Back to the Drawing Board
Bringing Security Constraints in an Architecture-centric Software Development Process

Stefanie Jasser[1], Katja Tuma[2], Riccardo Scandariato[2] and Matthias Riebisch[1]
[1]*Department of Informatics, Universität Hamburg, Hamburg, Germany*
[2]*Software Engineering Division, Chalmers and Gothenburg University, Göteborg, Sweden*

Keywords: Software Architecture, Security by Design, Privacy by Design, Secure Software Architecture, Architecture Erosion, Architectural Decay, Architecture Violations, Architecture Conformance Checking, Architecture Compliance Checking, Security Constraints.

Abstract: Today, security is still poorly considered in early phases of software engineering. Architects and software engineers still lack knowledge about architectural security design as well as implementing it compliantly. However, a software system that is not designed for security or does not adhere to this design can hardly meet its security requirements. In this paper, we present an approach we are working on. The approach consists of two parts: Firstly, we improve the architecture's security level through model transformation. Secondly, we derive rules and constraints from the secured architecture in order to check the implementation's conformance. Through these activities we aim to support architects and software developers in building a secure software system. We plan to evaluate our approach in industrial case studies.

1 INTRODUCTION

Designing and implementing a secure software system is a highly complex task that can hardly be done by a non-security expert today. However, it is essential to consider security as early as possible.

Software architects as well as software developers usually are no security experts. Hence, designing secure architectures and implementing a system in a secure way in compliance with architectural rules and constraints is a challenging task to most software engineers.

While security solutions such as patterns or tactics are thought of as encapsulated security expert knowledge, applying them correctly is still difficult. Supposing a solution has been applied correctly, several difficulties arrive afterwards: First, there are uncertainties of the solutions' interplay and how they impact each other. Second, the security related constraints or rules specified by the software architecture often remain ambiguous.

The first issue is related to the design phase: It is a complex task to define an architecture that is secure as a whole. I. e. a software architecture that uses a number of security solutions (e. g. patterns or tactics). These solutions aim at solving different problems and have to interact in such a manner that the system as a whole is secure.

The second issue is related to both, the design and the implementation phase: the defined architecture actually describes the architects intention. Hence, it could be named as ”intended architecture” or ”should-be architecture” (de Silva, 2014). This intended architecture defines rules and constraints that apply for the software system's implementation: Software developers have to implement the software system in compliance with these rules and constraints in order to avoid vulnerabilities. However, it is likely that software developers introduce violations of these constraints while implementing a system. The reason for these violations is commonly (Knodel and Naab, 2016; de Silva and Balasubramaniam, 2012) attributed to the lack of knowledge about or acceptance of the software's architecture, time pressure during development and out-dated or inadequate documentation.

To keep it's value, the intended architecture has to be implemented correctly, i. e. compliantly. Otherwise, reasoning about fundamental properties of the system is based on false assumptions about the system's actual architecture. Compliance checking approaches are used to ensure the system's adherence to

Jasser, S., Tuma, K., Scandariato, R. and Riebisch, M.
Back to the Drawing Board - Bringing Security Constraints in an Architecture-centric Software Development Process.
DOI: 10.5220/0006659904380446
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 438-446
ISBN: 978-989-758-282-0

the rules and constraints. As per (Knodel and Naab, 2016), conformance checking approaches should be automated to a large extend to deal with time and resource constraints.

Both issues – designing a secure architecture and implementing the software system in compliance with that architecture – are hard to solve. In this paper, we will highlight the mismatch between today's (intended) architecture models and architectural rules and constraints or information needed for performing compliance checks, respectively.

Our approach aims to support software architects and developers in performing those tasks by revising an architects' architecture to make it secure, enriching the intended architecture model with information needed to derive architectural rules and constraints and checking the implementation's compliance with it. We also examine the information and artefacts needed to extract the implemented architecture in order to reflect the architectural compliance rules.

