Evolution of Security Engineering Artifacts: A State of the Art Survey

Michael Felderer, University of Innsbruck, Austria
Basel Katt, University of Innsbruck, Austria
Philipp Kalb, University of Innsbruck, Austria
Jan Jürjens, Technical University of Dortmund, Germany
Martín Ochoa, Technical University of Munich, Germany
Federica Paci, University of Trento, Italy
Le Minh Sang Tran, University of Trento, Italy
Thein Than Tun, The Open University, UK
Koen Yskout, iMinds-DistriNet, KU Leuven, Belgium
Riccardo Scandariato, iMinds-DistriNet, KU Leuven, Belgium
Frank Piessens, iMinds-DistriNet, KU Leuven, Belgium
Dries Vanoverberghe, iMinds-DistriNet, KU Leuven, Belgium
Elizabeta Fourneret, University of Luxembourg, Luxembourg
Matthias Gander, University of Innsbruck, Austria
Bjørnar Solhaug, SINTEF ICT, Norway
Ruth Breu, University of Innsbruck, Austria

Abstract: *Security is an important quality aspect of modern open software systems. However, it is challenging to keep such systems secure because of evolution. Security evolution can only be managed adequately if it is considered for all artifacts throughout the software development lifecycle. This article provides state of the art on the evolution of security engineering artifacts. The article covers the state of the art on evolution of security requirements, security architectures, secure code, security tests, security models, and security risks as well as security monitoring. For each of these artifacts we give an overview of evolution and security aspects and discuss the state of the art on its security evolution in detail. Based on this comprehensive survey, we summarize key issues and discuss directions of future research.*

Keywords: *Software Evolution, Change Management, Security Evolution, Security Change Management, Security Engineering Artifacts, Secure Software Development Lifecycle, State of the Art Survey*

Introduction

Due to ever changing surroundings, new business needs, new regulations and new technologies, a software system must evolve, or it becomes progressively less satisfactory (Lehman, 1980, 1998). On the one hand, the continuous system evolution makes it especially challenging to keep software systems permanently secure as changes, either in the system itself or in its environment, may cause new threats and vulnerabilities. On the other hand, security artifacts themselves like security requirements, security architectures, and secure code or security tests have to be continuously adapted in long-running software systems. Because modern open and

dynamically-changing software systems like service-oriented architectures or cloud deployments determine business process implementations and deal with critical data, managing the evolution of their security artifacts in all phases of the software development lifecycle (SDLC) is of high importance.

The main phases of the SDLC are *analysis, design, implementation, testing*, as well as *deployment* and *operation* (Braude & Bernstein, 2011). In each phase, specific artifacts are created or adapted, i.e., requirements in the analysis phase, the architecture in the design phase, source code in the implementation phase, tests in the testing phase, as well as the running system in the deployment and operation phase. All these artifacts are subject to changes which is one of the main difficulties of software evolution (Mens & Demeyer, 2008) with high impact on security engineering.

Security engineering focuses on security aspects in the software development lifecycle. Security aims at protecting information and systems from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction. The main objective of security is to guarantee confidentiality, integrity and availability of information and systems. To be most effective, security must be integrated into the software development lifecycle from the very beginning (Kissel et al., 2008).

Risk management in general is the process allowing organizations to identify what assets need to be protected, what threats prevail and with what probability and severity losses could occur. As such it is an indispensable activity in security engineering to identifying and to manage threats and vulnerabilities to information as well as systems.

Model engineering involves the systematic use of models as essential artifacts throughout the software development process (Schmidt, 2006). It has recently been applied in security engineering to provide security models for all phases of the software development lifecycle to manage the evolution of security engineering artifacts.

Figure 1 gives an overview of the security engineering activities and the assigned artifacts in the secure software development lifecycle. In each iteration, the activities analysis, design, implementation, development as well as deployment are performed consecutively. Additionally, risk management and model engineering accompany these activities. As the system and its environment evolve, all these activities are executed iteratively which is represented by the surrounding border. Each phase handles specific artifacts, i.e., requirements, architecture, code, tests, running system, models and risks.

Each of these artifacts corresponds to sections in this article with respect to its security-specific evolution aspects. Managing the evolution of security engineering artifacts is an important task that needs specific approaches to continuously guarantee security.

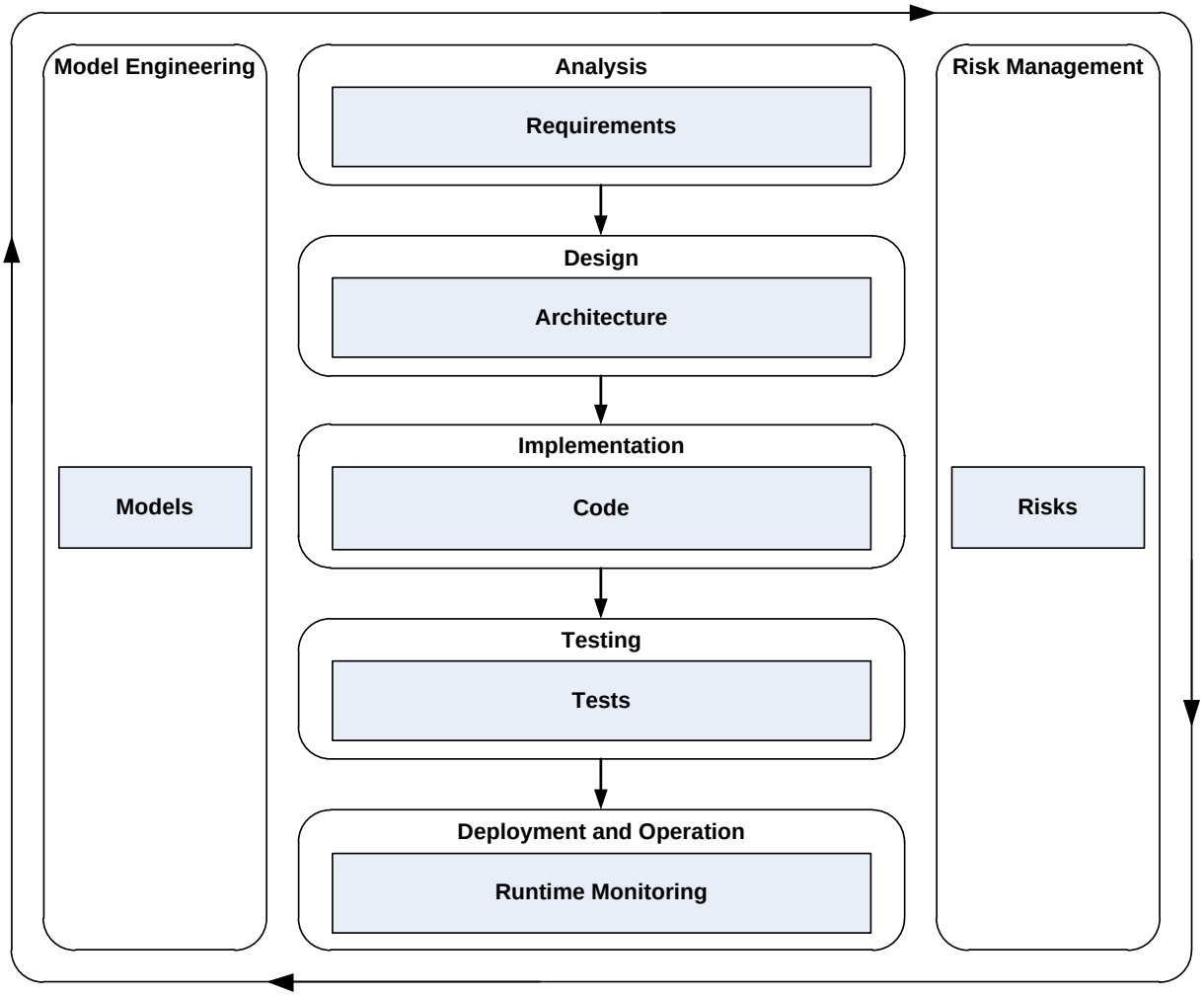

Figure 1. Security Engineering Activities, Security Artifacts and Section Overview

This article reviews the state of the art of evolution management of security engineering artifacts and draws conclusions for future research. Section *Security Model Evolution* discusses security model evolution, and Section *Security Requirements Evolution* covers evolution of security requirements. Section *Security Architecture Evolution* discusses security architecture evolution, while Section *Secure Code Evolution* analyzes evolution of secure code. Then, Section *Security Test Evolution* discusses evolution aspects of testing. Section *Security Monitoring Evolution* explains methods to control security monitoring of a running system. Section *Security Risk*s shows approaches for security risk evolution. Finally, we summarize the state of the art and sketch future directions of research. Each section follows a similar structure. After a short introduction to key aspects of the respective artifact, we discuss its security engineering and evolution. Finally, we combine these two aspects and discuss the state of the art of its security evolution in detail. Each section was written by proven experts in the respective area the article has therefore overall 16 authors who compiled the literature for the particular artifacts based on their in-depth knowledge and experience. This state of the art survey on evolution of security engineering artifacts is partially based on a previous state of the art survey (Felderer, Kalb, et al., 2011).

Security Model Evolution

Modern software and security engineering, uses models to define software systems and their components on a high level of abstraction. The de-facto industrial standard for modeling both structural and behavioral aspects of IT systems is the Unified Modeling Language (UML). In this section, we therefore focus on UML models. As modern IT systems are under continuous change, also the models change and evolve over time. As an implication of using models for security aspects, evolution has to be considered also for security concepts and methodologies. In the context of security there are at least two different dimensions of model evolution, i.e., target and granularity of evolution.

The first dimension, i.e., *target of evolution*, can be classified into two categories, i.e., *security properties* and *artifact* evolution. On the one hand, the security properties linked to an artifact can change, which cannot be avoided for life-long systems. This can have multiple reasons such as changes in the business security objectives and standards or the identification of new threats. One the other hand, the opposite may happen as well, if an artifact changes while its security properties remain untouched. In the context of this survey, we will consider evolution as changes made to an artifact.

The second dimension, i.e., *granularity of evolution*, has two categories as well, i.e., *fine-grained* evolution and *coarse-grained*. *Fine grained evolution* is used to cope up with changes in systems (for example, new or extended functionality). Several general-purpose evolution specification or model transformation approaches exist, for instance, (Heckel, 1998; Andries et al., 1999). Jürjens et al. (2011) discuss the use of stereotypes for annotating several future change possibilities in UML. The changes can be coordinated by constraints in first order logic. France & Bieman (2001) discuss a multi-view approach supporting cyclic evolution of object-oriented UML models. Breu et al. (2010) show how state machines are defined that control the lifecycle of specific model elements where changes to the state of a model element may propagate further stage changes on other model elements with the purpose of coordinating the tasks of various stakeholders in the maintenance process of complex models.

Coarse-grained evolution is a similar context with a different level of abstraction which deals with evolving architectures. Garlan et al. (Garlan, Barnes, Schmerl, & Celiku, 2009a) discuss different evolution styles for high-level architectural views of the system. It also discusses the possibility of having more than one evolution path and describes tool support for choosing the "correct" paths with respect to properties described in temporal logic.

In the course of this section we will continue by analyzing the current state of the art of security engineering using UML models in context of the previously defined security notions. This is followed by modern approaches to include model evolution in security considerations.

Security Engineering with UML

Models can be very useful to improve the quality of systems at early development stages. In particular, security issues associated with design can be spotted already in models. There exist several lines of research towards using UML for security systems development discussed in the following paragraphs.

UMLsec. *UMLsec* (Jürjens, 2005) is a comprehensive approach covering the aforementioned core of UML diagrams with respect to different security properties. UMLsec is a light-weight extension of UML using stereotypes, tags, and constraints, to

specify typical security requirements such as secrecy, integrity, or authenticity, and attacker models. Together with a formal semantics for a core of UML, it is possible to reason about Dolev-Yao secrecy in protocols specified using sequence diagrams or information flow in state charts. It therefore falls in almost all categories of our classification, availability being the less studied aspect in the context of UMLsec.

Access Control. Most of the work on security and UML focuses on access control mechanisms. SecureUML (Basin, Doser, & Lodderstedt, 2003; Basin, Doser, & Lodderstedt, 2006) shows how UML can be used to specify access control in an application and how one can then generate access control mechanisms from the specifications. The approach is based on role-based access control and gives additional support for specifying authorization constraints. Hawkins and Fernandez (1997) extend use cases and interaction diagrams to support distributed system architecture requirements. Brose et al. (2002) demonstrate how to deal with access control policies in UML. The specification of access control policies is integrated into UML. A graph-based formal semantics for the UML access control specification permits one to reason about the coherence of the access control specification.

Georg et al. (Georg, France, & Ray, 2002) demonstrate how to use UML for aspect-oriented development of security-critical systems. Design-level aspects are used to encapsulate security concerns that can be woven into the models. In Georg et al. (2003), authentication mechanism models are considered in an abstract aspect model and more detailed models are created from these. The models can be composed with primary decomposition models, allowing system architects to analyze different mechanisms to realize a particular concern, such as authentication.

Ray et al. (2003) propose to use aspect-oriented modeling for addressing access control concerns. Functionality that addresses a pervasive access control concern is defined in an aspect. The remaining functionality is specified in a so-called primary model. Composing access control aspects with a primary model then gives a system model that addresses access control concerns. Kim et al. (2004) use a variant of UML to model Role Based Access Control and Mandatory Access Control to compose access control policy frameworks.

Alghathbar and Wijesekera (2003) suggest a method for specifying access control policies with UML use cases and propose a methodology to resolve some issues of consistency and completeness of access control specifications.

Information Flow control. Heldal and Hultin (2003) provide support for the use of UML with secrecy annotations so that the code produced from the UML models can be validated by the Java information flow language-based checker. Ochoa et al. (2012) present an approximation to non-interference on UMLsec state charts. Ruhroth and Jürjens (2012) present a modular security analysis for supporting evolution using UMLsec is done based on possibilistic information flow properties as defined by Mantel (Mantel, 2002).

Availability. Availability is among the less studied aspects of security in the context of UML. Leangsuksun et al (2003) present an UML profile for general reliability that given specifications of failure rate and repair rates for components constructs a statistical model to calculate the system's reliability. Similarly Bernardi et al. (2007) describe an UML profile for dependability and analysis for real-time system. Trujillo et al. (2009) propose a UML 2.0 profile to define security requirements for Data Warehouses. Salehi et

al. (2010) discuss an UML-based domain specific language that allows specifying system configurations for an availability framework.

Authenticity and Secrecy against man-in-the-middle attackers. Apart from UMLsec, where the analysis of security protocols modelled with sequence diagrams plays an important role, Moebius et al. (2009) study an model-driven development approach for smart-cards that also focuses on man-in-the-middle attackers on the Dolev-Yao symbolic model.

Other aspects. Houmb and Hansen (2003) present SecurityAssessmentUML, a UML profile for security assessments, as well as a security assessment process with its associated documentation framework. The main objective is to support documentation of output based on risk identification and analysis in a security assessment. Blobel (2002) uses UML for modeling security-critical systems in the health sector.

Security Evolution and UML Models

Changes in models might have a tremendous impact on security because newly introduced functionality might inadvertently conflict with security requirements of the system. On the other hand, changes on security policies might render formerly secure systems insecure. However, to the best of our knowledge little research has been done on the impact of evolution on the security of UML models. In general, there are at least two change dimensions that are interesting in this context: the evolution of the security properties for a fixed model and, vice-versa, the evolution of models for fixed security properties. On the other hand, there are at least two interesting notions of evolution in models: incremental changes on models (for example adding, removing or substituting classes and attributes in class diagrams or transitions in state charts) and coarse grained changes (addition, deletion and substitution of components).

Evolution of access control policies. Role-based Access Control is one of the most studied security mechanisms in the context of UML. Montrieux et al. (2011) review the state of the art of the evolution of RBAC policies which are specified on the basis of UML models, and discusses some open challenges. On the one hand, it would be desirable that when incremental changes are made to the specification, an efficient re-verification of consistency takes place (as opposed to a trivial re-checking of the complete model). On the other hand, merging (composing) two specifications is challenging because potential discrepancies could have a security impact (for example when a role with the same name exists in both specifications). Goncalves and Poniszewska-Maranda (2008) discuss the evolution of role-based access control policies and propose an algorithm to detect potential inconsistencies when new components are added to the system. Koch et al. (2001) present algorithms for coping with the transformation and integration of Labeled Based Access Control (LBAC) and Discretionary Access Control (DAC) policies specified using graphs. Although this formalism is close to UML, to the best of our knowledge these results have not been used or extended for UML access control specifications nor for the more popular RBAC policies.

Fine-grained evolution of security properties. For models annotated with the UMLsec profile, there exists work on studying the impact of model change for fixed security properties. The UMLseCh (Jürjens et al., 2011) approach discusses sufficient conditions for the preservation of the consistency of selected UMLsec security requirements when models are incrementally transformed and presents a general strategy

for proving soundness of those conditions using inductive reasoning. The approach is illustrated with the *secure dependencies* stereotype of UMLsec that requires consistency of security annotations on communicating classes. The efficiency of the approach is quantified empirically by comparing the running time of the proposed methodology against complete re-verification. This approach is *fine-grained* and focuses on confidentiality and integrity requirements for further analysis in the context of man-in-the-middle or/and non-interference.

Coarse-grained evolution of security properties. Ochoa et al. (2012) discuss a sound decision procedure for the compositionality of Dolev-Yao secrecy in UMLsec diagrams. Here the behavior is specified using sequence diagrams. Ochoa et al. (2012) also present a compositionality theorem for Non-interference on UMLsec state charts that is valid for a particular notion of composition (that does not allow callbacks). Ruhroth and Jürjens (2012) present a modular security analysis for supporting evolution using UMLsec is done based on possibilistic information flow properties as defined by Mantel (2002).

Security Requirements Evolution

The need of considering security in the early stages of the software development lifecycle is well recognized. A major role in addressing this need is played by Security Requirement Engineering (SRE) which comprises processes, techniques and tools to elicit, model and analyze security requirements. A challenging aspect in the security requirement engineering realm is evolution. Evolution is the phenomenon where changes are introduced in a requirement model or specification to respond to changes in stakeholders' needs, in the environment where a system operates, or because new regulations and standards are introduced.

Much of the existing research has been on requirements evolution and analysis of security requirements. As such, we organize this section accordingly. In the next section, we first give a brief overview of the security requirements engineering methods existing in literature. Then, we discuss the approaches related to requirements evolution in Section *Requirements Evolution*. We conclude by presenting the works specific to security requirements evolution management in Section *Security Requirements Evolution*.

Security Requirements

Several requirements engineering approaches have been proposed and they have been surveyed in (Nhlabatsi, Nuseibeh, & Yu, 2009). We categorize them in goal-oriented, problem-based and risk-based security requirements methods. In addition, also classical modeling techniques like UML (see Section *Security Model Evolution*) can be applied.

Goal-Oriented SRE Methods. Among goal-oriented approaches, van Lamsweerde extends KAOS by introducing the notions of *obstacle* (Van Lamsweerde & Letier, 2000) and *anti-goal* (van Lamsweerde, 2004) to analyze the security concerns of a system.

Liu et al. (2003) propose an extension of the i* framework to identify attackers, and analyze vulnerabilities through actors dependency links. In this framework, all actors are considered as potential attackers, and therefore their capabilities are analyzed and possible damages caused by actors are assessed. Li et al. (Li, Liu, & Bryant, 2010) propose a formal framework to support the attacker analysis. Similarly, Elahi et al. (2009) propose extensions to i* to model and analyze the vulnerabilities affecting system requirements. Massacci et al. (2010) design SI*, a modeling framework extending the i*

framework, which aims at modeling and analyzing organizational settings and their security and dependability requirements. Similarly, Mouratidis et al. (2003) extend the Tropos methodology to include security related concepts.

