l'apparition des attaques qu'ils envisagent. Une approche qui permet d'éviter ce type de problèmes est le développement d'une architecture de sécurité qui définie des exigences axées sur la sécurité et qui décrit une collaboration structurée et une interdépendance entre la conception de l'architecture et les exigences de la sécurité (SR) pour répondre aux besoins à long terme des systèmes [138]. Le but d'une architecture de sécurité est traditionnellement de mettre en évidence les principaux domaines de préoccupation en soulignant les critères de décision et le contexte de sécurité pour chaque aspect du système qui peut avoir une valeur directe ou indirecte pour un acteur. Le concept d'une architecture de sécurité englobe diverses notions techniques dans lesquelles la sécurité est introduite à différents niveaux d'abstraction et fondée sur des mécanismes différents. Thorn et al. [145] décrivent une architecture de sécurité comme «une conception cohérente de la sécurité, qui répond aux exigences de sécurité (par exemple, l'authentification, l'autorisation, etc.) et en particulier aux risques d'un environnement ou scénario spécifique, et spécifie les contrôles de sécurité qui doivent être appliqués et où est ce qu'ils peuvent être appliqués". À cette fin, l'un des aspects clés d'une architecture de sécurité comme étant un outil d'une conception sécurisée c'est de fournir un framework des exigences de sécurité (SRE: Security Requirement Engineering) à travers lequel des exigences réalistes et concrètes peuvent être identifiées et mises en œuvre.

Du point de vue de système embarqué, cette activité, SRE, devient encore plus critique et présente des défis. Ces défis découlent de la relation étroite entre la conception de l'architecture et de ses exigences fonctionnelles et non fonctionnelles ainsi que de leur impact sur l'autre. Par exemple, si la conception de l'architecture du système évolue, les exigences de sécurité devraient atteindre les nouveaux objectifs de la nouvelle conception de l'architecture. C'est particulièrement vrai lorsque ces systèmes font partie intégrante des systèmes critiques de sécurité tels que les systèmes automobiles [129, 9]. Ceci est lié à Koscher et al. [80] citation, "les systèmes automobiles doivent non seulement être extrêmement fiable et sans défaut, mais aussi extrêmement résistant aux menaces et l'exploitation des vulnérabilités". Plus précisément, les applications de sécurité doivent être assurés contre les attaques malveillantes. Plusieurs activités de recherche ont décrit les vulnérabilités

potentielles et contre-mesures dans les systèmes automobiles, par exemple, [51, 14], que nous allons en faire référence dans la suite de cette thèse. À quelques exceptions près, la plupart de ces efforts considèrent les SR d'une façon abstraite, seule l'étape d'identification des exigences est considérée, et ne visent pas particulièrement le raffinement des exigences, et les propriétés de traçabilité des exigences. Cependant, il y a des approches bien reconnues comme KAOS [152] ou UMLsec [69] qui ont déjà montré des résultats intéressants dans le domaine de SRE pour gérer les problèmes de sécurité. Pourtant, avant de considérer ces approches, nous devons d'abord faire une distinction claire entre ce que nous entendons par système embarqué et quelles sont leurs fonctions, ainsi que les problèmes de sécurité non-fonctionnels.

En général, les systèmes embarqués sont définis comme une combinaison de matériel et de logiciels qui forment une partie d'un système plus vaste et sont généralement conçus pour accomplir une tâche spécifique. Plus précisément, ce qui rend les systèmes embarqués distribués différent des systèmes à usage général sont des caractéristiques spécifiques : ces systèmes ont des ressources limitées concernant leurs capacités (et par conséquent dans leurs défenses). Ils ont des problèmes de fiabilité et de performance, ainsi que les contraintes de calcul en temps réel. Ces systèmes sont souvent portables ou mobiles, et ils sont facilement accessibles aux adversaires au niveau de la couche physique. Cette accessibilité a donné lieu à plusieurs nouvelles attaques de sécurité au cours de ces dernières années [80, 163, 164]. Par exemple, Koscher et al. [80] démontrent la capacité de contrôler un large nombres de fonctions automobiles tout en ignorant les données du conducteur. Ces attaques ont été faites par le simple accès aux diagnostics internes sur portuaires (OBD- II) et l'incorporation d'un code malveillant dans l'unité télématique d'une voiture. Cela permet à un adversaire de contrôler pratiquement différentes fonctionnalités de bord - y compris la désactivation des freins, un freinage sélectif des roues individuelles sur demande, l'arrêt du moteur, et ainsi de suite. En outre, l'aspect le plus important du système embarqué tel que c'est défini par Noergaard [100]:

"*. . . aucun des éléments à l'intérieur d'un système embarqué fonctionne en vase clos. Chaque élément au sein d'un dispositif interagit avec un autre élément d'une certaine façon. En outre, plusieurs caractéristiques qui sont visibles de l'extérieur des éléments peuvent changer étant donné un ensemble d'autres éléments qui sont censés fonctionner avec. Sans comprendre les «pourquoi» fournies derrière la fonctionnalité d'un élément, sa performance, etc, il serait difficile de déterminer comment le système se comporte dans une variété de circonstances dans le monde réel*".