Basic Concepts

Basic concepts of this paper are:

Software Security. A software system is called secure if it works correctly even if it is under malicious attack. Software security is achieved by *security engineering*. Security engineering is about constructively developing a secure software system, i. e. considering security concerns from the early phases of the software development process.

Secure Design. An architecture or design is called secure if it does not have any security flaws, i. e. security requirements and design principles have been respected and there are no fundamental vulnerabilities caused at the design level.

Architectural Rules and Constraints. A software architecture constrains the implementation of the system: Those rules and constraints must be respected in order to gain the quality level enabled by the intended software architecture.

Architectural Compliance. If an actually implemented software system is conform with it's intended architecture, it adheres to all rules and constraints defined by this intended architecture. The term "architectural conformance" is used synonymously by some authors (de Silva and Balasubramaniam, 2012).

Architecture Divergences and Violations. Differences between the implemented architecture and the intended architecture are called divergences. They can be differentiated in missing or inadequate adaptions of the intended architecture during software evolution and architecture violations. Architecture violations are divergences that are caused in the implementation, i. e. the intended architecture is correct and up-to-date but not implemented compliantly.

Running Example

Throughout this paper, we use a running example to illustrate our approach:

Our exemplary software system provides functionality for small and medium-sized enterprises. To allow flexibility and customization, the system can be extend by plugins which lead to a diverse code base. As the plugins' sources are mainly unknown they are not trust worthy and, therefore, are assumed malicious.

In order to protect the system while running untrusted code, the architect decides to use a sandbox pattern (naming inspired by Google's chromium project (Google Chromium Project, 2017)). It is applied to ensure that code from untrusted sources can be executed securely: Such code must not access confidential information or make persistent changes illegally. The sandbox pattern follows the secure design principles of least privilege and assumes that the sandboxed code is malicious. Figure 1 illustrates the sandbox pattern's structure.

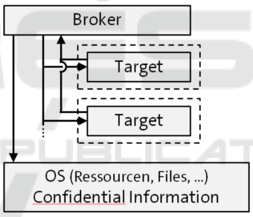

Figure 1: Running example: the sandbox pattern.

There are at least two processes needed: the *broker* is a privileged controller which manages one or more *target* processes:

A target hosts the client-side sandbox infrastructure and the sandboxed code. The infrastructure code includes so called interceptions or hooks. Interceptions are then forwarded to the broker.

The broker manages and controls the target processes. It defines a policy per target process. This policy is used to evaluate interceptions of the targets' action requests. The broker uses an interception manager to forward certain API calls to the broker. If an action is allowed due to the policy, the broker process performs it on the target's behalf.

2 OUR APPROACH

Figure 2 shows the proposed approach towards bringing security constraints in an architecture-centric soft-

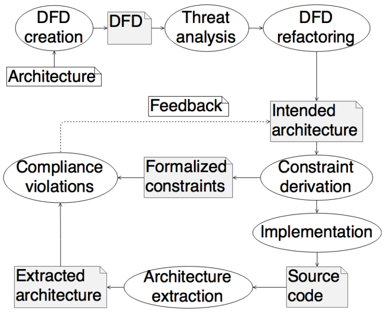

Figure 2: Proposed approach.

ware development process. In the design phase the Data-Flow Diagram (DFD) is commonly included in the creation of several architectural views. The architectural representation is then used to perform a threat analysis of the (as designed) system. By means of pattern-based model refactoring, the intended architecture is obtained. In this work, we refer to the intended architecture as the result of the model refactoring, including the information gained from threat analysis. Given the appropriate transformations, security compliance constraints can be derived from the intended architecture. The proposed approach does not envision the identification of constraint violations in the intended architecture, but rather in the implemented instance of the system. To this aim, the formalized constraints are used to identify compliance violations in the extracted architecture.

2.1 Securing Software Architectures

In this section, we discuss the activities in the design stage of software engineering and focus on how model refactoring can help identifying rules for compliance.