Problem-based SRE Methods. Jackson (2001) has introduced the notion of Problem Frames as a way to describe the structure of recurring software problems. Extending this framework, there are lines of research that examine the security issues at requirements level. Abuse Frames (Lin, Nuseibeh, Ince, & Jackson, 2004) describe patterns of software problems viewed from an attacker's perspective. It describes the attacker's requirement, together with the software and the context in which the software might be attacked. A central idea in the Problem Frames approach is the notion of adequacy argument. This is an argument showing, that the specification, together with the descriptions of the domain properties, satisfies the requirement. Haley et al. (2008) extend and apply the notion of arguments to security requirements. Extending the work of Haley et al. (2008), Franqueira et al. (2011) suggest that, when it is prohibitive to draw on expert knowledge about security risks, some of the arguments can be based on security catalogues that are publicly available.

Risk-Based SRE Methods. Mead et al. (2005) propose SQUARE, a Security Quality Requirements Engineering (SQUARE) Methodology which considers security in the early phases of the software development lifecycle. The SQUARE process consists of nine steps. First, the requirements engineering team and project stakeholders agree on technical definitions that serve as a baseline for all future communication. Next, business and security goals are outlined. Third, artifacts and documentation are created. A risk assessment is conducted to determine the likelihood and impact of possible threats to the system. Then, the requirements engineering team selects a method for eliciting an initial set of security requirements, which are then categorized and prioritized. Finally, an inspection stage is included to ensure the consistency and accuracy of the security requirements that have been generated. A similar process to elicit security requirements has been proposed by Mellado et al. (2008). The authors introduce the Security Requirements Engineering Process (SREP), which is an asset-based and risk-driven security requirements engineering method.

Requirements Evolution

Recent works on requirements evolution aim at understanding the nature of the problem and to model it, or focus on methods and tools to assess and manage the impact of change.

As a way to understand how requirements evolve, research in the PROTEUS project (Project PROTEUS, 1996) classifies requirements into *stable* and *changing*, and further refines changing requirements into five types, which are related to the development environment, stakeholder, development processes, requirement understanding and requirement relation. In addition, the project proposes a formal representation for requirements evolution that is based on *goal-structure framework*. Later, Lam and Loomes (1998) present the EVE framework which supports a meta-model for requirements evolution. Other notable approaches include (Brier, Rapanotti, & Hall, 2006; Felici, 2004; Stark, Oman, Skillicorn, & Ameele, 1999). Brier et al. (2006) propose a problem frames based approach to help in the analysis of changes which impact an organization, in the identification and codification of recurrent change scenarios, and in the application of codified wisdom to new change problems. Felici et al. (2004) empirically investigate

the requirements evolution problem by conducting two case studies on avionics systems and smart cards. Stark et al. (Stark et al., 1999) study how change occurs in the software system and attempt to produce a prediction model of changes.

Inconsistencies Checking. Zowghi and Offen (1997) work at meta-level logic to capture intuitive aspects of managing changes to requirement models. Their approach is based on theory construction, which commences with the development of a requirements model seen as a theory of some non-monotonic logic. Requirements evolution then involves the mapping of one such theory to another. Exploiting the deductive power of the theory of belief, revision and non-monotonic reasoning, the authors develop a formal description of this mapping, as well as the requirements engineering process. Russo et al. (1999) propose an analysis and revision approach to restructure requirements to detect inconsistency and manage changes. The main idea is to allow evolutionary changes to occur first and then, in the next step, verify their impact on requirement satisfaction. Based on the same idea, d'Avila Garcez et al. (2003) target the preservation of goals and requirements during evolution. They propose an analysis which checks if a specification satisfies a given requirement. If it does not, diagnosis information is generated to guide the modification of specification in order to satisfy the requirement. Similar to d'Avila Garcez et al., Ghose's framework (Ghose, 1999) proposes an approach for handling inconsistencies due to the introduction of new requirements. The approach is based on formal default reasoning and belief revision, and it is tool supported (Ghose, 2000). Another work related to inconsistencies handling, is the one by Fabbrinni et al. (2007). They deal with controlling consistency requirements evolution expressed in natural language based on formal concept analysis.

Change Impact Analysis. Other approaches focus on analyzing the impact of requirements evolution. Chechik et al. (2009) propose a model-based approach to propagate changes between requirements and design models that utilize the relationship between the models to automatically propagate changes. Hassine et al. (2005) present an approach to change impact analysis that applies both slicing and dependency analysis at the Use Case Map specification level to identify the potential impact of requirement changes on the overall system. Lin et al. (2009) propose capturing requirement changes as a series of atomic changes in specifications and using algorithms to relate changes in requirements to corresponding changes in specifications. Label propagation has been used in goal-oriented requirements engineering to handle change (Giorgini, Massacci, & Zannone, 2005).

Evolution Management. In addition to the approaches mentioned before, there exists a number of approaches aiming to identify optimal design solutions to support requirements evolution (Bryl, Giorgini, & Mylopoulos, 2009; Heaven & Letier, 2011; Souza, Lapouchnian, & Mylopoulos, 2011; Tran & Massacci, 2011; Ernst, Borgida, & Jureta, 2011; Letier & van Lamsweerde, 2004). These approaches address requirements evolution using *adaptation* or *mutation* mechanisms. The adaptation mechanism refers to the case that system changes its behavior at runtime in order to continue to meet its requirements in response to feedback. The success (or failure) of a behavior depends on *control variables* and *indicators*. The former determine respective resource allocation for fulfilling requirements and the latter measure the quality of satisfaction or performance. As result, it raises a problem, namely Control Variables and Indicators (CV&I),

that consists of finding a design solution that is good enough with respect to one or multiple indicators. Studies in this realm include (Bryl et al., 2009; Heaven & Letier, 2011; Souza et al., 2011). Bryl et al. (2009) propose an approach that generates and evaluates multiple requirement models that can meet the stakeholders' desire to find a right trade-off between the technical and social dimensions. Letier et al. (2004) develop the *quantitative goal model* that extends the KAOS (van Lamsweerde, 2009) framework by annotating goals with quantitative attributes. On this basis, the same authors (Heaven & Letier, 2011) developed techniques that automate the evaluation of requirements satisfaction of different alternative system designs in quantitative goal models. In another work, Souza et al. (2011) propose a systematic system identification method for adaptive software system. In this approach, the dynamic behavior of the system is governed by a set of (in)equations, called *qualitative differential constraints*.

The mutation mechanism is the case that the system changes in response to changes to its requirements. At runtime, the requirements of a system might change and some old requirements are obsolete. Mutation thus comprises two classes of requirements changes: *known unknown* and *unknown*. The known unknown class includes changes that are anticipated (the known), but it is not sure whether these changes will actually happen (the unknown). Tran and Massacci (Tran & Massacci, 2011) proposed an approach to handle known unknown changes. The approach captures requirements evolution in terms of evolution rules, including controllable evolution rule and observable evolution rule. The approach also supports a reasoning to measure the level of usefulness of different design solutions based on two metrics, i.e., MaxBelief and ResidualRisk. MaxBelief measures the maximum evolution probability that a solution is still useful after evolution occurs. ResidualRisk is the probability that a solution becomes useless after evolution. Ernst et al. (2011) instead propose an approach to handle unknown changes, that are unanticipated changes and, thus, they cannot be modeled. They present algorithms to find new solutions that use as much as possible of the old solution (i.e., maximize familiarity), and minimize the number of tasks that need to be implemented (i.e., minimize effort).

Security Requirements Evolution

The issue of maintaining security while introducing changes to a requirements model or specification has not been extensively studied. The only work we are aware of that investigates this problem, is Bergmann et al. (2011), which presents the SecureChange Methodology for Evolutionary Requirements (SeCMER), a model driven engineering methodology to represent, analyze and detect security issues that arise because of requirements' evolution. The core features of the methodology are:

- a conceptual model for evolving security requirements (Massacci, Mylopoulos, Paci, Tun, & Yu, 2011),
- a pattern-based analysis to automatically detect changes in a requirement model that lead to a violation of security principles, e.g., least privilege and need to know, and
- argumentation-based analysis to assess the impact of changes to security.

The conceptual model allows explicitly linking security knowledge such as assets and threats to stakeholders' security goals. The pattern-based analysis for automatic security requirements change detection is built upon EMFIncQuery (Bergmann, Ujhelyi, Rath, & Varro, 2011) which is a framework with a language for defining declarative local and global queries over Eclipse Modeling Framework (EMF) models, and a runtime engine for executing

the queries efficiently. The analysis is based on the specification of a security property as a pattern in the EMF-IncQuery language where the pattern defines a set of constraints on requirements model elements. If as a consequence of the introduction of a change(s) in the requirement model the pattern disappears, EMF-IncQuery engine notes the violation of the security property represented by the pattern and suggests corrective actions to solve the security issue. Argumentation analysis complements the automatic pattern-based analysis in that it checks whether there are new security properties to be added or to be removed (Δ Security Properties) as a result of changes in the requirement model. The execution of the steps of the methodology is tool-supported (Bergmann et al., 2011).

Security Architecture Evolution

Software architecture is critical to secure and evolve a system, because it inhibits or enables the system's quality attributes (including modifiability and security) and makes it easier to reason about and manage change (Bass, Clements, & Kazman, 2003). Architectural decisions are crucial for the security of the system, as security flaws at the architectural level are hard or even impossible to fix afterwards without changing the architecture, which can be a very costly operation. Despite the importance of understanding the impact of evolution on security at the architectural level, however, it appears that research in this area is scarce.

The architecture of a system should attempt to ensure the security of that system when it is first deployed. As it will then evolve together with the system, it should be constructed with future evolution paths in mind. Hence, to guarantee the security of the system over time, the architecture of the system should allow future security-related modifications to be performed with a low impact. For example, it may be necessary to upgrade the used security mechanisms, such as cryptographic protocols, or to install a secure link between two different parts of the system.

Software Architecture

Various definitions of software architecture have been proposed by different authors (Rozanski & Woods, 2005; Taylor, Medividovic, & Dashofy, 2010). Software Architecture can be defiend as a collection of elements (processing, data or connecting elements), their relationships, and some degree of rationale (Rozanski & Woods, 2005; Taylor, Medividovic, & Dashofy, 2010).

There are many different ways to interpret an element, which can be captured in the notion of an architectural style. Rozanski and Woods (2005) state the following: "An architectural style expresses a fundamental structural organization schema for software systems. It provides a set of predefined element types, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them". A style thus defines the vocabulary and rules that can be used to define an architecture conforming to that style. A somewhat different definition of software architecture is given by Klusener et al. (2005): "The software architecture of deployed software is determined by those aspects that are the hardest to change". The attribute driven design (ADD) process (Bass et al. 2003) reflects this as well. It is based on the premise that the main drivers for an architecture are the system's quality attributes (non-functional requirements), including modifiability and security. It prescribes to develop the system based on these quality attributes, such that that the most important qualities of the system are certainly fulfilled. Of course, this also implies that the decisions regarding these

qualities are the hardest to change afterwards, and appropriate care must be taken to include them right from the beginning.

Security and Software Architecture

Books on software architecture generally include some general advice related to security. For example, Rozanski and Woods (2005) define a security perspective, which presents security-oriented architectural concerns, activities, tactics, problems and pitfalls, and checklists. Taylor et al. (2010) discuss secure design principles, access control and trust management. Bass et al. (2003) provide general security tactics that can be used. The tactics are categorized as tactics for resisting attacks, detecting attacks, or recovering from an attack. When going beyond the general advice, security at the architectural level can be looked at from different viewpoints. From a *constructive* viewpoint, the question that has to be answered is how to create an architecture that has certain security qualities. Work in this area is mainly concerned with techniques, processes and guidelines to setup a secure software architecture. On the other hand, from the *modeling* viewpoint, security aspects of an architecture have to be expressed and documented, so that they can be stored and communicated. Finally, from the *analysis* viewpoint, it can be investigated what security analyses can be performed based on some architectural model. Note that these viewpoints are not strictly separated; for instance, analysis techniques will often depend on specific model types, and models are usually constructed with the help of constructive guidance. The following paragraphs provide some references to work in each of these three areas.

Constructive viewpoint. We believe that the most important aspect in the constructive viewpoint is the transition from the security requirements to the architectural design. Security requirements can be captured in a variety of ways, as demonstrated in Section Security Requirements Evolution. An example approach to connect these two artifacts is given by Mouratidis and Jürjens (2010), who describe how to integrate Secure Tropos (one of the goal-oriented security requirements engineering methodologies) with UMLSec (see Section *Security Engineering with UML*), in order to assist software engineers with deriving a secure design from security requirements.

Often, security patterns (Yoshioka, Washizaki, & Maruyama, 2008; Hafiz, Adamczyk, & Johnson, 2007) are advocated as a means to construct secure software. Such patterns capture recurrent solutions for specific problems, such that they can easily be reused by others. They can be structured and categorized to assist the architect in picking a suitable pattern (Hafiz et al., 2007; Yskout, Scandariato, & Joosen, 2012b). For instance, in the NFR framework, patterns are used to create secure designs (Weiss, 2007). Van Lamsweerde (2003) also proposes a pattern-based refinement approach for creating architectures from KAOS requirements models, including examples for security as a non-functional goal. More recently, another approach using patterns was proposed by Alebrahim et al. (2011). This approach starts from Problem Frames models, and also takes non-functional requirements (in particular, security and performance) into account. Nhlabatsi et al. (2010) have reviewed and compared multiple approaches for bringing architectural security patterns closer to the requirements engineering space, to reduce the gap between these two areas.

Besides patterns, security design principles are also commonly used as guidance for creating secure software. Taylor et al. (2010) highlight some principles that are particularly well-suited for architectural design, for example the principle of least privilege, complete mediation or defense in depth.

Modeling viewpoint. Despite the importance of architecture for security, little work exists that specifically aims at modeling security at the architectural level. Some existing architectural description languages (ADLs) have been extended to support security. For instance, Secure xADL (Ren & Taylor, 2005) enriches the xADL language with concepts for access control. Security-specific extensions to Data Flow Diagrams (DFDs) have also been proposed (Abi-Antoun, Wang, & Torr, 2007). UML, when considered as an ADL, may also be used or extended to represent security properties. We refer to Section *Security Engineering with UML* for a discussion of the usage of UML for security.

Security analysis viewpoint. Besides providing security-specific notations, UMLsec can also be used to perform a formal analysis on the design. The STRIDE (Hernan, Lambert, Ostwald, & Shostack, 2006) risk analysis method is performed using an architectural description (i.e., a data flow diagram) as input. Measuring the security of a design can be done using security metrics. For instance, an attack surface metric (Manadhata & Wing, 2011) can be used to estimate the security of a design, and subsequently improve it. Additionally, adherence to security principles can be analyzed at the architectural level. For example, analysis techniques exist to detect violations of the least privilege principle (Scandariato, Buyens, & Joosen, 2010). For more background on analysis techniques, we refer to the survey by Dai and Cooper (2007).

Software Architecture Evolution

Evolution of a system's architecture originates from a change in the requirements of the system, or a change in the assumptions and constraints regarding the systems environment. Therefore, as before, it is important to relate evolution at the requirements level to architectural evolution. Also, the impact of a change at the architectural level must be assessed. Of course, changes to the architecture of a system are not isolated and need to be propagated further to other development artifacts (Mens, Magee, & Rumpe, 2010).

A systematic literature review of software architecture evolution has been performed by Breivold et al. (2012), to which we refer for a comprehensive overview of this research area. In this section, rather than aiming at completeness, we give some illustrative and additional references concerning evolution at the architectural level from the constructive, modeling and analysis viewpoint, in an attempt to provide an indication of this topic's extent.

Constructive viewpoint. As before, the reuse of known solutions (for example, in the form of patterns, tactics, or guidelines) is a popular technique used by software architects to achieve a modifiable system, i.e., a system in which the impact of possible changes is reduced. For example, Bass, Clements and Kazman (2003) provide a set of modifiability tactics. These tactics are divided in three groups: The first set of tactics aims to *localize modifications*, for instance by maintaining semantic coherence or anticipating expected changes. The second set *prevents ripple effects*, for instance by hiding information (encapsulation), maintaining existing interfaces, or adding intermediaries. The third set contains tactics to *defer binding time*, for example by making use of configuration files or polymorphism. By applying these tactics at the right places in the architecture, the architect creates a system that can be modified with low architectural impact.

To ease the transition from requirements to architecture, Côté et al. (2007) describe an informal approach based on problem frames,

a requirements engineering methodology. Architectural patterns are assumed to be associated to corresponding problem frames via traceability links. In light of evolution, new sub-frames may emerge in the specification and the architect can extend the design by incorporating additional architectural patterns, whose selection is facilitated by the traceability links.

Change patterns (Yskout, Scandariato, & Joosen, 2012a) precisely capture a generic change at the requirements level and associate this change with a set of architectural solutions. A solution provides a generic template, which needs to be instantiated in the architecture, and a transformation based on this template that evolves the architecture in accordance with the change at the requirements level. These patterns thus assist the architect in both preparing and executing change. They can also be used to propagate changes from the security requirements, via the software architecture, to the running system through runtime reconfiguration (Yskout, Ben David, Scandariato, & Baudry, 2012).

Ahmad et al. (2012) propose a technique for analyzing the history of changes in service based software architectures. The changes are captured in the form of patterns, which are also called change patterns. These patterns can then be queried and reused when change scenarios are executed.

To guide the gradual transition from one architecture to another, Garlan et al. (Garlan, Barnes, Schmerl, & Celiku, 2009b) propose the concept of 'evolution styles'. An evolution style is a pattern that can be used by the architect to plan incremental evolution paths (orthogonal to the evolution of functional requirements) from an initial architecture to some target architecture. For example, the evolution style that is used to illustrate the approach describes the transition from an ad hoc, peer-to-peer architecture to a hub-and-spoke architecture. The style characterizes the intermediate versions of the architecture and a set of operations to assist the architect with evolving from one version to another while making trade-offs. Furthermore, properties and constraints can be added to the path of evolution, in order to perform analyzes, trade-offs or correctness checks.

Modeling viewpoint. The use of graph transformations has been proposed to model architectural evolution. For example, Tamzalit and Mens (2010) describe how they can be used to describe architectural restructuring. In particular, the approach is tailored for defining generic restructurings that introduce an architectural style. The approach starts from defining an Architectural Description Language (ADL) as a type graph, coupled with a set of invariants (constraints). A concrete architectural description is then a graph that conforms to the ADL's type graph. Similarly, architectural styles are modeled as type graphs that extend the type graph of the ADL. A set of graph transformation rules formalizes the process of transforming an architecture to a new one that conforms to a certain architectural style, for example a client-server style. This process (i.e., a sequence of rules) is called an *evolution pattern*. Through *Critical Pair Analysis* (CPA), conflicts and dependencies between the individual rules can be detected automatically.

Barais (2008) studies the support of existing approaches to evolving software architecture based on the separation of concerns principle. It is argued that traditional ADLs do not sufficiently support separation of concerns, for instance for crosscutting concerns such as security. As an alternative, the TranSAT framework is presented, which considers architectural evolution from an aspect-oriented angle. In this framework, evolution is achieved by weaving a new concern (the 'architectural aspect') into an existing architecture (the 'base plan'). The architectural aspect consists of three parts, specified using the

TranSAT transformation language: (1) a *plan*, which captures the structure and behavior that needs to be injected; (2) a *join point mask*, which defines the necessary preconditions on the base plan before the plan can be injected; and (3) a *set of transformation rules*, which specify how the plan needs to be injected. Analysis is supported by static and dynamic checks that can be performed to ensure that the injection results in a valid software architecture.