Du point de vue sécurité, cette définition implique que, pour un système intégré, pour être sécurisé, chaque élément ainsi que ses relations avec les autres éléments à différents niveaux d'abstraction (par exemple , application, au niveau du protocole, au niveau du middleware, le niveau infrastructure, niveau de stockage, et ainsi de suite) doivent être sécurisés . Par exemple, l'unité de contrôle électrique (ECU), peut compter sur un module de sécurité matériel pour le traitement des opérations cryptographiques, mais si les couches supérieures (par exemple, la couche de middleware) gèrent ces attributs d'authentification différemment et permettent à l'adversaire de fausser ces attributs (c'est à dire , les tickets d'authentification) , la sécurité globale est violée. En particulier, nous pouvons identifier les failles de sécurité et les problèmes en examinant les interactions et les collaborations entre les couches subtiles. De même, plusieurs approches [79, 55, 164, 161] ont montré que nous ne pouvons pas résoudre le problème de sécurité d'un système intégré à un seul niveau d'abstraction. Par conséquent, il est naturel de développer une spécification des exigences de sécurité en mettant l'accent sur les caractéristiques distinctes des systèmes embarqués

et en particulier en tenant compte d'une structuration en couches pour la représentation d'une architecture de système embarqué, qui peut nous aider à développer une architecture de sécurité modulaire. Dans ce contexte, l'état actuel des approches de SRE, comme KAOS et UMLsec, sont loin de Capturer le fonctionnement de base des architectures de systèmes embarqués. Par exemple, le cadre KAOS se concentre principalement sur la satisfaction des objectifs et sur la synthèse de modèles de comportement [151] et ne considère pas l'architecture système du système. Par exemple, il est difficile de capturer et de modéliser dans KAOS une architecture et encore moins plusieurs couches architecturales. En revanche, UMLsec, qui est une approche d'ingénierie dirigée par les modèles, considère à la fois les aspects structurels et comportementaux de SR. Cette approche considère que les exigences bien formées ont déjà été obtenues et raffinées jusqu'au niveau de la conception à travers la définition des comportements normaux des composants du système. UMLsec se concentre plus particulièrement sur l'amélioration de ces conditions de sécurité dans les mécanismes de sécurité.

E.2 Contributions de la thèse

Dans cette thèse, nous proposons une approche pour la gestion des exigences de sécurité (SRE) qui permet la conception d'une architecture de sécurité pour les systèmes embarqués. Nous mettons l'accent sur l'acquisition de connaissances liées à la sécurité et à la gestion à travers la définition d'un processus de SRE qui permet de concevoir un système qui est intrinsèquement sécurisé dès sa conception. Notre approche se compose de trois parties successive:

1. **Approche basée sur les connaissances pour l'ingénierie des exigences de la sécurité:** Dans la première phase de cette thèse, nous présentons les principaux éléments constitutifs de notre méthodologie SRE proposée et nous discutons de son intégration avec un système d'ingénierie à travers un langage de modélisation.

    - Nous avons analysé systématiquement diverses sources telles que les normes de sécurité, un ensemble de méthodologies représentant l'état actuel des approches existantes, afin de construire une méthodologie unifiée de SRE. La méthodologie proposée montre comment les capacités des modèles et des approches des différents SRE peuvent être intégrés à un processus de SRE axé sur la connaissance. En outre, nous considérons les concepts clés définis dans ces approches et nous construisons des ontologies de sécurité pour chaque concept afin de guider notre processus de SRE avec une base de connaissances. Ainsi, il sera possible d'analyser différents concepts de sécurité et de permettre une structuration particulière, une réutilisation et une base de connaissances sur les concepts de sécurité qui peut être partagée dans le processus de SRE. Bien que notre méthodologie proposée soit dédiée aux systèmes embarqués, il est encore assez souple pour l'adapter à tout type d'architecture de système d'usage général comme dans le cadre des architectures orientées services (SOA) [115], et aussi capables de produire des exigences de sécurité précises. Les résultats

sont présentés dans [60].