Security-by-design is a practice within organizations which enables planning for security in early phases of the product life-cycle. Designing for security begins with a disciplinary regard to best practices (such as threat analysis), security standards (ISO 15408:2009(E), 2009; ISO 27001:2005(E), 2005; ISO 27000:2009(E), 2009; NIST 800-53, 2013) and derived security policies. However, in practice priorities are commonly shifted due to a lack of resources required for thoroughly addressing security issues at the design phase. As a result, semi-automated approaches aim to partially analyze and secure the architectural design. Several design notations encompass security concepts (van den Berghe et al., 2015), some of which have emerged in the model-driven development domain. Security-design models can be used as a basis for generating systems and their architectures. If models are extended with security semantics, formal signatures can be used to reason about security-related properties of systems.

Our approach bases on a threat analysis of the architect's intended architecture that have to be performed as a first step. We use the analysis results as an indication for securing the software architecture through model transformations:

Software-intensive systems continuously evolve due to constantly changing requirements, generally increasing the system's complexity. In the domain of model-driven software quality and evolution, models and their transformations have been studied in the context of system maintenance (Ivkovic and Kontogiannis, 2006; Mouheb et al., 2015b; Goulão et al., 2016).

Model transformations have also been studied in the context of model refactoring for security. The term refactoring – originally introduced in the context of object-oriented programming by Opdyke (Opdyke, 1992) – is commonly used to signify source code modification for the purpose of strengthening the code structure: ”the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure” (Fowler and Beck, 1999). In the MDE domain, refactoring is applied on a model level, eventually leading to source code modification for the purpose of strengthening a certain software quality characteristic of the system, without changing its external behavior (Mens, 2006).

In terms of subsequent architectural compliance checking, it would be beneficial for the software architects to introduce the notion of 'checkpoints' for compliance already in the design phase. To this aim we introduce such checkpoints for compliance by means of architectural refactoring: Enriching the model with this information aids the formalization of compliance rules. Lano et al. (Lano and Kolahdouz-Rahimi, 2014) consider patterns as forms of transformations: "Patterns can in some cases be considered as forms of transformations: the application of a pattern rewrites a software model or system with a problematic structure into a model or system with an improved structure." In the domain of aspect oriented modeling (Mouheb et al., 2015a), previous work has introduced reusable aspect models with a refinement process for specifying security design patterns on several levels of abstraction, based on System of Security design Patterns (SoSPA) (Nguyen, 2015).

Design patterns are commonly defined as a set of problem specification, solution specification and transformation rules (Kim et al., 2017). To this aim, we specify the problem and the solution with a metamodel and define the transformation rules to enable refactoring as described in Section2.3.

2.2 Checking the Implementation's Architectural Compliance

Besides the planned architecture of a software system that is designed by the system's architect, there is the actual architecture. It describes the architecture-level structure and behavior that are actually implemented. Usually, the actual architecture is not totally compliant to the planned architecture, i e. the implementation diverges from the planned architecture to some degree and thereby violates the architectural constraints. Architecture violations are important to consider as they

(a) cause incorrect assumptions on the system's actual architecture. Subsequently, violations lead to a lack of comprehension and further architecture violations.

(b) may cause the system's failure to comply with it's non-functional requirements. In case of security-related architectural rules and constraints, violations may introduce vulnerabilities.

In order to monitor and control architecture violations compliance checks are performed. When these compliance checks are done, the planned architecture is not questioned. It is thought of as ideal and defines the rules or constraints that must be met by the implementation.

However, checking the implementations compliance with these rules is a highly complex and exhausting task. Hence, automation is needed when applying compliance checking to non-trivial software systems. Our approach will provide tool-support for conducting compliance checks based on the revised security architecture that results from the previous step.

Architectural Rules and Constraints

As a first step in performing architecture compliance checks we need to define the rules and constraints that have to be met. We use the secure design that is the prior step's output an input for this activity. To be valuable, the design needs to specify the constraints and rules that apply for the implementation in a way that

(a) software engineers know which constraints and rules they must respect while implementing the system.