To support architectural practice, Razavian and Lago (2012) define a viewpoint specifically for representing and reasoning about architectural change. The viewpoint is based on the mental model that the researchers have identified by observing how architects deal with change in a case study. Briefly, architects appear to focus on one source of change at a time, find the elements that are influenced by that source, and then perform the actual change. The proposed viewpoint can be used to model the necessary traceability paths that support this manner of working.

A similar approach is proposed by Szlenk et al. (2012), who present a model and graphical notation for modeling architectural decisions (MAD) that can be used to deal with change at the architectural level. Architectural decisions are modeled using a graph, where an edge denotes a 'leads to'-relationship from one decision to another one. Using this model, changing an earlier decision (due to a changed requirement, for instance) identifies which other decisions may need to be re-examined, and which can be left as is. McVeigh (2009) formally defines a component based architectural description language (Backbone) that supports extensibility, together with a runtime environment and a modeling tool (Evolve). In the Backbone ADL, changes are not made by directly modifying the structure of the system, but rather by explicitly specifying the differences. Backbone is built on three concepts to achieve extensibility, namely resemblance (defining a new component as a set of deltas from an existing one), replacement (globally replacing one component with another), and strata (modules which can group definitions and declare dependencies).

Analysis viewpoint. An important set of techniques for software architecture analysis related to evolution are directed towards conducting a *modifiability analysis*. Such an analysis attempts to assess the impact of a certain evolution scenario on the system's architecture. It has various applications, for example reducing future maintenance effort, comparing multiple alternatives to determine which one will be cheaper in the long term, or performing risk assessments.

Bengtsson et al. (2004) present the architecture-level modifiability analysis (ALMA) process. The method can be used to perform a modifiability analysis of a software architecture with respect to changes in the environment, changes in the requirements and changes in the functional specification The process consists of five steps, namely (1) setting the goal of the analysis; (2) describing the software architecture; (3) elicit change scenarios and classify and select them (using a top-down approach, starting from a classification, or a bottom-up approach, starting from change scenarios); (4) evaluate the effect of the change scenarios; and (5) interpret the results. It has been applied to three case studies, each demonstrating another analysis goal, namely the predicton of maintenance effort, the assessment of risks due to inflexibility, or the comparison of architectural alternatives.

Breivold et al. (2012) proposes another process, AREA, which includes both a qualitative and quantitative method for assessing the evolvability of a software architecture. Furthermore, they define a software evolvability model, which identifies important sub-characteristics of software evolution such as changeability, extensibility and portability that should be

taken into account by practitioners. Their techniques have also been evaluated using two industrial case studies.

On a more fundamental level, Williams and Carver (2010) define the Software Architecture Change Characterization Scheme (SACCS). This scheme is based on an extensive literature review and aims at providing insight into the architectural change process. The impact of a change is assessed using different criteria, such as the motivation, importance, or granularity of the change. In the future, this framework may result in the creation of a change decision support system that can be used by architects to predict the difficulty of a change.

Khan et al. (2008) explicitly acknowledge the relationship of requirements to architectural elements by means of dependencies. In the context of evolution, they first identify six relevant categories of dependencies, i.e., (1) goal dependencies, which relate the quality requirements to architectural elements; (2) service dependencies, which relate functional requirements to operations at the architectural level; (3) conditional dependencies from requirements specifying triggering events and conditions to the architectural elements realizing them; (4) temporal dependencies, relating requirements on the time frame of events or processes to their realization; (5) task dependencies, connecting user input or feedback to the architecture; and (6) infrastructure dependencies, relating requirements on resources, networks, technical details etc. to the architecture. Furthermore, they have assessed the impact on the architecture of changes that belong to these categories.

Secure Software Architecture Evolution

As mentioned at the start of this section, little work exists in the intersection of the secure software engineering and architectural evolution areas. Therefore, this area lends itself well to further research. Section *Security Evolution and UML Models* already provides an overview of security evolution and UML models, which is certainly relevant for the software architecture domain as well. Furthermore, it would be worthwhile to analyze concrete security-related architectural changes, in order to elicit patterns or best practices to securely evolve a system. Hafiz and Johnson (2008) provide a first contribution in this respect, by describing the evolution of Mail Transfer Agent architectures. These systems were confronted with progressively more demanding security requirements due to their rising popularity. The authors show how early architectural decisions impact the security of the system over time and how some architectural choices lead to a more secure system than others. Additionally, it would be valuable to investigate how existing techniques for architectural evolution support security. For example, the change patterns approach mentioned before (Yskout, Scandariato, & Joosen, 2012a) is illustrated with a catalog of patterns for changing trust assumptions forming an important class of security-related changes.

Secure Code Evolution

Source code is the central artifact of any software system. It also reflects many of the properties of other artifacts. The architecture of a software system is reflected in the source code, and a well-designed architecture will make it easy to evolve the source code in specific directions. If security goals have been elicited well during requirements engineering, the source code is more likely to implement appropriate protection measures for all relevant assets. And well-tested code is likely to have fewer bugs and vulnerabilities than untested code. Yet, the coding phase itself introduces its own set of potential problems and pitfalls,

and in this section we focus on these purely implementation-level aspects of security and evolution.

Source Code

Source code is written in a programming language and the choice of programming language potentially has a significant impact on the security and evolvability of code.

A first key property of the programming language that affects security is *safety*. A safe programming language ensures that programs always have a well-defined behavior, i.e., they leave no cases where behavior of a program is *undefined* (Pierce, 2002). In contrast, unsafe languages such as C or C++ may leave the behavior of programs unspecified in some cases, and assume that the programmer takes the necessary measures to avoid such cases. Safe languages are important for secure code evolution since they guarantee the absence of certain classes of security bugs (e.g., buffer overflows).

The most common ways to achieve safety are through *static typing* or through *dynamic typing* (Cardelli, 1997). Dynamically typed languages such as JavaScript or Python perform type checks during the execution of the program and report an error when an operation is executed on the wrong type. Statically typed languages such as Java or ML avoid as many runtime checks as possible by using a static analysis during compilation. At the expense of additional type annotations, statically typed languages guarantee that all referenced methods and operations are defined for the given argument types. Showing that the type system indeed guarantees safety is done by means of a *type soundness* proof (Pierce, 2002). Dynamically typed languages often have rich reflective capacities that allow adding new code or transforming existing code at runtime. While this is beneficial for runtime code evolution, it also allows breaking the invariants of other modules which is detrimental for secure code evolution.

A second key property of a programming language that affects secure evolution is its support for *modules* and *modularity*. Roughly speaking a module is a blob of code that has an *interface* and an *implementation* (Sullivan, Griswold, Cai, & Hallen, 2001). Code is modular, if the correctness of a module only depends on the interfaces of the modules it depends on.

Modules can exist at widely varying levels of granularity. In mainstream object-oriented languages, it is common to decompose a program into packages consisting of classes, which again sprout methods. Packages, classes and methods can all be considered modules. Aspect-oriented programming provides even more advanced modularization mechanisms. Aspects support the implementation of crosscutting concerns in separate modules, which are then weaved together with the base program. This has an important positive impact on evolvability and security (De Win, Joosen, & Piessens, 2004).

Once the programming language is fixed, there are still an infinite number of ways to implement the same functionality, and some of these implementations will be better (from the point of view of security and/or from the point of view of evolution) than others. The most common approach to steer programmers to “good” implementations is the use of coding principles and guidelines (M. Howard & Leblanc, 2001).

One of the important guidelines for the design of software is the concept of *information hiding* (Parnas, 1972), which means that the internal implementation of a module should be hidden to the clients of that module. As a result, the implementation of that module can evolve without requiring changes to the clients. Furthermore, *low coupling* and *high cohesion* (Stevens, Myers, & Constantine, 1974) are two important principles to evaluate the design of systems. On the one hand, modules should have as

little dependencies on each other as possible. On the other hand, the responsibilities of each module should be logically coherent.

Secure Coding

Vulnerabilities that are introduced during the implementation phase are essentially coding bugs with security consequences (exploitable coding bugs). Well-known examples of such implementation-level vulnerabilities include buffer overflows (Erlingsson, Younan, & Piessens, 2010), and command injection attacks such as SQL injection (Halfond, Viegas, & Orso, 2006) or cross-site scripting (XSS) (Johns, 2008). Because a wide range of such bugs exists, a significant amount of research has been conducted regarding their classification. Three important scopes that have been studied intensively are:

1. Vulnerabilities in infrastructural software such as operating systems, web servers or application servers. If an attacker can penetrate the runtime infrastructure on which an application is running, he can also easily penetrate the application itself. Since vulnerabilities in infrastructural software were the main cause of security incidents in the seventies, eighties and early nineties, many of the early classifications and taxonomies focused on operating system vulnerabilities (Abbott et al., 1976). In the early nineties, Landwehr et al. (1994) published a catalog of 50 actual flaws, and proposed a taxonomy for them.
2. Vulnerabilities in security components. If a security component is implemented incorrectly, the protection it provides can be bypassed by attackers. Well-known papers analyzing vulnerabilities in cryptographic components (both primitives and protocols) are the papers by Anderson (1993) or Ferguson and Schneier (2003).
3. Vulnerabilities in applications. As the importance of application-level vulnerabilities increases, the systematic study of these vulnerabilities has attracted more attention. Both academia (Wang & Wang, 2003), as well as industry (MITRE, n.d.; The Open Web Application Security Project, 2011) have investigated suitable taxonomies for application level security flaws. While academic work is typically more analytical, and looks for sound scientific grounds for taxonomies, the work from industry is more pragmatic, and its primary aim is to come up with useful taxonomies to help building and comparing tools that scan for vulnerabilities. Some classifications, such as the OWASP Top 10, even have as primary goal to raise awareness among developers. The most systematic approach appears to be CWE, the Common Weaknesses Enumeration (MITRE, n.d.), and a formal list of types of software weaknesses. It is intended to unify the jargon on vulnerabilities, and to provide a way to measure effectiveness of vulnerability scanning tools.

Countermeasures against code-level vulnerabilities can also take many forms. They range from improvements in programming language and type system design, over static verification techniques, coding guidelines or runtime monitoring of the code.

As discussed above, *safety* is an important security-related property of a programming language. Dealing with the security consequences of non-safety of programming languages, and more specifically dealing with the non-safety of C and C++ is a very important research area in itself, and good surveys of countermeasures exist (Younan, Joosen, & Piessens, 2012). Hence, we do not discuss these further in this article.

Language-based security (F. Schneider,

Morrisett, & Harper, 2001) for safe programming languages has become an important area of research over the past years. Broadly speaking, language-based security uses tools and techniques from the programming languages research community to address security-related issues. Many important results have been achieved. Schneider, Morrisett, Hamlen and others (F. Schneider et al., 2001; Hamlen, Morrisett, & Schneider, 2006) have provided a broad and widely accepted definition of security policies for programs, and have investigated what classes of policies can be enforced with what kind of enforcement technologies.

For access control policies, the most widely studied enforcement technology is execution monitoring. Several projects, e.g., (Erlingsson & Schneider, 2000), have built execution monitors that can enforce configurable and expressive policies. More powerful than execution monitoring are approaches that can change the flow of events generated by a program. Edit automata (Ligatti, Bauer, & Walker, 2005) enhance security automata with the ability to replace, delete or insert program actions. The Polymer system (Bauer, Ligatti, & Walker, 2005) is a Java-based implementation of edit automata.

Precise enforcement of certain security policies (for instance, information flow policies) is impossible using only execution monitoring, as these policies describe properties of sets of executions instead of single executions. For such policies, static verification is an important enforcement technology. Static verification usually relies on program annotations, for instance, typing information, or specification annotations. Checking of information flow policies through static verification is a very rich and active research field. A survey of results is given by Myers and Sabelfeld (2003).

Very recently, more intricate dynamic techniques such as secure multi-execution (Devriese & Piessens, 2010) and faceted evaluation (Austin & Flanagan, 2012) have been developed that support the dynamic enforcement of information flow policies, albeit at a non-negligible performance cost (De Groef, Devriese, Nikiforakis, & Piessens, 2012).

An interesting line of research has shown that certain classes of vulnerabilities can be addressed by imposing *a programming model* on developers. A programming model can be thought of as a formally specified set of coding guidelines that can be checked statically or dynamically and that provides specific formal security guarantees. Example programming models deal with concurrency related vulnerabilities (e.g., race conditions) (Jacobs, Leino, Piessens, & Schulte, 2005), with code access security related bugs (Smans, Jacobs, & Piessens, 2006) and with vulnerabilities related to forceful browsing (Desmet, Piessens, Joosen, & Verbaeten, 2006). Pluggable type systems (Andreae, Noble, Markstrum, & Millstein, 2006) operate in a similar way.

On the more pragmatic side, a variety of tools exists, that uses heuristics-based static analysis to detect security vulnerabilities. Well-known tools include FlawFinder (Wheeler, n.d.), FindBugs (FindBugs, n.d.), and so forth. They usually detect suspicious syntactical patterns in the program being analyzed. Some of these tools are user-extensible: new rules to detect vulnerabilities can be programmed into the tool.

Code Evolution

The key property to support evolution of source code is modularity. Modularity supports evolution in different ways. First, since other modules depend only on the interface of a given module, implementation-details of that module can be changed without impacting these other modules. This supports local updating and bug-fixing of code while making sure that the impact of such a change remains confined to a single module. Second, at a more coarse-granular scale, the modular

construction of an application supports the relatively flexible addition, removal, modification or re-composition of modules in the style of component-based development (Szyperski, 1998).

Of course, an important concern is whether modular reasoning about source code properties is sound in general. How can one be sure that the correctness of a particular module indeed only depends on the interfaces of other modules? If certain correctness properties (including also security properties) depend on implementation details of other modules, then a local change in one module can break properties of another module. Modular reasoning about imperative code turns out to be challenging, as all modules have side-effects to a single program heap.

An important breakthrough in the past decade has been the development of separation-logic (Reynolds, 2002) and other related program logics (Kassios, 2006; Smans, Jacobs, & Piessens, 2009) for reasoning about imperative code. Several mature tools exist (Jacobs, Smans, & Piessens, 2010; Cohen et al., 2009) that can verify correctness properties of non-trivial code bases (Penninckx, Mühlberg, Smans, Jacobs, & Piessens, 2012) in a modular way, albeit at the cost of a substantial additional effort from the developer. These program logics force developers to make all assumptions on which correctness properties depend explicit in the form of program annotations. If one of these assumptions is violated during code evolution, this will be detected by the verifier.

Another aspect of code evolution that is of particular relevance to this survey article is the research on advanced modularization concepts that aim to support the modular implementation of non-functional cross-cutting concerns (since security is a prime example of such a concern). The most common umbrella term for such modularization concepts is aspects, and languages that support aspects are referred to as aspect-oriented languages. There are several mature research prototypes of such general purpose *aspect-oriented programming languages*, the most prominent one being the AspectJ language (Kiczales et al., 2001). On the more practical side, mainstream component frameworks such as Spring, JBoss and Microsoft .Net provide general purpose aspect oriented extensions.

Secure Code Evolution

Code evolution mainly interacts with vulnerabilities in security components and vulnerabilities in applications. Vulnerabilities in security components can often be attributed to the fact that security functionality is extremely hard to modularize: code dealing with access control for instance, is typically spread throughout the entire code base. As an example, in the Sun JDK 1.3 implementation, a reference to the SecurityManager is created at 126 different places in the code base, and calls to the access decision function occur at 170 different places (Win, 2004). This crosscutting nature of security concerns makes implementing them error-prone, and maintaining them very hard. It is exactly for these kinds of vulnerabilities that aspect-oriented techniques as discussed above can provide solutions. By offering new modularization constructs that are better at modularizing crosscutting concerns such as access control. Aspect-oriented programming holds great promise for reducing the number of vulnerabilities of this type (De Win, Piessens, Joosen, & Verhanneman, 2002). In particular, by supporting modular implementations of security functionality such as access control or secure communication, these approaches support a flexible evolution of that security functionality. For example, it is possible to implement the stack inspection mechanism in Java using Inline Reference Monitors (IRMs) (Erlingsson, 2004) while

maintaining competitive performance (Erlingsson & Schneider, 2000). Such an approach has the advantage that the security policy can be modified without modifying the existing JDK implementations.

To make sure that no vulnerabilities in the application logic itself are introduced during evolution, the reasoning about their absence should be made modular. Two important approaches succeed in modularizing the analysis for security vulnerabilities: type systems (Pierce, 2002) and modular full functional verification (Jacobs et al., 2011). An important idea underlying both these approaches is the fact that the developer of the code is supposed to provide additional information (in the form of type annotations, assertions, module contracts and so forth) that explain why the code is secure. The type checker or *verifier* then checks these annotations. If this checking is modular, only the parts of the code that are being changed need to be rechecked.

For the *typing-based* approach, the first security-specific type systems focused on enforcing information flow properties (Volpano, Irvine, & Smith, 1996), and the development of such information flow security type systems has seen a significant amount of activity since then, leading up to two fairly mature programming languages that support them, FlowCaml (Pottier & Simonet, 2003) and JFlow/JIF (Myers, 1999). But type systems have also been used to guarantee other security properties such as access control (Walker, 2000).

The *verification-based* approach to secure code evolution builds on the general purpose verification tools mentioned in Section Secure Coding. Both information-flow style security properties (Barthe, D'argenio, & Rezk, 2011) as well as access control style properties (Smans et al., 2006) can be casted as assertions in a program logic. By specifying such assertions, developers can support the evolution of code with high assurance that such evolution will not break intended security properties of the code.

Security Test Evolution

Testing is the evaluation of software by observing its execution. *Model-based testing* (MBT) relies on models to generate tests, to execute tests or to evaluate their results. Model-based testing offers big potential for automation and adaptation in the testing process. It is therefore well-suited (1) for testing dynamically evolving systems where test suites have to be executed several times which is not possible if the test execution is not automated, (2) where tests have to be modified which is much easier on the abstract level of models than on the code level, and (3) where tests even have to be regenerated which is supported by the automated derivation of test cases in many model-based testing approaches. *Regression testing* is a selective retesting to verify that modifications have not caused unintended effects and that the system under test still complies with the specified requirements (IEEE, 1990). Regression testing is essential to verify evolving systems when they are changed and also takes the test evolution management into account. After an overview of security testing, we then discuss regression testing including test evolution management in Section *Regression Testing*, and finally consider their combination in Section *Security Test Evolution* on security testing and evolution management.

Security Testing

Security testing is software testing of security requirements like *confidentiality, integrity, authentication, authorization, availability*, and *non-repudiation*. Security testing can be divided into *security functional testing and security vulnerability testing* (Tian-yang, Yin-sheng, & You-yuan, 2010). Security functional testing ensures whether security functions (positive security requirements) are implemented correctly and consistently with respect to security

requirements. Security vulnerability testing addresses the identification and discovery of yet unknown vulnerabilities (negative security requirements) that are introduced by security design flaws or software defects. Security vulnerability testing lacks approaches for systematic design, execution and evaluation of tests and uses the simulation of attacks like performed by hackers which is called *penetration testing*.