- Nous avons d'abord exploré les capacités de SysML, le système Modeling Language [107] pour supporter notre méthodologie de SRE basée sur la connaissance. SysML est un standard OMG [106]pour le système de modélisation des applications d'ingénierie et a une expressivité suffisante pour décrire une conception détaillée du système. Cependant, une faiblesse majeure pour l'utilisation de SysML est le manque des aspects orientés sécurité. Pour tirer profit des capacités de SysML, nous avons proposé plusieurs extensions à la sémantique SysML pour pouvoir intégrer nos concepts de sécurité. En particulier, nous avons intégré l'exigence de sécurité et les schémas d'attaque de sécurité. De plus, nous avons enrichi ces diagrammes avec nos concepts ontologiques proposés, comme un vocabulaire contrôlé. L'utilisation des ontologies dans les langages de modélisation offre une occasion concrète de raisonner sur l'exactitude de ces modèles. En outre, nous avons implémenté ces fonctionnalités dans le moteur TTool [82] qui supporte également notre modèle SysML étendu pour définir les exigences de sécurité et de la modélisation de l'arbre d'attaque . Cet outil prend en charge facilement la méthodologie itérative que nous préconisons.

2. **Conception des exigences de sécurité:** Dans la deuxième partie de cette thèse, nous présentons chaque activité du processus des exigences de sécurité (SREP) en détails et nous expliquons comment une base de connaissances liées à la sécurité est générée et partagée entre toutes les activités.

   - Afin d'illustrer les différentes parties de cette thèse, nous introduisons un exemple pratique - sur la mise à jour de fireware - utilisé tout au long de la thèse pour expliquer nos propositions. L'exemple est originaire de la conception sécurisée d'un système embarqué de véhicule développé dans le projet européen EVITA [117]. L'étude de cas a été développée pour illustrer le processus de mise à jour du firmware sécurisé.

   - Nous abordons le problème de l'identification des attaques de sécurité et les vulnérabilités dans le cadre d'une architecture de système multicouche, où l'information liée à la sécurité est générée, traitée et stockée à différents niveaux. L'idée est d'extraire les connaissances sur les différentes activités du système et qui correspondent à diverses activités de développement du système, et d'utiliser ces connaissances à des fins d'analyse de la sécurité. En particulier, nous utilisons les bases de connaissances en s'appuyant sur différentes ontologies tels que l'ontologie de l'architecture du système, l'ontologie des objectifs, etc pour analyser la sécurité du système et pour spécifier comment un adversaire peut attaquer le système. En outre, le concept d'un arbre d'attaque basée sur la connaissance est introduit dans la représentation graphique fondamentale pour la modélisation d'attaque.

   - Nous avons illustré l'approche dans le cadre de l'identification et de raffinement des exigences de sécurité, et nous avons présenté un moyen de retracer les exigences de sécurité. Nous décrivons d'abord le processus d'identification des exigences de sécurité qui utilise les différentes bases de connaissances pro-

duites durant les différentes phases de la SREP. Il nous permet de découvrir les exigences de sécurité du stade de développement du système tout au début et par rapport aux différentes bases de connaissances disponibles. Ensuite, nous proposons le concept de modèle de raffinement pour remédier à certaines insuffisances et limites des approches existantes pour le raffinement des SR. Enfin, nous proposons une approche pour tracer les exigences afin de déterminer leurs sources et les raisons pour leurs existences. Nous utilisons en particulier nos diagrammes d'exigences de sécurité SysML étendu pour modéliser et partager des connaissances connexes sur les SR.

3. **Mise en oeuvre des exigences de sécurité:** Dans la troisième et dernière partie de cette thèse, nous traitons la mise en œuvre des SR et nous proposons des solutions pour la conception et le déploiement de protocoles cryptographiques et pour la mise en œuvre des exigences de sécurité liées au contrôle d'accès.

   - Nous avons proposé une approche basée sur l'utilisation des clés cryptographiques protégés avec du matériel peu coûteux pour construire le firmware montrant la spécification du protocole cryptographique. Nous montrons comment une racine de confiance dans le matériel peut être raisonnablement combinée avec des modules logiciels. Ces modules et primitives ont été appliquées pour montrer comment les mises à jour du firmware peut être faite en toute sécurité, tout en respectant les normes et les infrastructures existantes. Malgré le fait que le modèle de plateforme de confiance implique certaines contraintes, telles que l'obligation d'intégrer des clés cryptographiques à une configuration de démarrage donnée, nous montrons comment les protocoles que nous avons présenté gèrent la mise à jour des registres de référence de la plateforme pendant la phase de démarrage d'une unité de contrôle électronique - ECU.

   - La dernière contribution de cette thèse est consacrée à la mise en oeuvre des exigences de sécurité concernant le contrôle d'accès. Nous avons proposé et développé un module de décision de la politique qui est utilisé pour appliquer les différentes règles de contrôle d'accès en déployant plusieurs points d'application pour les différents niveaux d'abstraction du système. Nous discutons la façon de mettre en œuvre les politiques qui implémentent une application effective dans de telles architectures, malgré la complexité des piles de protocoles au bord des unités de contrôle électronique. On évalue également comment les politiques exprimées dans XACML peuvent être adaptées aux exigences d'efficacité dans des environnements automobiles malgré la puissance de calcul limitée de leurs unités et leurs limites de bande passante au niveau réseau.