(b) those constraints can be used for automatic or at least tool-supported conformance checking.

Hence, the constraints must be specified comprehensibly, so that software developers and architects can be actively aware of them. However, the specification must be machine-readable, i. e. formal. Assuming that the constraints are explicitly modeled that point is achieved as the revised, secure architecture is used as an input. However, an important task is to formalize the architectural constraints regarding the software systems structure as well as it's behavior.

To determine a set of common architectural security constraints and rules, we derive architectural security constraints from security solutions and best practices, such as patterns, tactics, frameworks and reference architectures.

Extracting the Actual Architecture

There are three components of source code analysis: the parser, the internal representation and the analysis of this representation (Binkley, 2007). The classic examples of internal source code representations are *control-flow graph* (CFG), *call graph*, *abstract syntax tree* (AST) and *value dependence graph* (VDG). Other representations include trace files, dynamic call graphs, XML like files, other executable files, etc. Code analysis techniques are typically categorized into static and dynamic analysis techniques. **Static** code analysis techniques perform the analysis 'off-line', e. g. based on a model instance or source code, without taking the program input into account. For instance, static techniques for information-flow control mainly assign security labels to input data, variables and APIs (Dennings' approach (Denning and Denning, 1977)), making it possible to separate public and private computation. Hence, static analysis can be performed on not executable artefacts such as an intended architecture model or informal documents. Static analysis is applicable to all executions of the program, which inevitably forces approximations. Consequently, such techniques suffer from a high false-positive rate rejecting many secure programs. Other static code analysis techniques include program logics (Banerjee et al., 2008), model checking (García-Ferreira et al., 2014; Marinescu et al., 2014; Jensen et al., 2015; Zuliani et al., 2013) and theorem proving (Bernasconi et al., 2017; Darvas et al., 2005; Guo et al., 2016). **Dynamic** code analysis techniques take program input into account (typically a single input). They use runtime program information to perform the analysis. This allows greater precision for the particular input, but causes lower result correctness guarantees for other program inputs.

In order to perform meaningful compliance checks on the implemented architecture, our analysis technique will include both dynamic and static elements: *Static analysis* techniques are used to extract information on information and control flows, error handling, the access control policy and the software system's structure by identifying code that runs more concisely. We plan to apply dynamic analysis to find information on the software system's memory management as well as time-dependent constraints which is mainly security-related behavior. This information may be extracted by data mining approaches such as sequential pattern discovery (Lo et al., 2007). In our running example we could check, if the sandboxed code is exclusively executed within the target.

2.3 Illustration on the Running Example

Assuming a given DFD has been analyzed for threats beforehand, we implemented a transformation to a pattern-based DFD for a preliminary set of possible DFDs. For illustration purposes, we have defined a source meta-model of a regular DFD, a target meta-model of the sandbox pattern and a set of M2M transformation rules using Eclipse Ecore with ATL. When run, the transformation will refactor the input model (conforming to the source meta-model) to an output model (conforming to the target meta-model). Due to space limitations, we only show the first transformation rule below.

```
create OUT : SandboxDFD from IN : DFD;

rule DFD2SandboxDFD {
from s: DFD!DFD
to t: SandboxDFD!SandboxDFD (
outerdfdelements<-s.getOuterElements()
    ->collect (a|thisModule.
    DFDEL2SANDEL(a)),
broker<-s.element->select (el|el.
    oclIsTypeOf(DFD!PN))->select (e|e.
    Type=#BROKER)->collect(a|thisModule
    .PN2Broker(a)),
target<-s.element->select (el|el.
    oclIsTypeOf(DFD!PN))->select (e|e.
    Type=#TARGET)->collect(a|thisModule
    .PN2Target(a)))
}
```

When run on a simple DFD, the transformation imposes restrictions in the design by requiring a separation of broker, target and outer elements in the diagram. Moreover, the broker process is required to have been initialized before the target process. Upon executing the transformation rules for the broker and target, the transformed processes include a reference to the Inter-Process Communication (IPC) service and client, respectively.