Model-based testing techniques provide additional support to lower the required level of expertise needed for security testing (Felderer, Agreiter, Zech, & Breu, 2011). By using models, the level of abstraction is raised which enables more people to design tests, and the model can be employed to automatically generate test cases. Additionally, security models are often created in conjunction with a risk analysis. This risk information can, on the one hand, be used for deriving test cases, and, on the other hand, for prioritizing test execution. Felderer et al. (2011) classify model-based testing approaches along the two dimensions risk integration and automated test generation. Based on the different perspectives used in securing a system, Schieferdecker et al. (Schieferdecker, Grossmann, & Schneider, 2012) distinguish several types of input models for test generation, i.e., (1) architectural and functional models, (2) threat, fault and risk models, and (3) weakness and vulnerability models. A powerful test approach which is especially useful for security testing is mutation (Weiglhofer, Aichernig, & Wotawa, 2009). Typically, mutation testing is used to qualify test suites by running tests against a mutation of the system under test. The quality of the test suite is stated with respect to the number of mutants being detected by the test suite. For security testing, models of the system under test are mutated in a way that the mutants represent known vulnerabilities (Schieferdecker et al., 2012). These vulnerability models can then be used for test generation by various MBT approaches. The generated tests are used to check whether the system under test is vulnerable with respect to the vulnerabilities in the model.

Regression Testing

Regression testing is considered very expensive but an essential activity in the software maintenance process. Hence, evolving systems must be validated before their redeployment on the market and this is even more crucial for critical evolving systems, on which safety and security depends. Thus, the process of regression testing takes into account testing the code before delivery of the product. It aims to ensure that changes at code level, which correspond to changes at requirements level described in the specification, do not impact the non-modified code. It can be performed on several testing levels, i.e., unit, integration, and system level.

According to the accessibility of test design artifacts, regression testing can be *code-based*, also called white box regression testing or *model-based*. Indeed, these two types are considered as complementary and both of them are effective in revealing regression faults. Rothermel et al. (2001) clarify the regression testing strategies and separate them in the two categories *retest-All* techniques and *selective* techniques. Retest-All techniques require execution of the entire test suite, provided from the previous version, on the new system's version. Selective techniques require selection of a subset of the test suite, which may reveal errors in the new system, using different approaches. Thus, selective regression testing techniques can be classified into the categories minimization, coverage-based, safe as well as ad-hoc random. The goal of minimization techniques is to choose the minimal number of tests from the original test suite, those crossing through the modified instructions or affected by the program modification. The goal of coverage-

based approaches is to take into account the path and data dependence graph coverage by tests. Safe techniques ensure that all tests selected from the initial test suite may reveal faults in the system. Most of the minimization and flow-based techniques are said not to be safe. Ad-hoc techniques consider that all tests from the initial suite may reveal faults and selects randomly a certain number of tests. As mentioned before, all these techniques are based either on code, or on models. Latter have several benefits. Model-based techniques (1) permit the traceability between the specification and the testing activity, (2) from scalability point of view, these techniques easy the work with very complex and industry scale systems, and finally (3) they are independent from programming languages.

Security Test Evolution

During the last years research shows increasing work in the domain of validation and verification of security requirements, in particular for critical systems. A great attention is given to testing different variants of positive and negative security requirements like security properties, access control policies or vulnerabilities (see Section *Security Testing*). It has been achieved to create stable and certified critical systems and, in case of evolution and introducing new requirements, the main goal is to preserve these security requirements. Practitioners are aware that the security requirements must be preserved and verified for the new and evolved system.

However, to the best of our knowledge, very few research is done in the field of regression testing for security requirements, called security regression testing (SRT) although studies on the system development lifecycle (Mehta, 2007) underline the importance of regression testing while verifying the vulnerabilities of a system. Authors in (Alnatheer, Gravell, & Argles, 2010; Kongsli, 2006), remind the need of regression testing for security requirements in agile development. Kongsli (2006) suggests applying *misuse stories*, contrary to *use case stories*, and considers them as possibility to take into account requirements link to security issues. Moreover, he proposes to use tests dedicated to *misuse stories* in order to ensure regression testing.

Furthermore, authors in (SecureChange, 2012) present a model-based regression testing approach for system security. The security requirements are captured through schemas (or scenarios), written in the Smartesting Schema Language, which are further used to drive the test generation. The evolving aspects of the systems are captured by adapting the SeTGaM technique (Fourneret, Bouquet, Dadeau & Debricon, 2011) to security requirements. They manage the test's life cycle for the new system version through finer grained status and update. Further, their approach selects a precise set of security requirements from the new version not covered by current tests and generates tests only for this set, thereby avoiding the unnecessary full regeneration of tests for the new version.

Felderer et al. (2011) consider model-based security regression testing of service-centric systems. They attach state machines to all model elements of the requirements, system, and test model to obtain consistent and traceable evolution. Adding, modifying or deleting model elements trigger change events and fire transitions in the state machines. Tests have an additional type following the classification of Leung and White (1989) which can be evolution, for testing novelties of the system, *regression*, for testing non-modified parts and ensuring that evolution did not unintentionally take place on other parts, *stagnation*, for ensuring that evolution did actually take place and changed the behavior of the system, and *obsolete*, for tests which are not relevant any more. Based on a test requirement expressed as an OCL query and considering the actual state of model elements as well as the type

of tests, a regression test suite is selected and executed.

In addition, Hwang et al. (2012) suggest three regression test selection techniques for access control policies specified in XACML. The techniques are based on: (1) mutation, (2) coverage, and (3) recorded request evaluation of access-control policies. Each policy P is composed of rules r_i. The first technique, first makes a correlation between rules and tests, then selects the rules r_i from P and creates mutants of the policy by changing the rules decision, denoted $M(r_i)$. This technique selects tests revealing different behaviors of the policy when executing tests on the program in interaction with the policy P and its mutants $M(r_i)$. It is very costly, since it executes tests $2 \cdot n$ times, where n is the number of rules in the policy. The second technique monitors which rules are evaluated for requests issued from the execution of test case on the program in interaction with the policy. Then it establishes a correlation, as the previous technique, between rules and tests. Finally, the last technique records requests issued from security checks, called Policy Enforcement Points (PEPs) when executing tests on the program. The tests which encapsulate different decisions for the given policy and the modified one are selected for regression testing.

Hence, much research work is done in the field of regression testing of functionalities in a system, but Yoo and Harman (2010) point out the need to continue the investigation in regression testing non-functional requirements like security.

Security Monitoring

In this section, we discuss a specific branch of monitoring called *security monitoring*. We do not discuss any particular flavor of security monitoring in abundant detail, but focus however on providing a coarse, yet broad, discussion of ongoing research and industrial efforts in various areas of security monitoring.

Monitoring can be seen as the ongoing process of evaluating artifacts, i.e., computers, processes, whole infrastructures or even people based on certain criteria, e.g., specified through policies. Security monitoring itself is more specific. On an abstract level, its major goal is to detect security violations during operation. Such violations can be classified as technical, e.g., data leakage, or business violations, e.g., compliance violation of legal requirements such as HIPAA (Health Insurance Portability and Accountability Act). The tasks for security monitoring can be further refined. Attack detection, for instance, is used to detect insider and/or outsider ongoing attacks on a technical more abstract level. The tools employed for this kind of monitoring depend on the focus point. The technical point of view is provided by Intrusion Detection Systems (IDSs) (Denning, 1987; Lunt, 1993) with insider attack detection capabilities as proposed by Schultz et al. (Schultz, 2002). Whereas, an insider attack focused solely on business activity, would be considered by monitoring in the area of fraud detection. A substantial survey of work on fraud detection is provided by Phua et al. (2005).

The following description of a security monitoring schematic is the standard architecture for IDSs, yet it is general enough to apply to any monitoring architecture discussed here. The *Common Intrusion Detection Framework* (CIDF) managed by the *Intrusion Detection Working Group* (IDWG) defined a common architecture for IDS (Garcia-Teodoro, Diaz-Verdejo, MaciaFernandez, & Vazquez, 2009) by using four functional components which are as follows:

- E blocks, also called Event-boxes, are basically configured sensors that monitor given target systems.

- D blocks, also called Database-boxes, store event data received by the E blocks and allow further processing.
- A blocks, also called Analysis-boxes, are components that allow further analysis on data within D blocks.
- R blocks, also called Response-boxes, allow reacting on alerts, e.g., stop execution of a compromised system or revoke access for a user.

Different locations (e.g., layers in the ISO/OSI model) for event-boxes imply a variety of different security monitoring branches. E blocks may collect network traffic, database activity, host activity, user activity, or Web Service activity. Such a general architecture allows, therefore, to split IDS further into *Host-based/Network-based IDS* ((H)NIDS). Other types include, for instance, Web Services Monitoring if the focus is on Web Services. If gathered data is mainly business-oriented, such as credit card transactions, trade activity, product sale statistics, it is called *Business Activity Monitoring*.

Altering A blocks, does not change the branch of monitoring, but rather the reasoning/evaluation technique that is used. Standard A blocks usually consist of either signatures, statistical, or machine learning algorithms. Signatures are provided by experts to perform a rule-based analysis of an incident. In short, signatures are basically a look-up table to discern if features, extracted from event data in D blocks, are malicious. Statistical algorithms usually revolve around statistical outlier tests, i.e., testing deviations in given distributions via the Grubbs test, or hypothesis tests to test if samples correspond to certain distributions, among others the $\chi 2$ or the Kolmogorow-Smirnow test (Denning, 1987; Kruegel & Vigna, 2003; Garcia-Teodoro et al., 2009; Chandola, Banerjee, & Kumar, 2009). Machine learning-based algorithms usually orbit around, e.g., SVMs, Clustering, Bayesian, or Markovian methods to classify given data instances as attacks or anomalies (Garcia-Teodoro et al., 2009; Chandola et al., 2009; K. Leung & Leckie, 2005; Gu, Perdisci, Zhang, & Lee, 2008).

The most common analysis types in security monitoring are *signature-based* and *anomaly-based* approaches. And among those two, signature-based methods seem to be used more often in the industry (Sourcefire, n.d.; Alienvault, n.d.; Trend Micro, n.d.). The main reason why signature-based approaches are popular among industrial areas are basically the ease of deployment, the ease of adapting the monitoring system to new threats (assuming there are known and significant patterns), a low number of false-positives, and intelligible reports. On the other side, algorithms for anomaly detection (either statistical or machine learning-based) are, due to their intricacy, generally hard to maintain. They produce a high number of false-positives if used incorrectly, and conveying results over reports tends to be hard. Yet, anomaly detection approaches have shown their value in identifying attacks, never identifiable with signature-based methods, for instance, user masquerading attacks (Schonlau et al., 2001). The IDWG schematic above makes it clear that by changing the focus of the E boxes and the type of analysis of the A boxes it is possible to build various variants for security monitoring systems, including hybrid ones. A signature-based IDS can, for instance easily be combined with anomaly detection elements (e.g., SNORT by using the network anomaly plugin SPADE, or applying HTTP traffic extraction to perform a semantic anomaly-based intrusion detection (EstévezTapiador, Garcıa-Teodoro, & Dıaz-Verdejo, 2004)).

A specialized signature-based monitoring technique is known as *Complex Event Processing* (CEP) (Luckham, 2008). CEP helps security experts to correlate the large quantity of different events from various sources. CEP itself is an umbrella term for

methods processing events, in real-time through sensors, query languages, event databases, and internal query representations. In short, multiple minor events are matched with queries in the database and imply a complex, more severe or meaningful, event.

Correlation

A huge problem for security operators are the number of false-positives and meaningless alerts from the network. The reduction of false-positives and increase for the confidence of alerts, led to correlation and aggregation algorithms (Julisch, 2003; Ning, Cui, & Reeves, 2002), the creation and handling of attack models, i.e., attack graphs (Phillips & Swiler, 1998; Noel & Jajodia, 2004), automated model creation tools (Ou, Govindavajhala, & Appel, 2005), and even model-based correlation (Roschke, Cheng, & Meinel, 2011).

Julisch et al. (2003) leverage clustering to perform aggregation of alerts via generalization hierarchies, distance metrics, and derived dendrograms and achieve a massive reduction of alerts. In Ning et al. (2002), attack scenarios are mined by matching alert information, e.g., formal prerequisites and consequences assigned to them, and thus aggregating them. Attack graphs were introduced by (Phillips & Swiler, 1998) to provide a graph-based vulnerability analysis to assess consequences for assets and, hence, also to provide means to assess their risk. In their paper they also show that analyzing an attack graph can be computationally hard (i.e., finding the longest paths in such an attack graph is NPcomplete). Noel and Jajodia (2004) describe a system to manage network attack graphs that renders large attack graphs feasible for human interaction, for instance by allowing to aggregate subsets of the attack graph in a hierarchical manner. Recently, attack graphs have also been used for correlation purposes. In Roschke et al. (2011), alerts are linked to nodes in the attack tree. Then, a dependency relation is forced upon the alerts to attain a dependency graph. Finally, the aggregated alerts are processed to identify the most suspicious subsets of the set.

In the following sections, we discuss various branches of security monitoring that sometimes rely on signature-based approaches (including CEP), statistical modeling, and/or machine learning. It is important to distinguish these different classes of monitoring systems, because of their different properties. These groups are subsets of *Business Activity Monitoring* (BAM), i.e., fraud closely tied to the business layer, *Web Services Monitoring* used to measure the execution of workflows, and *Security Infrastructure and Event Management* (SIEM) providing a holistic aggregation of security information. Finally, we discuss the importance of evolution in the area of security monitoring.

Business Activity Monitoring

BAM is a variant of process mining considering business relevant services and providing a high-level view on workflows, transactions, quality of service, but also compliance to service-level-agreements. Typically, BAM is not considered to be a part of security monitoring, with the exception if it is done with the purpose of fraud detection, which goes hand in hand with process mining (van Dongen, De Medeiros, Verbeek, Weijters, & van der Aalst, 2005). In contrast to IDS, the emphasis is on detecting fraudulent behavior that causes financial loss, for instance, due to insiders. Examples include credit card fraud, telecommunication fraud, or fraud in the health-care system (Bolton & Hand, 2002). Doctors, for instance, may prescribe more expensive alternative drugs than cheaper generic drugs. The usual procedure is to monitor databases, traffic, and messages, to

either semantically analyze the content or check for statistical obliquities.

Giblin et al. (2005) take regulations and transform them via the use of specialized temporal patterns to a timed propositional temporal logic which describes a hand-crafted domain specific model of the regulation to monitor. Afterwards, by using transformation rules, a monitoring architecture is configured.

Mulo et al. (2009) propose monitoring compliance of business processes in SOA via CEP means. A service invocation is regarded as an event and business process activities as event-trails. These event trails guide the creation of queries which a CEP engine uses to identify and monitor business activities. Since the business activities are rendered identifiable it is possible to monitor the flow of a business process at runtime, hence, it is possible to detect anomalous process executions. In BAM it is common to summarize desired properties in key performance indicators (KPI), e.g., the average process duration.

Wetzstein et al. (2009) use that technique to do performance monitoring for the analysis of WS-BPEL processes, combining process events and QoS measurements. They propose a framework which uses machine learning techniques to construct tree structures (binary decision tree generation), which represent the dependencies of a KPI on process and QoS metrics. Analysts then study these dependency trees to determine the impact of lower-level process metrics and QoS characteristics on the process KPIs.

To tackle regulatory compliance (for example SOX) Holmes et al. (2010) propose a model-aware repository and service environment (MORSE). In a nutshell, it allows the generation of services, business process code and monitoring directives via model-driven development (MDD) techniques. A so created business process has compliance models applied to it denoting a certain regulation or implementation details. Afterwards, this annotated process is automatically applied on a business process engine. During workflow execution, the annotations will be retrieved and evaluated by the monitoring infrastructure to detect violations. Traceability of generated events, code as well as model artifacts is managed via UUIDs which are attached to events and artifacts. The business expert can then improve the process via updating the model in the repository.

Web Services Monitoring

Although Web Services may be used in BAM or IDS, standalone security monitoring solutions explicitly for service-based systems relying on Web Services technologies exist (Baresi, Guinea, & Plebani, 2006a; Erradi, Maheshwari, & Tosic, 2007). To our understanding the key difference between BAM and the monitoring of Web Services is that the latter mostly focuses of events extracted from Web Services concerning Web Services, e.g., monitoring of dynamic compositions of Web Services for SLAs.

Baresi et al. (Baresi, Guinea, & Plebani, 2006b) and Erradi et al. (2007) focus on monitoring the execution of centrally orchestrated Web Services compositions (specified in WS-BPEL) in order to detect, correlate and react meaningfully to incidents. Baresi et al. (Baresi et al., 2006b) extends WS-Policy with a language for constraints to monitor functional and non-functional requirements (weaved with the BPEL process at deployment-time). This approach focuses on monitoring very low-level security requirements such as signature algorithms used.

Erradi et al. (2007) present a hybrid approach for functional and QoS monitoring combining synchronous and asynchronous monitoring techniques. The authors extend WS-Policy to WS-Policy4MASC, and present a middleware to provide Web Services compositions with policy-enabled

monitoring capabilities. In Leitner et al. (2010) a framework called PREvent is introduced (based upon event-based monitoring of composed services) which enables prediction of SLA violations using machine learning, but also runtime prevention of those violations via triggered adaptation actions. The novelty of this idea lies in the way the automated composition adaptation is done. The framework does not wait for violations to take place but rather predicts future violation and guarantees successful execution by preemptively altering the services compositions.

Intrusion Detection

IDS differ from previous monitoring technologies mostly by its technical layer and its singular view on intrusions. Topics applying to IDS are detecting incidents, such as, malware propagating in the network, infiltration of the network from outside, and violation of security policies from internal users. Already during the Internet's infancy work in IDS technology was made, which led to an excellent summary of IDS capabilities in (Denning, 1987) and a follow-up taxonomy based on relevant IDS solutions of the time (Axelsson, 2000). As mentioned above, IDS come in various shapes, e.g., host-based, network-based, Garcia et al. (2009) even go so far as to coin the term *anomaly-based* NIDS (A-IDS) to denote specific NIDS that are tailored to detect anomalies. Yet they all share the goal to help a security expert to detect if an attack or intrusion is taking (or took) place. In recent years, A-IDS have seen some progress and are included in more and more enterprise-level commercial intrusion detection systems, for a list consider (Garcia-Teodoro et al., 2009).

Basic techniques in anomaly-detection include statistical, information-theoretic and machine learning-based approaches. Statistical approaches use univariate and multivariate modeling, e.g., for statistical correlation analysis (Ye, Emran, Chen, & Vilbert, 2002). Seminal work in information theory and intrusion detection (Lee & Xiang, 2001) has shown that anomaly detection for audit data (ranging from system calls to network data) can benefit from information theoretic measurements such as, e.g., *entropy, conditional entropy,* and *information gain*. Entropy itself can be used to detect aberrations of regularity in sequences of records, conditional entropy can be used to determine similarity of datasets (Lee & Xiang, 2001), whereas information gain is known to describe a classifier's ability to classify data. Information gain is used explicitly in decision tree learning algorithms like ID3 (Quinlan, 1986) and C4.5 (Quinlan, 1996). Techniques for machine learning involve both unsupervised, i.e., clustering and supervised approaches. Clustering is a quite versatile tool as several approaches (Oldmeadow, Ravinutala, & Leckie, 2004; K. Leung & Leckie, 2005; Gu et al., 2008) show. Gu et al. (2008) use clustering for the detection of botnets by a framework called "Botminer". Leung and Leckie (2005) improve clustering for NIDS by using a density-based clustering algorithm and a grid-based metric and evaluate their efforts on the KDD 1999 data set. Supervised approaches involve, for instance, Support Vector Machines (Mukkamala, Janoski, & Sung, 2002), Neural Networks (Debar, Becker, & Siboni, 1992), Bayesian Networks (Kruegel, Mutz, Robertson, & Valeur, 2003), and fuzzy data mining (Jin, Sun, Chen, & Han, 2004).