In our running example, the following architectural rules can be defined for the sandbox pattern:

- At least two components must be involved in a sandbox pattern's implementation.
- The control flow must neither start within that sandbox (”target”) nor the controller process (”broker”).
- The target must not (directly) access system resources, keyboard or mouse events etc.
- The broker manages all targets' rights by hosting the policy engine.
- All control and data flow enters and leaves the target via the broker component. I. e. the broker intercepts and handles all targets actions.
- All code to be sandboxed is executed only and solely within the target.
- There may be separate processes for the broker and each target

However, not all of these constraints can be derived from today's design models: for instance, time-dependent constrains can hardly be derived from common design models which are mostly DFD or UML models.

An architect could define additional constraints, e. g. regarding naming conventions that apply for the actual project.

The constraints mentioned above can directly be derived from the applied sandbox pattern in the software's architecture. However, there are further constraints such as constraints regarding secure error handling: assuming the broker finds that an action requested by a target is not allowed due to the policy. The broker now may inform the target on this failed execution. Such error messages must not reveal information on the system beyond the broker or the broker's implementation. To ensure this, the error types and messages should be custom but not default.

2.4 Formalizing the Architecture

As we want to provide tool support for the conformance checking as well as for the security architecture improvement we need to formalize the software architecture and it's constraints. This is done rarely today (Caracciolo, 2016), for instance it is done in (Pene et al., 2017). The formalization approach must allow to describe all information needed for the security analysis, structural as well as behavioral constraints. That includes the expression of temporal characteristics. Due to these and further requirements like a good documentation and community, we have limited the eligible ADLs to Alloy (Jackson, 2011) and π-ADL (Oquendo, 2008).

In our approach, we use Alloy to describe the rules and constraints of a secure software architecture. It is a description language for expression structural and behavioral constraints which allows to explicitly model time-dependencies. There is tool-support available for analyzing and visualizing Alloy models to identify over- and underspecification.

As modeling a complete architecture including all information needed for security compliance checking in Alloy is a rather complex task, we plan develop an export and import mechanism to common UML and DFD tools in order to ease it's usage. These tools will then by extended for our approach, i. e. for providing the security revision functionality, extending the models by information needed for deriving architectural constraints from that architecture and specifying these constraints.

2.5 Evaluation Plan

While we present work in progress in this paper, we want to outline how we intend to show our approach's feasibility:

First, we will apply our approach to two industrial case studies of different domains. One project is a distributed system in the context of energy network controls, i. e. critical infrastructures with high demands on it's security and safety level. The other project is on a software that automatically archives enterprise documents by analyzing their content. Hence, this software greatly deals with sensitive enterprise data and requires a high security and privacy level. For both projects reasonable documentation and an intended architecture are available.

If needed, we will perform additional case studies using open source projects with security requirements. This could for instance be the Google Chromium Project or the Apache WildFly Project.

3 RELATED WORK

We address the colliding body of knowledge on securing software architectures and architectural conformance checking.

Secure Design Automation. Threat analysis is a method for identifying, analyzing and prioritizing threats of early software architectural models. STRIDE is most commonly used to this aim. This method's systematicity and repetitiveness results in a so called threat explosion, where too many unimportant threats are identified and analyzed (Scandariato et al., 2015). Apart from existing tools – e. g. (tmt, 2016) –, only few attempts at automating STRIDE were made (Schaad and Borozdin, 2012). Yet, automation is a promising approach to prevent human flaws occurring downstream the software production line (Yskout et al., 2008).

Apart from the software-oriented STRIDE there are other methods used to perform threat analysis: CORAS (Lund et al., 2011), PASTA (UcedaVelez and Morana, 2015), attack trees (Mauw and Oostdijk, 2005), Trike (Saitta et al., 2005), to name a few. J. Berger et al. (Berger et al., 2016) have proposed an approach for a semi-automated architectural risk analysis. They provide a a tool which automatically extracts architectural vulnerabilities based on an extended DFD notation and proposes known mitigations. To create the threat model, formalized rules are build based on information obtained from lowering EDFDs to simple graphs. The authors recognize the need to limit the time spent by security experts and therefore separate the responsibilities to include the expert only when needed. However, the knowledge base rules are used to discover only cataloged vulnerabilities, as opposed to finding all possible threats. Furthermore, Almorsy et al. (Almorsy et al., 2013) have introduced an approach for a security risk analysis by means of formalized signatures of scenarios and metrics. The authors developed a prototype tool, which takes as input a (sub)set of design, architecture and code level artifacts. Using formalized signatures of attack scenarios and security metrics (with OCL), the tool is able to identify signature matches in the incrementally built architectural model.

Closely related to goal-oriented requirements engineering methods, architectural analysis has been studied in the context of automating the generation of attack trees (Li et al., 2016; Birkholz et al., 2010), attack graphs (Sheyner et al., 2002) and attack paths (Chen et al., 2007). Aforementioned approaches aim toward a static architectural analysis, where the implemented solution is not the main focus. For a more complete list of attack and defense modeling methods, we refer the reader to the work of Kordy et al. (Kordy et al., 2014). Complementary to architectural security patterns (Yskout et al., 2012), the so called *problem frames* (Beckers et al., 2013) and *treat patterns* have emerged. Abe et al. (Abe et al., 2013) proposed to model negative scenarios (as defined by the CC standard (ISO 15408:2009(E), 2009)) in a semi-automated way with threat patterns and use them during business process modeling.

Architecture Compliance Checking. A secure design does not automatically result in a secure software system: it must be implemented correctly. Most approaches for identifying violations of architectural constraints concentrate on constraints regarding the static structure of the system, e. g. identifying the allowed and forbidden dependencies between components or layers on the architectural level. Such approaches are usually based on reflexion models introduced in (Murphy et al., 2001), dependency structure matrices or relation conformance rules. This is mainly related to the software system's maintainability, which is commonly addressed in the related work (e. g. (Knodel and Naab, 2016), (Berger et al., 2013)), whereas other quality attributes such as security are poorly understood today.

There have also been a few approaches that dynamically check the conformance at runtime: (Nicolaescu and Lichter, 2016; de Silva, 2014). Yet, none of these works consider security constraints. Other authors use aspect-orientation to dynamically enforce architecture conformance: (Merson, 2007; Ganesan et al., 2008; Wang et al., 2007). Another approach that intends runtime compliance checking is the SECORIA-approach (Abi-Antoun and Barnes, 2010). Abi-Antoun and Barnes define some security-specific constraints which define permitted, and prohibited connections based on security requirements. The authors do not consider changes over time in the system's behavior and structure.

4 CONCLUSION: DISCUSSION AND FUTURE WORK

In this paper we present our work towards an approach for developing secure software systems. The approach focuses on the constructive phases of the development process, i. e. the design phase, the implementation phase and the transition between them.

The first step in securing the design is to analyze the architectural model for potential security threats. This non-trivial task raises questions about threat granularity (when to stop detailing threats), stopping conditions of analysis, quality assurance of analysis results, to name a few. To this aim, our future work will include semi-automated approaches reducing manual intervention and helping the exploration of the problem space in a computerized fashion.

The presented approach extracts compliance constraints from the resulting secure design as a result of threat analysis. Therefore, the soundness of the secured architecture is of utter importance. In order to complement the security requirements refinement during threat analysis, we plan to explore the potential of design flaw detection. We plan to realize this detection by mining documented design flaws by using techniques such as graph pattern matching. We will analyze design flaw and issue catalogs (e. g. OWASP) to find flaws and issues or identification criteria in order to find them in an input architecture model.