Security Monitoring Evolution

In this section, we discuss trends and developments of security monitoring in research and industry. Security monitoring is arguably a very mature field, so during the last decade many problems have been solved. For instance identifying potential event sources and detection mechanisms

(Denning, 1987), various ways to perform anomaly detection (Garcia-Teodoro et al., 2009), scalable and distributed architectures for monitoring (Balasubramaniyan, Garcia-Fernandez, Isacoff, Spafford & Zamboni, 1998 Tierney et al., 2002) and formats to let security monitoring systems communicate with each other, i.e., the *Intrusion Message Exchange Format* (ID -MEF) (Debar, Curry, & Feinstein, 2007). The main methods of detection based signatures or statistics (including anomaly detection), defeat known malware, if signatures and distributions are known. But attackers, their attacks and the patterns thereof *evolve*. Therefore security monitoring systems have to evolve as well. *Advanced Persistent Threats* (APTs) are an example for this type of evolution. The difference between normal attacks and APTs is best explained by elaborating on the keywords.

Advanced denotes a series of well-coordinated attacks by an organization with massive financial means and a high degree of expertise. Culprits of such attacks are therefore often criminal organizations or governments. The tools in use are usually tailored to the scenario.

Persistent denotes that attacks are usually part of a larger process that involves scouting, intrusion, impersonation, and knowledge acquisition. Since the attack is spread over a larger amount of time, e.g., months, suspicions are low.

Issues related to ongoing espionage, destruction of industrial units, or new generation tools (e.g., Stuxnet, Duqu, Sykipot) have always been associated with the term APT (Gao, Morris, Reaves, & Richey, 2010; Bencsáth, Pék, Buttyán, & Félegyházi, 2012; Sood & Enbody, 2013). APTs are currently an open research problem since they demand comprehensive monitoring, correlation and context information by threat intelligence, as well as means to determine changes in behavior, e.g., via anomaly detection (Binde, McRee, & O'Connor, 2011). To address the problem of APTs, industrial as well as scientific security monitoring grow together to build systems that incorporate current best practices, such as signature-based monitoring, rule-based correlation, as well as other features, such as context information, multi-layer anomaly detection, and collaborative intrusion detection (CID). An industry example of context information and CID is provided by AlienVault's Open Threat Exchange (OTE). OTE allows users of OSSIM (Alienvault, n.d.) to share threat intelligence, e.g., bad IP address ranges. Recently, CID is also employed in the area of cloud security monitoring (Sood & Enbody, 2013; Zargar, Takabi, & Joshi, 2011). Context information is often provided by models, e.g., Gander et al. (2011) provide a metamodel that allows modeling of an IT landscape to link infrastructure artifacts, such as workflows, executing services, hosts, and users, to each other in order to provide a better context for anomaly detection and detect complex attack patterns.

Anomaly detection itself undergoes improvements. Horng et al. (2011) show, that they were capable to improve the detection performance of IDS on the KDD dataset by combining SVMs and hierarchical clustering in a single algorithm.

Preliminary work of Gander et al. (2012) tries to combine CEP for workflow monitoring and anomaly detection to create profiles of database and network usage. By linking profile information to service events access control violations are detected.

Industrial detection systems, e.g., OSSIM and Prelude, allow to aggregate information from many sources, but are not explicitly designed to handle the amount of data that supports wider statistical analysis. Advances in database research and log management, especially in tool support, could make this easier. Logstash and Graylog2, for instance, are two open source log management frameworks allowing efficient storage of large amounts of data in a distributed

manner. They incorporate state of the art large database querying support (e.g., leveraging ElasticSearch) and allow traditional and non-traditional databases (e.g., SQL, MongoDB).

To sum up, detection systems leveraging agile, pluggable, frameworks may handle large amounts of data much more easily and perform anomaly detection in all kinds of layers. For instance the application layer, detecting application anomalies, service invocation anomalies, or in the network layer, detecting network anomalies (e.g., TCP, UDP). The achieved combination of aforementioned best practices, such as rule-based, or model-based, correlation of events and statistical methods crunching through large datasets will provide better detection rates and reduce false-positives.

Security Risk Evolution

Risk management is defined as the "coordinated activities to direct an organization with regard to risk" ("ISO 31000 – Risk management – Principles and guidelines", 2009). A core part of these activities is a regularly conducted risk analysis to identify threats, vulnerabilities and unwanted incidents with respect to critical assets. The severity of the identified risks must be estimated, and then evaluated with respect to predefined criteria to determine which risks need to be modified by risk treatment.

Traditional approaches to risk analysis typically focus on a particular configuration of the target at a particular point in time, and are valid under the assumptions made in the analysis (Lund, Solhaug, & Stølen, 2010). However, the target of analysis, its environment and the assumptions we make will change and evolve over time, during software development, during deployment, and at runtime. Such evolutions may render previous risk analyzes invalid and require the whole risk analysis to be conducted from scratch. There is therefore a need for methods and techniques to handle change and evolution in a systematic way so as to maintain the validity of risk analysis results under change. This section gives an overview of the state of the art in this domain, focusing on security risk analysis in the software development life cycle. In the next sub-section we give an introduction to risk analysis in general, give the most important definitions and explain which artifacts that are used and produced. Subsequently we focus on security and explain the most important aspects and activities of security risk analysis. In the remaining two subsections we give an overview of the state of the art for managing and analyzing evolving risks; first we address evolving risks in general, and then we address evolving security risks.

Risk Analysis

The ISO 31000 standard ("ISO 31000 – Risk management – Principles and guidelines", 2009) defines risk management as an iterative process. The activities include the specification of the target of analysis with its focus and scope, the risk identification, analysis and estimation, and finally the risk treatment. While differing in methods and techniques for risk assessment and risk modeling, most of the established approaches to risk analysis, such as OCTAVE (Alberts & Dorofee, 2001), CORAS (Lund, Solhaug, & Stølen, 2011a) and CRAMM (Siemens, n.d.), follow the ISO 31000 process. The same process is also followed by several more security tailored approaches, such as EBIOS ("EBIOS 2010 – Expression of Needs and Identification of Security Objectives", 2010), the Microsoft Security Risk Management Guide ("The Security Risk Management Guide", 2006) and FRAAP (Peltier, 2010).

When conducting a risk analysis, there is a need for techniques and means to reason

about various aspects of risks, and to document the results, while following the overall risk management process. Risk modeling refers to techniques that are used to aid the process of identifying and estimating likelihood and consequence values. A risk model is a structured way of representing an event, its causes and consequences using graphs, trees or block diagrams (Robinson, 2007). Some well-known risk modeling techniques are fault tree analysis (FTA) ("IEC 61025 Fault Tree Analysis (FTA)", 1990), event tree analysis (ETA) ("IEC 60300-9 Dependability management Part 3: Application guide Section 9: Risk analysis of technological systems Event Tree Analysis (ETA)", 1995), attack trees (S. Schneider, 1999), cause-consequence diagrams (Robinson, 2007; Mannan, 2005) and Bayesian networks (Ben-Gal, 2007).

An inherent challenge in most risk analyzes is the modeling and assessment of uncertainty, which is often due to lack of knowledge or imprecise and insufficient data. Many traditional risk models are based on probability theory and classical set theory where uncertainty is not easily represented. For this reason various approaches based on fuzzy logic, see, e.g., (Zadeh, 1965; Cox, 1994), have been proposed. Fuzzy logic allows the uncertainty to be made explicit, and comes with rules and operations for reasoning about this uncertainty when assessing the risks.

No matter which risk analysis method and risk modeling technique has been chosen, the objective is to build and maintain a risk model that provides a valid documentation of the risks, given the *target description* as documented during the context establishment. The target description should include the assets and stakeholders of the analysis, the focus and scope, the assumptions we make, as well as the target model. The target model is a specification of the elements of the target of analysis, including software and hardware components, users and roles, information and communication networks, business and work processes, and so forth. The target model is created using a suitable notation such as activity diagrams, class diagrams, data flow diagrams or business process modeling. When using risk analysis to support the software development life cycle, such models can be received as input from, for example, the software design and architecture or from the requirements engineering. Obviously, a change in any part or aspect of the *target description* may have impact on the risk picture, requiring an updated analysis and risk model. After introducing security to the setting of risk management in the next sub-section, we proceed by presenting existing approaches to handle such change and evolution.

Security Risk Analysis

Security risk analysis can be understood as a specialization of risk analysis, where the focus is on preservation of security and the protection of information assets. The ISO/IEC 27005 ("ISO/IEC 27005 – Information technology – Security techniques – Information security risk management", 2011) standard on information security risk management builds on ISO 31000 and follows the same overall process. The purpose is to provide guidelines to support the requirements of an information security management system (ISMS) according to ISO/IEC 27001 ("ISO/IEC 27001 – Information technology – Security techniques – Information security management systems – Requirements", 2005).

For risk management in general, assets can be of any kind, for example revenue, property, market share, personnel, life and health, reputation, and so forth. Security risk management, on the other hand, typically concerns business processes and activities, as well as information assets. It is moreover

concerned with the preservation of confidentiality, integrity and availability of information and services ("ISO/IEC 27001 – Information technology – Security techniques – Information security management systems – Requirements", 2005) by preventing information security incidents. Other security properties that may be taken into account are authentication, non-repudiation and authorization (Hernan et al., 2006).

In secure software engineering, security risk analysis should be an integrated part of the development lifecycle. For this purpose, the models and specifications that are developed during security requirements engineering, secure software modeling, security architecture and security test modeling should provide input to the security risk analysis process. In an iterative engineering process, the security risk analysis has then the potential to identify possible security design flaws and provide feedback to the security engineering activities.

Risk Evolution

Evolution is unavoidable in most systems and organizations, and while systems change the associated risks change too. This is not new, and as prescribed by ISO 31000, risk management should detect "changes in the external and internal context, including changes to the risk criteria and the risk itself, which can require revision of risk treatment and priorities" ("ISO 31000 – Risk management – Principles and guidelines", 2009). However, as software and information systems become more and more heterogeneous, dynamic and interoperable, evolution becomes a critical factor that needs to be dealt with systematically. For the management and analysis of evolving risks, a main challenge is how to respond to system and software changes, either during development or at runtime. The objective is to maintain the validity of the risk model and the risk analysis results without conducting a full risk analysis from scratch every time (Lund et al., 2010; Lund, Solhaug, & Stølen, 2011b). Moreover, when risks are changing and evolving they should be analyzed and understood as such. In other words, the management of evolving risks should be supported by techniques for modeling, assessing and reasoning about risk changes. Such techniques will allow planning and proactive decisions regarding desired or possible software and system changes.

Traditional methods and techniques for risk management, including the ISO 31000 standard, are not well equipped to address evolution in a methodical and systematic way. However, the increased awareness of these challenges, both in industry and in the research communities, has led to substantial advances during the recent few years. In the remainder of this section we give an overview of some of these approaches, focusing on security in software systems.

Security Risk Evolution

For software and systems that are rapidly evolving, either during development or operation, there is a need for strong traceability between the target model and the risk model. This can be done by building and maintaining traceability links between the model artifacts of the two domains and propagate changes between them. Another approach, which is more relevant at runtime during operation, is security risk monitoring by the monitoring of security indicators or metrics in the target of analysis. Changes in software or system attributes with relevance for the security risk level can then be monitored in order to continuously assess risks and respond to unacceptable risks when they arise. In the following we describe existing approaches by considering in turn methodological support, risk modeling techniques, tool support and risk monitoring.

Methods to Analyze Evolving Risks. Lund et

al. (Lund et al., 2011b) address the problem of insufficient methodological support for handling change in traditional and standard risk management frameworks. To mitigate this they propose guidelines and techniques for systematically tracing changes from the target model to the risk model, and thereby updating only the part of the risk picture that is affected by the changes.

The contribution regarding methodology is twofold. The first contribution is a generalization of the ISO 31000 guidelines to provide support for handling change in all activities of the risk management process. The guiding principle is that only the risks that may be affected by changes in the target should be analyzed anew. The second contribution is the instantiation of the generalized guidelines in CORAS (Lund et al., 2011a). The ISO 31000 standard provides guidelines on which activities to conduct and what should be achieved in each activity, but it comes with no techniques for how to do this in practice. Such techniques, including risk modeling techniques, are typically offered by risk analysis frameworks that instantiate the standard, such as OCTAVE, CORAS and CRAMM. The goal of generalizing the standard to support change management is to offer the necessary generic guidelines that can be instantiated in any approach that is compliant with the standard, as demonstrated with CORAS by Lund et al. (Lund et al., 2011b).

An important aspect of this generalized approach is the support for explicitly documenting risk changes. Hence, the approach not only supports updating and maintaining the validity of risk models under change, but also the modeling and assessment of how the risks evolve while the target of analysis evolves. For this purpose, the risk graph notation (Brændeland, Refsdal, & Stølen, 2010) for formal risk modeling and analysis is generalized to capture change. The risk graph notation is moreover extended with support for modeling the traceability links to the target model. Further details on this modeling support are presented in the next subsection.

At the level of methodology, we are not aware of any other methods or frameworks that embed change and evolution management as an explicit aspect of the whole security risk management process. The relevance and importance of change management is of course widely recognized as indicated, not only by ISO 31000, but also established methods like OCTAVE (Alberts & Dorofee, 2001). The recommendations and guidelines are, however, usually limited to the general monitoring and reviewing activities.

Modeling Evolving Risks. While the state of the art on evolution in risk management and security risk assessment is limited at the level of methodology, there has been more progress at the level of risk modeling. These are techniques that can be utilized by analysts to handle change and evolution in a systematic and efficient way, even when the underlying methodology as such is more traditional and conventional, for instance those based on ISO 31000 or ISO/IEC 27005.

Some of the established techniques for risk and threat modeling facilitate automatic updating of the values that are annotated on the diagrams; by changing input values to capture changes in the target of analysis, the derived output values can be generated. These techniques include fault trees ("IEC 61025 Fault Tree Analysis (FTA)", 1990), Markov models (R. A. Howard, 1971; "IEC 61165 Application of Markov Techniques", 1995) and Bayesian networks (BenGal, 2007). Influence diagrams (R. A. Howard & Matheson, 2005) were originally a graphical language designed to support decision making by specifying the factors influencing a decision. In (EEC, 2006), such diagrams are connected to the leaf nodes of fault trees supporting the propagation of influence to

the unwanted incidents specified at the root of the tree. Similar, but simpler, are the risk influence diagrams, detailed in (Aven, Sklet, & Vinnem, 2006), where influencing factors are connected to the nodes in event trees.
Several other notations have support for associating elements of risk models to parts of the target description, which may facilitate the identification of possible risk changes due0 to target changes. Approaches based on the UML, such as misuse cases (Sindre & Opdahl, 2000), may utilize built-in mechanisms in the UML for relating elements from different UML diagrams that serve as the target model.

As mentioned above, Lund et al. (Lund et al., 2011b) make use of risk graphs (Brændeland et al., 2010) to provide modeling support for their process and guidelines for security risk management of changing and evolving systems. An advantage of risk graphs is that they can be understood as a common abstraction of several established risk modeling techniques, such as fault trees, event trees, cause-consequence diagrams, Bayesian networks and CORAS threat diagrams. Lund et al. extend the risk graph notation with support for specifying risk elements that emerge after change, risk elements that become obsolete, and risk elements that are modified. Semantics is provided for this extension, and the risk graph calculus is extended to provide support for the reasoning about risk graphs with change. The syntax is moreover extended with support for relating risk graph elements to elements of the target model. The specification of these relations is referred to as the trace model, as it facilitates the systematic traceability of changes from the target model to the risk model.

Thales Research & Technology has developed their own industrial model-based approach to risk assessment, supported by the Rinforzando (Paul & Delande, 2011; Bergomi, Paul, Solhaug, & Vignon-Davillier, 2013) tool. The security risk assessment and modeling can be performed as standalone, but is also designed to serve as an integrated part of their mainstream system engineering workbench (Voirin, 2008). For this purpose, dynamic links can be built and maintained between the risk models and the system engineering models, the latter specified using a service-oriented architecture (SOA) modeling suit. When any model changes are implemented during the system development process, either on the risk model or the system model, the changes are immediately propagated via the links to trigger updates and maintain the mutual consistency between the modeling domains.

The problem of traceability between model artifacts is well-addressed in the model-driven engineering (MDE) community, where a strong trend is to develop a viewpoint ("ISO/IEC/IEEE 42010 – Systems and software engineering – Architecture description", 2011) for each engineering concern. Each viewpoint should come with its own modeling and analysis techniques, but also with means for mapping of its model artifacts to the related artifacts of the other viewpoints. In the context of safety critical systems, such traceability between the system development process and the risk assessment process is proposed in (Katta & Stålhane, 2011).

Tool Support for Analyzing Evolving Risks. A full risk analysis will typically result in a large number of risk diagrams covering different parts of the target model. Without any automated tool support, the task of tracing changes from the target model to the risk model must be conducted manually and can easily become infeasible.

The tool presented in (Seehusen & Solhaug, 2012; Solhaug & Seehusen, 2013) is developed to support the CORAS instantiation of the method and language for security risk assessment of evolving systems proposed by Lund et al. (Lund et al., 2011b). The main feature of the tool is the diagram editor for creating all kinds of CORAS

diagrams to model and assess changing and evolving risks. However, the tool also supports the specification of the trace model, i.e., the traceability links between elements of the risk model and elements of the model of the target of analysis. Using the trace model the tool automatically flags all risk diagrams and elements that may be affected by changes in the target and therefore need to be re-assessed. The tool also comes with automated support for detecting and resolving inconsistencies that may arise during the process of updating the risk models.

The Rinforzando (Paul & Delande, 2011) tool mentioned above is similar in the sense of using traceability links between the risk model and the system model to maintain validity and mutual consistency. The integration with their system engineering process is hard-coded and much tighter than what is offered by CORAS. However, this is at the cost of general applicability as Rinforzando is tied to the Thales engineering workbench, whereas CORAS allows any notation to be used for target modeling. The integration with the mainstream system engineering using Rinforzando not only allows establishing the traceability links and maintaining online consistency between the domains; it also allows the annotation of design elements in the engineering workbench when they are linked to risk model elements, in order to support engineers in detecting possible security design flaws.

Model Versioning and Evolution (MoVE) (Breu, Breu, & Löw, 2011) is an approach to build an infrastructure to maintain the validity, mutual consistency and interdependencies between models as they evolve over time within MDE. The approach does not target security and risk in particular, but rather builds a tool-supported infrastructure for versioning of several interdependent models, for example for software architecture and design, business processes, services, security and risk. Similar to the aforementioned tools, the underlying idea is to provide support for tracing changes from one model to another to ensure that they are globally up-to-date and mutually consistent

Monitoring Evolving Risks. Risk monitoring is a means to facilitate continuous risk assessment by the monitoring of relevant key indicators or metrics. In order to enable security risk monitoring there is a need not only to identify the relevant indicators, but also to understand how to relate the indicators to potential security risks, and how to aggregate the monitored values into risk levels. The benefit of security risk monitoring is, of course, that risk assessment results can be automatically updated as they evolve while the target of analysis evolves.

Refsdal and Stølen (2009) present a model-based approach to make use of measurable indicators in order to obtain a risk picture that is continuously or periodically updated. The approach comes with a process of three steps. First, an initial risk analysis is conducted to identify and model possible threat scenarios and unwanted incidents. Second, key indicators are identified that may be relevant for determining likelihoods and consequences for the risk model. Third, functions are defined for calculating likelihoods and consequences based on the indicators. Using this model-based approach, managers and other stakeholders are provided a high-level view of the current system security by observing the updates of the risk models.

A similar approach is proposed by Ligaarden et al. (Ligaarden, Refsdal, & Stølen, 2012b). However, they focus on the security of dynamic services in the more complex setting of systems of systems. The latter are collections of systems interconnected through the exchange of services. The authors propose a method to support the capturing and the monitoring of the impact of service dependencies on the

security of the provided services. The method is divided into four main steps: (1) documenting the system of systems and IT service dependencies, (2) establishing the impact of service dependencies on the security risk of provided services, (3) identifying measurable indicators for dynamic monitoring, and (4) specifying the indicator design and use. In a different publication (Ligaarden, Refsdal, & Stølen, 2012a), the same authors address the related problem of designing the indicators to be monitored. For the security risk monitoring to be correct, it is of course crucial that the selected indicators provide a valid view of the risk picture and the monitored risk level.

Adequate tool-support is obviously a necessity for enabling security risk monitoring and the continuous aggregation of measured indicator values to generate the updated risk levels. Ligaarden et al. (2011) propose an architectural pattern for enterprise level monitoring tools. Their idea is that the pattern should serve as a generic basis for building tools with features for collecting low-level data from the ICT infrastructure, aggregating the collected low-level data, evaluating the aggregated data, presenting the aggregated data and the evaluation results to different stakeholders, as well as features for doing the necessary configurations.

Krautsevich et al. (2010) propose an approach to make use of runtime attribute monitoring to support risk-based enforcement of usage control (UCON) policies. The approach targets the dynamic nature of UCON where authorization may change over time. Because UCON decisions are based on mutable attributes, the values of which evolve, the reference monitor continuously needs to re-evaluate the enforcement decisions. Correctly registering all attribute changes is challenging, especially if the attribute provider and the reference monitor reside in different security domains; changes may be missed, delayed or event corrupted. There is therefore a risk of granting erroneous access and usage. To mitigate this, the authors propose a set of policy enforcement models with tolerance of the inherent uncertainties of current attribute values. In these models, the reference monitor evaluates logical predicates over attributes as usual, but additionally makes estimates on how much the observed attribute values differ from the actual values. By considering the cost of erroneous enforcement combined with its probability, the associated risk is calculated. The risk assessment then serves as a basis for decision making. Although the main purpose of these approaches is not security risk analysis as such, but rather usage control, they are still relevant in the setting of runtime security assessment and risk monitoring because of the use of runtime risk analysis of evolving systems.

Conclusion

The increased usage of new service-based computing paradigms, like service-oriented architecture and cloud computing, results in software systems that are distributed, open, complex and dynamically changing. Security is one of the main issues that must be tackled in such environments. Especially, it is challenging to face the changing and evolutionary nature of such systems. In this article we provided a comprehensive state of the art survey regarding security evolution from a software and security engineering perspective. We considered the individual phases of the security engineering lifecycle and their associated artifacts separately. These include *modeling*, *analysis*, *design*, *implementation*, *testing*, *deployment* and *operation* as well as *risk management*. The associated artifacts are *models*, *requirements*, *architectures*, *code*, *tests*, *runtime monitoring* and *risks*, respectively.

In the following, we first summarize the key issues for each artifact presented in the

article and then discuss resulting directions of future work.

Summary

In this section, we summarize the key issues for each artifact type presented in this article, i.e., models, requirements, architecture, code, tests, monitoring as well as risks. First, we list the references to security evolution publications for each artifact in Table 1 to provide a comprehensive overview, and then we discuss each artifact type in detail.

Artifact	*References*
Models	(Koch et al., 2001), (Goncalves & Poniszewska-Maranda, 2008), (Jürjens et al., 2011), (Montrieux et al., 2011), (Ochoa, Jürjens, & Cuéllar, 2012), (Ochoa, Jürjens, & Warzecha, 2012), (Ruhroth & Jürjens., 2012)
Requirements	(Bergmann et al., 2011), (Massacci et al., 2011) Architecture
Architecture	(Hafiz & Johnson, 2008), (Yskout, Scandariato, & Joosen, 2012a)
Code	(De Win et al., 2002), (Pierce, 2002) , (Jacobs et al., 2011)
Tests	(Kongsli, 2006), (Felderer, Agreiter, & Breu, 2011), (Hwang et al., 2012), (SecureChange, 2012)
Monitoring	(Gao et al., 2010), (Binde et al., 2011), (Zargar et al., 2011) , (Bencsáth et al., 2012), (Sood & Enbody, 2013)
Risks	(EEC, 2006), (Krautsevich et al., 2010), (Refsdal & Stølen, 2009), (Lund et al., 2011b), (Ligaarden et al., 2012b), (Ligaarden et al., 2012a)

Table 1. References to Security Evolution Publications for each Artifact

Models. The main artifacts being used in model-driven software and security engineering methodologies are models. A plethora of modeling languages has been developed for specifying models for different purposes. However, UML is considered a de-facto industrial standard for modeling and was the focus of this survey. Using UML as a modeling language in the security engineering process has been considered in different lines of work, in which security issues have been incorporated in the design models at early stages of the software development lifecycle. With regard to change management, different approaches have been proposed to consider evolution of (UML) design models throughout all development phases. However, the impact of change on the security of UML models has not been investigated in detail so far. An example of a framework that considers evolution aspects on security of UML models is UMLSec. In this context, the UMLSeCh approach has been developed to tackle the consistency problems of selected UML requirements when models evolve. Other approaches consider evolution of access control policies.

Requirements. A security requirements engineering process consists of three phases: (1) asset identification, (2) security goal elicitation, and finally, (3) security requirement specification. The main security requirements engineering approaches that can be identified are, goal-based, problem-based, and risk-based security requirements engineering approaches. Evolution in security requirements engineering indicates how new security needs can be accommodated in the requirements models and specifications. The recent work with regard to requirements evolution can be considered from different perspectives. Some focus on modeling the evolution, others considers the problems that arise from evolution, like inconsistency, and finally, other work considers the methods and tools to assess and manage the impact of change. While requirement evolution has been

extensively researched, tackling security issue while introducing changes to the requirement model has not been well studied. The only available approach is a model-driven methodology to represent, analyze and detect security issues that are because of requirements' evolution.

Architecture. Software architecture can be defined as a set of elements, their relationships, and some degree of rationale. Security at the architectural level may be considered form different points of view: constructive, modeling and analysis points of views. The first perspective tackles the problem of *how* architectures can be created with certain security properties. The second issue considers modeling security at the architectural level. Finally, the analysis viewpoint allows performing formal analysis on the design of the architecture. With regard to evolution, changes of the system architecture originate form a change in the requirements or the environment of the system. These changes at the architectural level should be propagated further to other deployment artifacts.
Thus, the evolution of the software architecture is tightly related with the evolution of requirement, code, and deployment configuration. Besides security evolution of UML models which is certainly relevant for the software architecture as well, patterns to securely evolve a system and its architecture are under investigation.

Code. Secure code is the central artifact in the software engineering process, and it is tightly associated with other artifacts, e.g., requirements, architecture, testing, and the deployment. The properties of the programming language, in which the secure code is written, that affect security are safety and modularity. Vulnerabilities or code bugs with security consequences, that are associated with coding, can be categorized into vulnerabilities in the (1) infrastructural software, (2) security components, and (3) applications. Examples of countermeasure techniques against such vulnerabilities are static verification, code guidelines and runtime monitoring. Code evolution indicates the changes that must be done to the software code after being deployed. While modularity is an important property to support evolution of secure code, security functionality is often extremely hard to modularize. This crosscutting nature of security concerns makes their implementation error-prone and hard to maintain. For this reason, aspect-oriented programming can identify itself as the best solution, which is well studied in the literature. Code evolution mainly interacts with vulnerabilities in security components and vulnerabilities in applications. Vulnerabilities in security components are mainly addressed by new modularization concepts that are better at modularizing crosscutting concerns such as access control. To make sure that no vulnerabilities in the application logic itself are introduced during evolution, the reasoning about their absence is made modular. Two important approaches succeed in modularizing the analysis for security vulnerabilities, i.e., type systems and modular full functional verification.

Tests. Security testing is software testing of security requirements like confidentiality, integrity, authentication, authorization, availability, and non-repudiation. Security testing can be divided into security functional testing (testing positive security requirements) and security vulnerability testing (testing negative security requirements). An important type of software testing in the context of (security) evolution is regression testing, i.e., the selective retesting after changes have been made to the system under test (SUT). Its goal is to ensure that (1) modifications have not introduced new faults or caused unintended effects and (2) the SUT still compiles with the specified requirements. Regression testing has been well studied for

classical functional requirements. A great attention is given to testing different variants of positive and negative security requirements like security properties, security functionality as well as vulnerabilities. However, very little research has been done in the field of regression testing for security requirements. The few available security regression testing approaches focus on testing security properties and security functionality like access control policies.

Monitoring. Security monitoring is the processing of evaluating software artifacts during operation in order to detect security violations. Violation detection can occur on different levels of abstraction, e.g., technical or business levels, and for different types, e.g., security policy violation, data leakage prevention, fraud detection, or workflow compliance violation. The reasoning or analysis component of a security monitoring solution can be based either on signatures, e.g., complex event processing, or statistical, e.g., machine learning, algorithms. The first performs rule-based analysis to detect intrusions while the latter uses statistical algorithms to detect anomalies. In order to overcome and detect intrusions or anomalies, either signatures or statistical distributions of the attack must be known. We interpreted "security monitoring evolution" as the fact that when attacks evolve and their patterns change into unknown schemas, the security monitoring solution has to evolve accordingly. Beside machine learning based approaches, which aim at measuring the change in the behavior, artificial immune system based intrusion detection system have been studied for this purpose.

Risks. Risk management is a process that involves coordinated activities to direct organization decisions with regard to risks. The key element in this process is a risk analysis activity, whose goal is to identify threats and vulnerabilities with respect to critical assets. Security risk analysis focuses on the preservation of security and the protection of information assets. In the context of the software development lifecycle, the software artifacts under development are the target of (security) risk analysis. While traditional approaches of risk analysis focus on a particular configuration of the target at a specific point of time, target evolution was not been considered in these approaches. Methods to analyze evolving risks propose guidelines and techniques for systematically tracing changes from the target model to the risk mode, and thereby only updating the part of the risk model affected by the changes. Besides the research on methods that tackles evolution in risk management, modeling and monitoring evolving risk modeling. The idea of modeling evolving risk is to associate elements of risk models to parts of the target model, which facilitates the identification of possible risk changes when the target model evolves. Monitoring evolving risks is based on security risk monitoring and automatically updates risk assessment results while the target of analysis evolves.

Discussion and future work

This article reveals that evolution of artifacts in the different phases of the software development lifecycle is tightly coupled. Evolution typically starts with changes of requirements which are propagated further to the design, architectural and implementation artifacts. This in turn impacts testing, deployment and risk management. Although the artifacts are tightly coupled, actual work on evolution of security artifacts mostly discusses security evolution for a specific artifact type in isolation. Modeling can be seen as an overlapping aspect across all phases to overcome this limitation. As discussed in this state of the art survey, models play an important role in (1) defining security requirements, (2) designing

architectures, (3) automating security test case generation, especially for regression testing purposes, and (4) supporting the risk management in the process of defining and documenting risks. Thus, modeling is an enabler to manage evolution of arbitrary security artifacts in an integrated way for the entire software development lifecycle. In this article we focused on the graphical de-facto standard modeling language UML and provided a comprehensive review of change and evolution aspects in this regard, especially in the line of the UMLSec approach. Modeling tools and techniques, however, go beyond UML. For example, domain specific languages recently gained great attention in research and industry. However, their potential for security evolution has not been exploited.

This state of the art survey showed that security evolution is an actual area of research founded in software evolution and security engineering which requires further investigation. The security evolution research for some artifacts of the software development lifecycle is still rare and needs further investigation. At the modeling level, research on the impact of change to the security of UML models is still at the beginning. While requirements evolution has been extensively researched, tackling security issue while introducing changes to the requirement model has not been well studied so far. The same situation holds for architectures and regression testing which both have been studied extensively (H. P. Breivold et al., 2012; Yoo&Harman, 2010). But approaches to security evolution on the architectural level as well as regression testing approaches for specific security properties are still rare. It would be valuable to investigate how existing techniques for architectural evolution support security. For example, the use of graph transformations to model security architecture evolution and modifiability analysis technique to assess the impact of a certain evolution scenario on the security architecture could be of interest. Evolution for the remaining security artifacts of the software development lifecycle is better studied as remarkable work has been carried out in the areas of secure code evolution, security monitoring, and risk evolution. But for the security evolution of all artifact types, empirical studies, especially in an industrial context, are missing so far.

Acknowledgements

This work is partially funded by the EternalS Coordination Action (FP7-247758), the EU projects NESSoS (FP7-256980) and SecureChange (FP7-231101), the FFG project “QE LaB Living Models for Open Systems” (FFG 822740), the FWF project MOBSTECO (FWF P 26194-N15), the DFG project “SecVolution” which is part of DFG Priority Program 1593 Design for Future, the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the Research Fund KU Leuven, as well as the Fonds National de la Recherche Luxembourg (FNR/P10/03).
Thein Tun is supported by NPRP grant 05-079-1-018 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

Abbott, R., Chin, J., Donnelley, J., Konigsford, W., Tokubo, S., & Webb, D. (1976). Security analysis and enhancements of computer operating systems (Tech. Rep.). DTIC Document.

Abi-Antoun, M., Wang, D., & Torr, P. (2007). Checking threat modeling data flow diagrams for implementation conformance and security. In 22nd IEEE/ACM international conference on automated software engineering (pp. 393–396).

Ahmad, A., Jamshidi, P., & Pahl, C. (2012). Pattern-driven reuse in architecture-centric evolution for service software. In 7th international conference on software paradigm trends ICSOFT'2012.

Alberts, C. J., & Dorofee, A. J. (2001). OCTAVE Criteria (Tech. Rep. No. CMU/SEI-2001-TR-016). CERT.

Alebrahim, A., Hatebur, D., & Heisel, M. (2011). To-wards systematic integration of quality requirements into software architecture. In I. Crnkovic, V. Gruhn, & M. Book (Eds.), Software architecture (p. 17-25). Springer.

Alghathbar, K., & Wijesekera, D. (2003). Consistent and complete access control policies in use cases. In P. Stevens, J. Whittle, & G. Booch (Eds.), UML (pp. 373–387). Springer.

Alienvault. (n.d.). Ossim documentation. http://www.alienvault.com/community.php?section= Docs [accessed: January 15, 2013].

Alnatheer, A., Gravell, A. M., & Argles, D. (2010). Agile security issues: an empirical study. In Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement (pp. 58:1– 58:1). New York, NY, USA: ACM.

Anderson, R. (1993). Why cryptosystems fail. Proceedings of the ACM Conference in Computer and Communications Security, 215–227.

Andreae, C., Noble, J., Markstrum, S., & Millstein, T. (2006). A framework for implementing pluggable type systems. ACM SIGPLAN Notices., 41(10), 57–74.

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Taentzer, G. (1999). Graph transformation for specification and programming. Science of Computer Programming, 34(1), 1 – 54.

Austin, T. H., & Flanagan, C. (2012). Multiple Facets for Dynamic Information Flow. In Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

Aven, T., Sklet, S., & Vinnem, J. E. (2006). Barrier and operational risk analysis of hydrocarbon releases (BORA-Release). Part I. Method description. J. Haz. Mat., A137, 681–691.

Axelsson, S. (2000). Intrusion detection systems: A survey and taxonomy (Tech. Rep.). Technical report.

Balasubramaniyan, J., Garcia-Fernandez, J., Isacoff, D., Spafford, E., & Zamboni, D. (1998). An architecture for intrusion detection using autonomous agents. In Computer security applications conference, 1998. Proceedings. 14th annual (pp. 13–24).

Barais, O., Le Meur, A.-F., Duchien, L., & Lawall, J. L. (2008). Software architecture evolution. In Software evolution (p. 233-262). Springer.

Baresi, L., Guinea, S., & Plebani, P. (2006a). Ws-policy for service monitoring. In C. Bussler & M.-C. Shan (Eds.), Technologies for e-services (Vol. 3811, p. 72-83). Springer Berlin Heidelberg.

Baresi, L., Guinea, S., & Plebani, P. (2006b). WS-Policy for service monitoring. Technologies for E-Services, 72–83.

Barthe, G., D'argenio, P. R., & Rezk, T. (2011). Secure information flow by self-composition. Mathematical. Structures in Comp. Sci., 21(6), 1207–1252.

Basin, D., Doser, J., & Lodderstedt, T. (2003). Model driven security for process-oriented systems. In Proceedings of the eighth ACM symposium on access control models and technologies (pp. 100–109). New York, NY, USA: ACM.

Basin, D. A., Doser, J., & Lodderstedt, T. (2006). Model driven security: From EDOC models to access control infrastructures. ACM Trans. Software. Engineering Methodology, 15(1), 39-91.

Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice (Second ed.). Addison-Wesley.

Bauer, L., Ligatti, J., & Walker, D. (2005). Composing security policies with polymer. In Pldi '05: Proceedings of the 2005 ACM sigplan conference on programming language design and

implementation (pp. 305–314). New York, NY, USA: ACM Press.

Bencsáth, B., Pék, G., Buttyán, L., & Félegyházi, M. (2012). Duqu: Analysis, detection, and lessons learned. In ACM European workshop on system security (EUROSEC) (Vol. 2012).

Ben-Gal, I.(2007). Bayesian networks. In F. Ruggeri R. S. Kenett, & F. W. Faltin (Eds.), Encyclopedia of statistics in quality and reliability. John Wiley & Sons

Bengtsson, P., Lassing, N., Bosch, J., & van Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). The Journal of Systems & Software, 69(1-2), 129–147.

Bergmann, G., Massacci, F., Paci, F., Tun, T. T., Varró, D., & Yu, Y. (2011). A tool for managing evolving security requirements. In Caise forum (selected papers) (p. 110-125).

Bergomi, F., Paul, S., Solhaug, B., & Vignon-Davillier, R. (2013). Beyond traceability: Compared approaches to consistent security risk assessments. In Proc. international workshop on security in air traffic management and other critical infrastructures (secatm'13). (To appear)

Bernardi, S., & Merseguer, J. (2007). A EDOC profile for dependability analysis of real-time embedded systems. In Proceedings of the 6th international workshop on software and performance (pp. 115–124).

Binde, B., McRee, R., & O'Connor, T. (2011). Assessing outbound traffic to uncover advanced persistent threat. SANS Institute. Whitepaper.

Blobel, B. (2002). Aspects of modeling using the examples of Electronic Health Records (EHRs). In Coras workshop. (Part of International Conference on Telemedicine (ICT2002))

Bolton, R., & Hand, D. (2002). Statistical fraud detection: A review. Statistical Science, 235–249.

Brændeland, G., Refsdal, A., & Stølen, K. (2010). Modular analysis and modelling of risk scenarios with dependencies. Journal of Systems and Software, 83(10), 1995–2013

Braude, E., & Bernstein, M. (2011). Software engineering: Modern approaches. J. Wiley & Sons.

Breivold, H., Crnkovic, I., & Larsson, M. (2012). Software architecture evolution through evolvability analysis. Journal of Systems and Software, 85, 2574–2592.

Breivold, H. P., Crnkovic, I., & Larsson, M. (2012). A systematic review of software architecture evolution research. Information and Software Technology, 54(1), 16 – 40.

Breu, M., Breu, R., & Löw, S. (2011). MoVEing forward: Towards an architecture and processes for a Living Models infrastructure. International Journal on Advances in Life Sciences, 3(1-2), 12-22.

Brier, J., Rapanotti, L., & Hall, J. (2006). Problem-based analysis of organisational change: a real-world example. In Proc. of iwaapf '06. ACM.

Brose, G., Koch, M., & Löhr, K.-P. (2002). Integrating access control design into the software development process. In Integrated design and process technology (IDPT).

Bryl, V., Giorgini, P., & Mylopoulos, J. (2009). Designing socio-technical systems: from stakeholder goals to social networks. Requirement Engineering, 14(1), 47– 70.

Cardelli, L. (1997). Type systems. In The computer science and engineering handbook (p. 2208-2236). CRC press.

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. ACM Computing Surveys (CSUR), 41(3), 15.

Chechik, M., Lai, W., Nejati, S., Cabot, J., Diskin, Z., Easterbrook, S., Salay, R. (2009). Relationship-based change propagation: A case study. In Proceedings of the 2009 ICSE workshop on modeling in software engineering (pp. 7–12). Washington, DC, USA: IEEE Computer Society.

Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Tobies, S. (2009). Vcc: A practical system for verifying concurrent c. In Proceedings of the 22nd

international conference on theorem proving in higher order logics (pp. 23–42). Springer.

Côté, I., Heisel, M., & Wentzlaff, I. (2007). Pattern-based exploration of design alternatives for the evolution of software architectures. Int. J. Cooperative Inf. Syst., 16(3/4), 341-365.

Cox, E. (1994). The fuzzy systems handbook: A practitioner's guide to building, using, and maintaining fuzzy systems. Academic Press Professional.

Dai, L., & Cooper, K. (2007). A survey of modelling and analysis approaches for architecting secure software systems. International Journal of Network Security, 5(2), 187–198.

d'Avila Garcez, A., Russo, A., Nuseibeh, B., & Kramer, J. (2003). Combining adductive reasoning and inductive learning to evolve requirements specifications. In IEEE Proceedings - software (Vol. 150(1), p. 25-38).

Debar, H., Becker, M., & Siboni, D. (1992). A neural network component for an intrusion detection system. In Research in security and privacy, 1992. Proceedings. 1992 IEEE computer society symposium on (pp. 240–250).

Debar, H., Curry, D., & Feinstein, B. (2007). The intrusion detection message exchange format (IDMEF).

De Groef, W., Devriese, D., Nikiforakis, N., & Piessens, F. (2012). Flowfox: a web browser with flexible and precise information flow control. In Proceedings of the 19th ACM conference on computer and communications security (ccs 2012). ACM.

Denning, D. (1987). An intrusion-detection model. Software Engineering, IEEE Transactions on (2), 222–232. Desmet, L., Piessens, F., Joosen, W., & Verbaeten, P. (2006).

Bridging the Gap between Web Application Firewalls and Web Applications. In Proceedings of the 2006 ACM workshop on formal methods in security engineering (pp. 67–77).

Devriese, D., & Piessens, F. (2010). Noninterference through Secure Multi-Execution. In Proceedings of the IEEE symposium on security and privacy (pp. 109–124).

De Win, B., Joosen, W., & Piessens, F. (2004). Developing secure applications through aspect-oriented programming. Addison-Wesley.

De Win, B., Piessens, F., Joosen, W., & Verhanneman, T. (2002). On the importance of the separation-of-concerns principle in secure software engineering. Workshop on the Application of Engineering Principles to System Security Design, 1–10.

Ebios 2010 – expression of needs and identification of security objectives [Computer software manual]. (2010). (In French)

EEC. (2006). Methodology report for the 2005/2012 integrated risk picture for Air Traffic Management in Europe [Computer software manual]. (EEC Technical/Scientific Report No. 2006-041)

Elahi, G., Yu, E., & Zannone, N. (2009). A vulnerability-centric requirements engineering framework: analyzing security attacks, countermeasures, and requirements based on vulnerabilities. Requirements Engineering, 15(1), 41–62.

Erlingsson, U (2004). The inlined reference monitor approach to security policy enforcement (Unpublished doctoral dissertation). Ithaca, NY, USA. (AAI3114521)

Erlingsson, U., & Schneider, F. B. (2000). Irm enforcement of java stack inspection. In In IEEE symposium on security and privacy (pp. 246–255).

Erlingsson, U., Younan, Y., & Piessens, F. (2010). Low-level software security by example. Springer.

Ernst, N. A., Borgida, A., & Jureta, I. (2011). Finding incremental solutions for evolving requirements. In Re (p. 15-24). IEEE.

Erradi, A., Maheshwari, P., & Tosic, V. (2007). WS-Policy based monitoring of composite web services.

Estévez-Tapiador, J., Garcıa-Teodoro, P., & Dıaz-Verdejo, J. (2004). Measuring normality in

http traffic for anomaly-based intrusion detection. Computer Networks, 45(2), 175–193.

Fabbrini, F., Fusani, M., Gnesi, S., & Lami, G. (2007). Controlling requirements evolution: a formal concept analysis-based approach. In Proceedings of the international conference on software engineering advances. Washington, DC, USA: IEEE Computer Society.

Felderer, M., Agreiter, B., & Breu, R. (2011). Evolution of security requirements tests for service-centric systems. In Engineering secure software and systems: Third international symposium, ESSOS 2011 (pp. 181–194). Springer.

Felderer, M., Agreiter, B., Zech, P., & Breu, R. (2011). A classification for model-based security testing. In The third international conference on advances in system testing and validation lifecycle (valid 2011) (pp. 109–114).

Felderer, M., Kalb, P., Agreiter, B., Breu, R., Buyens, K., Farwick, M., Yskout, K. (2011). Survey on state of the art time awareness and management (Tech. Rep.). Deliverable 1.2 of the EternalS Coordination Action (FP7-247758).

Felici, M. (2004). Observational models of requirements evolution (Unpublished doctoral dissertation). University of Edinburgh.

Ferguson, N., & Schneier, B. (2003). A cryptographic evaluation of IPSEC (Tech. Rep.). Counterpane Internet Security, Inc. Retrieved from http://www.schneier .com/paper-ipsec.html

FindBugs. (n.d.). Find Bugs in Java Programs. (http://findbugs.sourceforge.net/ [accessed: January 15, 2013])

Fourneret, E., Bouquet, F., Dadeau, F., & Debricon, S. (2011). Selective test generation method for evolving critical systems. In Proceedings of the 2011 IEEE 4th international conference on software testing, verification and validation workshops (pp. 125–134). Washington, DC, USA: IEEE Computer Society.

France, R. B., & Bieman, J. M. (2001). Multi-view software evolution: A EDOC-based framework for evolving object-oriented software. In ICSM.

Franqueira, V. N. L., Tun, T. T., Yu, Y., Wieringa, R., & Nuseibeh, B. (2011). Risk and argument: A risk-based argumentation method for practical security. In Re 2011, 19th IEEE international requirements engineering conference (p. 239-248). IEEE.

Gabor Bergmann, Zoltan Ujhelyi, Istvan Rath, & Daniel Varro. (2011). A graph query language for EMF models. In 4th international conference theory and practice of model transformations (ICMT 2011) (p. 167-182). Springer.

Gander, M., Felderer, M., Katt, B., & Breu, R. (2012). Monitoring anomalies in it-landscapes using clustering techniques and complex event processing. In Machine learning for system construction (MLSC) 2011 (Vol. 336). Springer.

Gander, M., Katt, B., Felderer, M., & Breu, R. (2011). Towards a model- and learning-based framework for security anomaly detection. In Formal methods for components and objects (FMCO) 2011. Springer.

Gao, W., Morris, T., Reaves, B., & Richey, D. (2010). On scada control system command and response injection and intrusion detection. In ECRIME researchers summit (ECRIME), 2010 (pp. 1–9).

Garcia-Teodoro, P., Diaz-Verdejo, J., Macia-Fernandez, G., & Vazquez, E. (2009). Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security, 28(1-2), 18–28.

Garlan, D., Barnes, J., Schmerl, B., & Celiku, O. (2009a). Evolution styles: Foundations and tool support for software architecture evolution. In WICSA/ECSA 2009 (p. 131 -140).

Garlan, D., Barnes, J. M., Schmerl, B. R., & Celiku, O. (2009b). Evolution styles: Foundations and tool support for software architecture evolution. In Joint working IEEE/IFIP conference on software architecture and european conference on software architecture (WICSA/ECSA 2009) (p. 131-140). IEEE Computer Society.

Georg, G., France, R., & Ray, I. (2002). An aspect-based approach to modeling security concerns. In Critical systems development with EDOC (CSDUML 2002) (pp. 107– 120).

Georg, G., France, R., & Ray, I. (2003). Creating security mechanism aspect models from abstract security aspect models. In Critical systems development with UML (CSDUML 2003) (pp. 35–46).

Ghose, A. (1999). A formal basis for consistency, evolution and rationale management in requirements engineering. In Ictai (p. 77-84).

Ghose, A. (2000). Formal tools for managing inconsistency and change in re. In IWSSD ’00. Washington, DC, USA.

Giblin, C., Liu, A., Müller, S., Pfitzmann, B., Zhou, X., & Building, H. (2005). Regulations expressed as logical models (REALM). In Legal knowledge and information systems: Jurix 2005: the eighteenth annual conference (p. 37).

Giorgini, P., Massacci, F., & Zannone, N. (2005). Security and trust requirements engineering. , 237-272.

Goncalves, G., & Poniszewska-Maranda, A. (2008). Role engineering: From design to evolution of security schemes. Journal of Systems and Software, 81(8), 1306-1326.

Gu, G., Perdisci, R., Zhang, J., & Lee, W. (2008). Botminer: clustering analysis of network traffic for protocol-and structure-independent botnet detection. In Proceedings of the 17th conference on security symposium (pp. 139–154).

Hafiz, M., Adamczyk, P., & Johnson, R. (2007). Organizing security patterns. Software, IEEE, 24(4), 52–60.

Hafiz, M., & Johnson, R. (2008). Evolution of the MTA architecture: The impact of security. Software: Practice and Experience, 38(15), 1569–1599.

Haley, C. B., Laney, R. C., Moffett, J. D., & Nuseibeh, B. (2008). Security requirements engineering: A frame work for representation and analysis. IEEE Trans. Software Eng., 34(1), 133-153.

Halfond, W. G., Viegas, J., & Orso, A. (2006). A Classification of SQL-Injection Attacks and Countermeasures. In Proceedings of the IEEE international symposium on secure software engineering. Arlington, VA, USA.

Hamlen, K. W., Morrisett, G., & Schneider, F. B. (2006). Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst., 28(1), 175–205. Hassine, J., Rilling, J., Hewitt, J., & Dssouli, R. (2005). Change impact analysis for requirement evolution using use case maps. In IWPSE ’05.

Hawkins, J., & Fernandez, E. (1997). Extending use cases and interaction diagrams to develop distributed system architecture requirements (Tech. Rep. No. TR-CSE-97-47). Department of Computer Science & Engineering, Florida Atlantic University.

Heaven, W., & Letier, E. (2011). Simulating and optimising design decisions in quantitative goal models. In Requirements engineering conference (re), 2011 19th IEEE international (p. 79 -88).

Heckel, R. (1998). Compositional verification of reactive systems specified by graph transformation. In Proceedings of international conference on fundamental approaches to software engineering (FASE) (pp. 138– 153). Springer.

Heldal, R., & Hultin, F. (2003). Bridging model-based and language-based security. In E. Snekkenes & D. Gollmann (Eds.), 8th european symposium on research in computer security (ESORICS 2003) (pp. 235–252). Springer.

Hernan, S., Lambert, S., Ostwald, T., & Shostack, A. (2006). Threat modeling – uncover security design flaws using the STRIDE approach.

Holmes, T., Zdun, U., Daniel, F., & Dustdar, S. (2010). Monitoring and Analyzing Service-Based Internet Systems through a Model-Aware Service Environment. In Advanced information systems engineering (pp. 98– 112).

Horng, S., Su, M., Chen, Y., Kao, T., Chen, R., Lai, J., & Perkasa, C. (2011). A novel intrusion detection system based on hierarchical clustering and support vector machines. Expert systems with Applications, 38(1), 306–313.

Houmb, S., & Hansen, K. (2003). Towards a UML profile for model-based risk assessment of security critical systems. In Critical systems development with UML (CSDUML 2003) (pp. 95–104).

Howard, M., & Leblanc, D. (2001). Writing secure code. Redmond, WA, USA: Microsoft Press.

Howard, R. A. (1971). Dynamic probabilistic systems, volume i: Markov models. John Wiley & Sons.

Howard, R. A., & Matheson, J. E. (2005). Influence diagrams. Decis Anal., 2(3), 127–143.

Hwang, J., Xie, T., El Kateb, D., Mouelhi, T., & Le Traon, Y. (2012). Selection of regression system tests for security policy evolution. In Proceedings of the 27th IEEE/ACM international conference on automated software engineering (pp. 266–269).

IEC 60300-9 Dependability management - Part 3: Application guide - Section 9: Risk analysis of technological systems - Event Tree Analysis (ETA) [Computer software manual]. (1995).

IEC 61025 Fault Tree Analysis (FTA) [Computer software manual]. (1990).

IEC 61165 application of Markov techniques [Computer software manual]. (1995).

IEEE. (1990). Standard Glossary of Software Engineering Terminology. Author.

ISO 31000 – risk management – Principles and guidelines [Computer software manual]. (2009).

ISO/IEC 27001 – Information technology – Security techniques – Information security management systems – Requirements [Computer software manual]. (2005).

ISO/IEC 27005 – Information technology – Security techniques – Information security risk management [Computer software manual]. (2011).

ISO/IEC/IEEE 42010 – systems and software engineering – architecture description [Computer software manual]. (2011).

Jackson, M. (2001). Problem frames: Analyzing and structuring software development problems. Addison Wesley.

Jacobs, B., Leino, K. R. M., Piessens, F., & Schulte, W. (2005). Safe concurrency for aggregate objects with invariants. In Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (p. 137-146). IEEE Computer Society.

Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., & Piessens, F. (2011). Verifast: A powerful, sound, predictable, fast verifier for c and java. NASA Formal Methods, 41–55.

Jacobs, B., Smans, J., & Piessens, F. (2010). A quick tour of the verifast program verifier. In Aplas (p. 304-311).

Jin, H., Sun, J., Chen, H., & Han, Z. (2004). A fuzzy data mining based intrusion detection model. In Distributed computing systems, 2004. FTDFC 2004. Proceedings. 10th IEEE international workshop on future trends of (pp. 191–197).

Johns, M. (2008). On JavaScript Malware and related threats - Web page based attacks revisited. Journal in Computer Virology, 4(3), 161-178.

Julisch, K. (2003). Clustering intrusion detection alarms to support root cause analysis. ACM Transactions on Information and System Security (TISSEC), 6(4), 471. Jürjens, J. (2005). Secure systems development with UML. Springer Verlag.

Jürjens, J., Marchal, L., Ochoa, M., & Schmidt, H. (2011). Incremental Security Verification for Evolving UMLsec models. In Proceedings of the 7th European conference on modelling foundations and applications (ECMFA) (p. 52-68). Springer.

Kassios, I. T. (2006). Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Fm (p. 268-283).

Katta, V., & Stålhane, T. (2011). A conceptual model of traceability for safety systems. Poster session at 2nd International Conference on Complex Systems Design & Management (CSD&M'11).

Khan, S., Greenwood, P., Garcia, A., & Rashid, A. (2008). On the Impact of Evolving Requirements-Architecture Dependencies: An Exploratory Study. In advanced information systems engineering: 20th international conference, CAISE 2008 Montpellier, France, June 18-20, 2008, Proceedings (p. 243).

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An Overview of AspectJ. In J. L. Knudsen (Ed.), 15th European conference on object-oriented programming (ecoop) (pp. 327–355). Springer-Verlag.

Kim, D.-K., Ray, I., France, R., & Li, N. (2004). Modeling role-based access control using parameterized UML models. In M. Wermelinger & T. Margaria (Eds.), Fundamental approaches to software engineering (FASE 2000) (pp. 180–193). Springer.

Kissel, R., Stine, K., Scholl, M., Rossman, H., Fahlsing, J., & Gulick, J. (2008). Security considerations in the system development lifecycle [Computer software manual]. (NIST Special Publication 800-64 Revision 2)

Klusener, A. S., Lämmel, R., & Verhoef, C. (2005). Architectural modifications to deployed software. Science of Computer Programming, 54(2), 143–211.

Koch, M., Mancini, L. V., & Parisi-Presicce, F. (2001). On the specification and evolution of access control policies. In Proceedings of the sixth ACM symposium on access control models and technologies (pp. 121–130). New York, NY, USA: ACM.

Kongsli, V. (2006). Towards agile security in web applications. In Companion to the 21st ACM sigplan symposium on object-oriented programming systems, languages, and applications (pp. 805–808). New York, NY, USA: ACM.

Krautsevich, L., Lazouski, A., Martinelli, F., & Yautsiukhin, A. (2010). Risk-aware usage decision making in highly dynamic systems. In Proceedings of the fifth international conference on internet monitoring and protection (icimp'10). IEEE Computer Society.

Krautsevich, L., Lazouski, A., Martinelli, F., & Yautsiukhin, A. (2011). Cost-effective enforcement of UCONApolicies. In Proceedings of the 6th international conference on risks and security of internet and systems (crisis'11) (pp. 1–8). IEEE Computer Press.

Kruegel, C., Mutz, D., Robertson, W., & Valeur, F. (2003 Bayesian event classification for intrusion detection. In Computer security applications conference, 2003. Proceedings. 19th annual (p. 14-23). Published by the IEEE Computer Society

Kruegel, C., & Vigna, G. (2003). Anomaly detection of web-based attacks. In Proceedings of the 10th ACM conference on computer and communications security (pp. 251–261).

Lam, W., & Loomes, M. (1998). Requirements evolution in the midst of environmental change: a managed approach. In CSMR '98.

Landwehr, C., Bull, A., McDermott, J., & Choi, W. (1994). A taxonomy of computer program security flaws, with examples. ACM Computing Surveys, 26(3), 211–255. Leangsuksun, C., Song, H., & Shen, L. (2003). Reliability modeling using UML. Software Engineering Research and Practice, 2003, 259–262.

Lee, W., & Xiang, D. (2001). Information-theoretic measures for anomaly detection. In Security and privacy, 2001. S&P 2001. Proceedings. 2001 IEEE symposium on (pp. 130–143).

Lehman, M. (1980). On understanding laws, evolution, and conservation in the large-program lifecycle. Journal of Systems and Software, 1, 213–221.

Lehman, M. (1998). Software's future: Managing evolution. IEEE Software, 15(1), 40–44.

Leitner, P., Michlmayr, A., Rosenberg, F., & Dustdar, S. (2010). Monitoring, Prediction and Prevention of SLA Violations in Composite Services. In IEEE international conference on web services (ICWS 2010) (pp. 369–376).

Letier, E., & van Lamsweerde, A. (2004). Reasoning about partial goal satisfaction for requirements and design engineering. In Proceedings of the 12th ACM sigsoft twelfth international symposium on foundations of software engineering (pp. 53–62). New York, NY, USA: ACM.

Leung, H., & White, L. (1989). Insights into regression testing (software testing). In Proceedings conference on software maintenance 1989 (pp. 60–69). IEEE Comput. Soc. Press.

Leung, K., & Leckie, C. (2005). Unsupervised anomaly detection in network intrusion detection using clusters. In Proceedings of the twenty-eighth australasian conference on computer science-volume 38 (pp. 333–342).

Li, T., Liu, L., & Bryant, B. R. (2010). Service Security Analysis Based on i*: An Approach from the Attacker Viewpoint. In Security, trust, and privacy for software applications (STPSA 2010) (pp. 127–133). Seoul

Ligaarden, O. S., Lund, M. S., Refsdal, A., Seehusen, F., & Stølen, K. (2011). An architectural pattern for enterprise level monitoring tools. In Maintenance and evolution of service-oriented and cloud-based systems (mesoca'11) (pp. 1–10). IEEE Computer Society.

Ligaarden, O. S., Refsdal, A., & Stølen, K. (2012a). Designing indicators to monitor the fulfillment of business objectives with particular focus on quality and ICT-supported monitoring of indicators. International Journal On Advances in Intelligent Systems, 5(1-2).

Ligaarden, O. S., Refsdal, A., & Stølen, K. (2012b). It security governance innovations: Theory and research. In (pp. 256–292). IGI Global.

Ligatti, J., Bauer, L., & Walker, D. (2005). Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security, 4(1–2), 2–16.

Lin, L., Nuseibeh, B., Ince, D., & Jackson, M. (2004). Using abuse frames to bound the scope of security problems. In 12th IEEE international requirements engineering conference (re'04) (pp. 354–355). IEEE Computer Society.

Lin, L., Prowell, S. J., & Poore, J. H. (2009). The impact of requirements changes on specifications and state machines. SP&E, 39(6), 573–610.

Liu, L., Yu, E., & Mylopoulos, J. (2003). Security and privacy requirements analysis within a social setting. RE 2003, 3, 151–161.

Luckham, D. (2008). The power of events: an introduction to complex event processing in distributed enterprise systems. Rule Representation, Interchange and Reasoning on the Web, 3–3.

Lund, M. S., Solhaug, B., & Stølen, K. (2010). Evolution in relation to risk and trust management. Computer, 43(5), 49–55.

Lund, M. S., Solhaug, B., & Stølen, K. (2011a). Model-driven risk analysis – the coras approach. Springer.

Lund, M. S., Solhaug, B., & Stølen, K. (2011b). Risk analysis of changing and evolving systems using CORAS. In Foundations of security analysis and design vi (fosad vi) (pp. 231–274). Springer.

Lunt, T. (1993). A survey of intrusion detection techniques. Computers & Security, 12(4), 405–418.

Manadhata, P., & Wing, J. (2011). An attack surface metric. Software Engineering, IEEE Transactions on, 37(3), 371–386.

Mannan, S. (Ed.). (2005). Lees' loss prevention in the process industries. Hazard identification, assessment and control. Volume 1 (3rd ed.). Elsevier.

Mantel, H. (2002). On the composition of secure systems. In IEEE symposium on security and privacy (p. 88-101). IEEE.

Massacci, F., Mylopoulos, J., Paci, F., Tun, T. T., & Yu, Y. (2011). An extended ontology for security requirements. In CAISE workshops (p. 622-636).

Massacci, F., Mylopoulos, J., & Zannone, N. (2010). Security Requirements Engineering: The SI * Modeling Language and the Secure Tropos Methodology. In Z. Ras & L.-S. Tsay (Eds.), Advances in intelligent information systems (Vol. 265, pp. 147–174). Springer.

McVeigh, A. (2009). A rigorous, architectural approach to extensible applications (Unpublished doctoral dissertation). Imperial College London.

Mead, N. R., & Stehney, T. (2005). Security quality requirements engineering (square) methodology. SIGSOFT Software Engineering Notes, 30(4), 1–7.

Mehta, D. M. (2007). Effective software security management. OWASP. Retrieved from https:// www.owasp.org/images/2/28/Effective _Software_Security_Management.pdf

Mellado, D., Fernández-Medina, E., & Piattini, M. (2008). Towards security requirements management for software product lines: A security domain requirements engineering process. Computer Standards & Interfaces, 30(6), 361-371.

Mens, T., & Demeyer, S. (2008). Software evolution. Springer.

Mens, T., Magee, J., & Rumpe, B. (2010). Evolving software architecture descriptions of critical systems. Computer, 43(5), 42–48. MITRE. (n.d.). CWE - Common Weakness Enumeration. http://cwe.mitre.org/.

Moebius, N., Stenzel, K., Grandy, H., & Reif, W. (2009). Securemdd: A model-driven development method for secure smart card applications. In Availability, reliability and security, 2009. ARES'09. International conference on (pp. 841–846).

Montrieux, L., Wermelinger, M., & Yu, Y. (2011). Challenges in model-based evolution and merging of access control policies. In Proceedings of the 12th international workshop on principles of software evolution and the 7th annual ercim workshop on software evolution, EVOL/IWPSE 2011, szeged, hungary (p. 116-120).

Mouratidis, H., Giorgini, P., & Manson, G. (2003). Integrating security and systems engineering: Towards the modelling of secure information systems. In Proc. of CAISE'2003 (p. 1031-1031). Springer.

Mouratidis, H., & Jurjens, J. (2010). From goal-driven security requirements engineering to secure design. International Journal of Intelligent Systems, 25(8), 813– 840.

Mukkamala, S., Janoski, G., & Sung, A. (2002). Intrusion detection using neural networks and support vector machines. In Proceedings of the 2002 international joint conference on neural networks (Vol. 2, pp. 1702-1707)

Mulo, E., Zdun, U., & Dustdar, S. (2009). Monitoring web service event trails for business compliance. In Service-oriented computing and applications (SOCA), 2009 IEEE international conference on (pp. 1–8).

Myers, A. C. (1999). Jflow: practical mostly-static information flow control. In Proceedings of the 26th ACM sigplan-sigact symposium on principles of programming languages (pp. 228–241). New York, NY, USA: ACM.

Nhlabatsi, A., Bandara, A., Shinpei, H., Jurjens, J., Kaiya, H., Kubo, A,Yu, Y. (2010). Security patterns: Comparing modeling approaches. In Software engineering for secure systems: Industrial and research perspectives (pp. 75–111). IGI Global.

Nhlabatsi, A., Nuseibeh, B., & Yu, Y. (2009). Security requirements engineering for evolving software systems: A survey. Journal of Secure Software Engineering, 1, 54-73.

Ning, P., Cui, Y., & Reeves, D. (2002). Constructing attack scenarios through correlation of intrusion alerts. In Proceedings of the 9th ACM conference on computer and communications security (pp. 245–254).

Noel, S., & Jajodia, S. (2004). Managing attack graph complexity through visual hierarchical aggregation. In Proceedings of the 2004 ACM workshop on visualization and data mining for computer security (pp. 109–118).

Ochoa, M., Jürjens, J., & Cuéllar, J. (2012). Non-interference on UML Statecharts. In 50th international conference on objects, models, components, patterns (TOOLS Europe 2012). Springer.

Ochoa, M., Jürjens, J., & Warzecha, D. (2012). A sound decision procedure for the compositionality of secrecy. In Proc. of the 4th international symposium on engineering secure software and systems, ESSOS 2012 (p. 97-105). Springer.

Oldmeadow, J., Ravinutala, S., & Leckie, C. (2004). Adaptive clustering for network intrusion detection. Advances in Knowledge Discovery and Data Mining, 255–259.

Ou, X., Govindavajhala, S., & Appel, A. (2005). Mulval: A logic-based network security analyzer. In 14th usenix security symposium (pp. 1–16).

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communication ACM, 15(12), 1053–1058.

Paul, S., & Delande, O. (2011). Integrability of design modelling solution. SecureChange FP7 project deliverable D4.4b.

Peltier, T. R. (2010). Information security risk analysis (3rd ed.). Auerbach Publications.

Penninckx, W., Mühlberg, J. T., Smans, J., Jacobs, B., & Piessens, F. (2012). Sound formal verification of linux's usb bp keyboard driver. In Nasa formal methods (p. 210-215).

Phillips, C., & Swiler, L. (1998). A graph-based system for network-vulnerability analysis. In Proceedings of the 1998 workshop on new security paradigms (pp. 71– 79).

Phua, C., Lee, V., Smith, K., & Gayler, R. (2005). A Comprehensive Survey of Data Mining-based Fraud Detection Research. Artificial Intelligence Review, 1–14.

Pierce, B. C. (2002). Types and programming languages. Cambridge, MA, USA: MIT Press.

Pottier, F., & Simonet, V. (2003). Information flow inference for ml. ACM Trans. Program. Lang. Syst., 25(1), 117– 158.

Project PROTEUS. (June 1996). Deliverable 1.3: Meeting the challenge of changing requirements (Tech. Rep.). Centre for Software Reliability, University of Newcastle upon Tyne.

Quinlan, J. R. (1986). Induction of decision trees. Machine learning, 1(1), 81–106.

Quinlan, J. R. (1996). Bagging, boosting, and c4. 5. In Aaai/iaai, vol. 1 (pp. 725–730).

Ray, I., France, R., Li, N., & Georg, G. (2003). An aspect-based approach to modeling access control concerns. Information & Software Technology. (To be published)

Razavian, M., & Lago, P. (2012). A viewpoint for dealing with change in migration to services. In Joint conference on software architecture & European conference on software architecture (WICSA/ECSA).

Refsdal, A., & Stølen, K. (2009). Employing key indicators to provide a dynamic risk picture with a notion of confidence. In Trust management iii (Vol. 300, p. 215-233). Springer.

Ren, J., & Taylor, R. (2005). A secure software architecture description language. In Workshop on software security assurance tools, techniques, and metrics.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In Proceedings of the 17th annual IEEE symposium on logic in computer science (pp. 55–74). Washington, DC, USA: IEEE Computer Society.

Robinson, R. M. (2007). Risk and reliability – an introductory text (7th ed.). Risk and Reliability Associates Pty Ltd.

Roschke, S., Cheng, F., & Meinel, C. (2011). A new alert correlation algorithm based on attack graph. Computational Intelligence in Security for Information Systems, 58–67.

Rothermel, G., Harrold, M. J., Graves, T. L., Kim, J.-M., & Porter, A. (2001). An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology, 10, 184-208.

Rozanski, N., & Woods, E. (2005). Software systems architecture: Working with stakeholders using viewpoints and perspectives. Addison-Wesley Professional.

Ruhroth, T., & Jürjens., J. (2012). Supporting security assurance in the context of evolution: Modular modeling and analysis with UMLSEC. In 16th IEEE international symposium on high assurance systems engineering (HASE 2012). IEEE.

Russo, A., Nuseibeh, B., & Kramer, J. (1999). Restructuring requirements specifications. In IEEE Proceedings: Software (Vol. 146, pp. 44 – 53).

Sabelfeld, A., & Myers, A. C. (2003). Language-based information-flow security. IEEE J. Selected Areas in Communications, 21(1), 5–19.

Salehi, P., Hamoud-Lhadj, A., Colombo, P., Khendek, F., & Toeroe, M. (2010). A EDOC-based domain specific modeling language for the availability management framework. In Proceedings of the 2010 IEEE 12th international symposium on high-assurance systems engineering (pp. 35–44). Washington, DC, USA: IEEE Computer Society.

Scandariato, R., Buyens, K., & Joosen, W. (2010). Automated detection of least privilege violations in software architectures. Software Architecture, 6285, 150–165.

Schieferdecker, I., Grossmann, J., & Schneider, M. (2012). Model-based security testing. In Proceedings 7th workshop on model-based testing.

Schmidt, D. C. (2006). Model-Driven Engineering. IEEE Computer, 39(2).

Schneider, F., Morrisett, G., & Harper, R. (2001). A language-based approach to security. Informatics – 10 Years Back. 10 Years Ahead, 2000, 86–101.

Schneider, S. (1999). Attack trees: Modeling security threats. Dr. Dobb's J., 24, 21–29.

Schonlau, M., DuMouchel, W., Ju, W., Karr, A., Theus, M. & Vardi, Y. (2001). Computer intrusion: Detecting masquerades. Statistical Science, 58–74.

Schultz, E. (2002). A framework for understanding and predicting insider attacks. Computers & Security, 21(6), 526–531.

SecureChange, W. P (2012). Deliverable 7.4: Results of test campaign on case studies. (SecureChange (EU ICT-FET-231101) [accessed: January 15, 2013])

The security risk management guide [Computer software manual]. (2006).

Seehusen, F., & Solhaug, B. (2012). Tool-supported risk modeling and analysis of evolving critical infrastructures. In Multidisciplinary research and practice for information systems (Vol. 7465, pp. 562–577). Springer.

Siemens. (n.d.). CRAMM – The total information security toolkit. http://www.cramm.com/ [accessed: January 15, 2013].

Sindre, G., & Opdahl, A. L. (2000). Eliciting security requirements by misuse cases. In 37th international conference on technology of object-oriented languages and systems (tools pacific'00) (pp. 120–131). IEEE Computer Society.

Smans, J., Jacobs, B., & Piessens, F. (2006). Static verification of code access security policy compliance of .NET applications. Journal of Object Technology, 5(3).

Smans, J., Jacobs, B., & Piessens, F. (2009). Implicit dynamic frames: Combining dynamic frames and separation logic. In ECOOP (p. 148-172).

Solhaug, B., & Seehusen, F. (2013). Model-driven risk analysis of evolving critical infrastructures. Journal of Ambient Intelligence and Humanized Computing, 1-18.

Sood, A., & Enbody, R. (2013). Targeted cyberattacks: A superset of advanced persistent threats. IEEE Security and Privacy, 11(1), 54–61. Sourcefire, I. (n.d.). Snort. (http://www.snort.org/ [accessed: January 15, 2013])

Souza, V. E. S., Lapouchnian, A., & Mylopoulos, J. (2011). System identification for adaptive software systems: a requirements engineering perspective. In 30th International conference on conceptual modeling (pp. 346–361). Springer.

Stark, G. E., Oman, P., Skillicorn, A., & Ameele, A. (1999). An examination of the effects of requirements changes on software maintenance releases. Journal of Software Maintenance: Research and Practice, 11(5), 293–309.

Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. IBM Systems Journal, 13(2), 115-139.

Sullivan, K. J., Griswold, W. G., Cai, Y., & Hallen, B. (2001). The structure and value of modularity in software design. SIGSOFT Softw. Eng. Notes, 26(5), 99–108.

Szlenk, M., Zalewski, A., & Kijas, S. (2012). Modelling architectural decisions under changing requirements. In Joint conference on software architecture & european conference on software architecture (WICAE/ECSA).

Szyperski, C. A. (1998). Component software - beyond object-oriented programming. Addison-Wesley-Longman.

Tamzalit, D., & Mens, T. (2010). Guiding architectural re-structuring through architectural styles. In R. Sterritt, B. Eames, & J. Sprinkle (Eds.), International conference and workshops

on the engineering of computer-based systems (ECBS 2010) (p. 69-78). IEEE Computer Society.

Taylor, R. N., Medividovic, N., & Dashofy, E. M. (2010). Software architecture foundations, theory, and practice. Wiley.

The Open Web Application Security Project. (2011). Owasp website. http://www.owasp.org. [accessed, January 15, 2013]

Tian-yang, G., Yin-sheng, S., & You-yuan, F. (2010). Research on software security testing. World Academy of Science, Engineering and Technology, 70.

Tierney, B., Aydt, R., Gunter, D., Smith, W., Swany, M., Taylor, V., & Wolski, R. (2002). A grid monitoring architecture. In The global grid forum gwd-gp-16-2.

Tran, L. M. S., & Massacci, F. (2011). Dealing with known unknowns: towards a game-theoretic foundation for software requirement evolution. In Proceedings of the 23rd international conference on advanced information systems engineering (pp. 62–76). Springer.

Trend Micro, I. (n.d.). Ossec documentation. (http://www.ossec.net/ [accessed: January 15, 2013])

Trojer, T., Breu, M., & Löw, S. (2010). Change-driven model evolution for living models. In 3rd workshop model-driven tool and process integration (MDTPI), ECMFA 2010. Paris, France.

Trujillo, J., Soler, E., Fernández-Medina, E., & Piattini, M. (2009). A EDOC 2.0 profile to define security requirements for data warehouses. Computer Standards & Interfaces, 31(5), 969–983.

van Lamsweerde, A., & Letier, E. (2000). Handling obstacles in goal-oriented requirements engineering. IEEE Transactions on Software Engineering, 26(10), 978–1005.

van Dongen, B., De Medeiros, A., Verbeek, H., Weijters, A., & van der Aalst, W. (2005). The prom framework: A new era in process mining tool support. Applications and Theory of Petri Nets 2005, 1105–1116.

van Lamsweerde, A. (2003). From system goals to software architecture. In Formal methods for software architectures (p. 25-43). Springer.

van Lamsweerde, A. (2004). Elaborating security requirements by construction of intentional anti-models. In Software engineering, 2004. ICSE 2004. Proceedings. 26th international conference on (pp. 148–157).

van Lamsweerde, A. (2009). Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley & Sons.

Voirin, J.-L. (2008). Method & tools for constrained system architecting. In 18th annual international symposium of the international council on systems engineering (incose'08) (pp. 775–789). Curran Associates, Inc.

Volpano, D., Irvine, C., & Smith, G. (1996). A sound type system for secure flow analysis. J. Computer. Security (2-3), 167–187.

Walker, D. (2000). A type system for expressive security policies. In Proceedings of the 27th ACM sigplan-sigact symposium on principles of programming languages (pp. 254–267). New York, NY, USA: ACM.

Wang, H., & Wang, C. (2003). Taxonomy of security considerations and software quality. Commun. ACM, 46(6), 75–78.

Weiglhofer, M., Aichernig, B., & Wotawa, F. (2009). Fault-based conformance testing in practice. International Journal of Software and Informatics, 3(2-3), 375–411.

Weiss, M. (2007). Modeling security patterns using NFR analysis. In H. Mouratidis & P. Giorgini (Eds.), Integrating security and software engineering: advances and future visions. IGI Global.

Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Dustdar, S., & Leymann, F. (2009). Monitoring and analyzing influential factors of business process performance. In Enterprise distributed object computing conference (EDOC'09) (pp. 141–150).

Wheeler, D. A. (n.d.). The Flawfinder homepage. http://www.dwheeler.com/flawfinder/ [accessed: January 15, 2013]

Williams, B., & Carver, J. (2010). Characterizing software architecture changes: A systematic review. Information and Software Technology, 52(1), 31–51.

Win, B. D. (2004). Engineering application-level security through aspect-oriented software development (Unpublished doctoral dissertation).

Ye, N., Emran, S., Chen, Q., & Vilbert, S. (2002). Multivariate statistical analysis of audit trails for host-based intrusion detection. IEEE Transactions on Computers, 51(7), 810–820.

Yoo, S., & Harman, M. (2010). Regression testing minimisation, selection and prioritisation: A survey. Software Testing, Verification, and Reliability, 1(1), 121-141

Yoshioka, N., Washizaki, H., & Maruyama, K. (2008). A survey on security patterns. Progress in Informatics, 5(5), 35–47.

Younan, Y., Joosen, W., & Piessens, F. (2012). Runtime countermeasures for code injection attacks against C and C++ programs. ACM Comput. Surv., 44(3), 17.

Yskout, K., Ben David, O.-N., Scandariato, R., & Baudry, B. (2012). Requirements-driven runtime reconfiguration for security. In A. Moschitti & R. Scandariato (Eds.), Eternal systems (Vol. 255, p. 25-33). Springer.

Yskout, K., Scandariato, R., & Joosen, W. (2012a). Change patterns. Software and Systems Modeling, 1-24.

Yskout, K., Scandariato, R., & Joosen, W. (2012b). Does organizing security patterns focus architectural choices? In Software engineering (ICSE), 2012 34th international conference on (pp. 617–627).

Zadeh, L. A. (1965). Fuzzy sets. Information and Control, 8(3), 338–353.

Zargar, S., Takabi, H., & Joshi, J. (2011). Dcdidp: A distributed, collaborative, and data-driven intrusion detection and prevention framework for cloud computing environments. In Collaborative computing: Networking, applications and worksharing (collaboratecom), 2011 7th international conference on (pp. 332–341).

Zowghi, D., & Offen, R. (1997). A logical framework for modeling and reasoning about the evolution of requirements. In 3rd IEEE international symposium on requirements engineering (pp. 247–257). IEEE.