Regarding the pattern-based model transformation presented in this approach, we are interested in how to maintain the conflicts introduced by several non-functional concerns. For example, security requirements commonly effect safety requirements. Further, changes in the design model due to the transformations must not obstruct or corrupt solutions which have been applied to the input design models to enable other non-functional quality characteristics.

Besides our future work on secure design transformation, more work has to be done on checking the implementations compliance with the resulting secure design: Architectural design needs to be good to enable sound compliance checking as architectural rules depend on the amount of information provided. I. e. the design model needs to comprehensibly present the constraints that apply to the implementation. An essential challenge related to today's architectural models is their lack of information about security solutions: an adequate notation for architecture models is needed. To preserve compatibility and comprehensibility common notations such as DFD or UML models should be enriched with more security semantics. Architectural constraints will be derived from these enhanced architecture models. To perform the conformance checking, the constraints must be specified in a formal way. To further support software engineers in applying our approach we will provide a catalog of frequently used transformations as a result of the common solutions' analysis (e. g. common flaws, patterns). This catalog will also comprise frequent constraints and rules derived of such security solutions.

As previously mentioned, we will apply a hybrid approach to extract the actual architecture from the system, i. e. we will combine static and dynamic analysis techniques. To enable checking for compliance of the implemented architecture, our techniques need to be able to extract the required information for reflecting compliance rules. For example, information gained from control and data flow analysis, as well as information about behavioral aspect of the system. Further investigations about the required information for extraction, as well as the source of this information in the implemented architecture (e. g. source code, AST, CFG, executable files, etc.) is needed.

Another challenge related to security architecture compliance checking is to display architecture violations clearly and comprehensibly. This is an essential usability requirement of our approach as we need to give adequate feedback on software system's compliance with it's intended architecture.

REFERENCES

(2016). Sustainable Application Security microsofts new threat modeling tool. blog.secodis.com/2016/07/06/microsofts-new-threat-modeling-tool/. Accessed: 2017-11-15.

Abe, T., Hayashi, S., and Saeki, M. (2013). Modeling security threat patterns to derive negative scenarios. In *Software Engineering Conference)*, volume 1, pages 58–66. IEEE.

Abi-Antoun, M. and Barnes, J. M. (2010). Analyzing security architectures. In *25th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–12.

Almorsy, M., Grundy, J., and Ibrahim, A. S. (2013). Automated software architecture security risk analysis using formalized signatures. In *International Conference on Software Engineering*, pages 662–671. IEEE Press.

Banerjee, A., Naumann, D. A., and Rosenberg, S. (2008). Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353. IEEE.

Beckers, K., Hatebur, D., and Heisel, M. (2013). A problem-based threat analysis in compliance with common criteria. In *8th International Conference on Availability, Reliability and Security*, pages 111–120. IEEE.

Berger, B. J., Sohr, K., and Koschke, R. (2013). Extracting and analyzing the implemented security architecture of business applications. In *17th European Conference on Software Maintenance and Reengineering*, pages 285–294. IEEE.

Berger, B. J., Sohr, K., and Koschke, R. (2016). Automatically extracting threats from extended data flow diagrams. In *International Symposium on Engineering Secure Software and Systems*, pages 56–71. Springer.

Bernasconi, A., Menghi, C., Spoletini, P., Zuck, L. D., and Ghezzi, C. (2017). From model checking to a temporal proof for partial models. In *International Conference on Software Engineering and Formal Methods*, pages 54–69. Springer.

Binkley, D. (2007). Source code analysis: A road map. In *Future of Software Engineering*, pages 104–119. IEEE.

Birkholz, H., Edelkamp, S., Junge, F., and Sohr, K. (2010). Efficient automated generation of attack trees from vulnerability databases. In *Working Notes for the 2010 AAAI Workshop on Intelligent Security*, pages 47–55.

Caracciolo, A. (March 2016). *A Unified Approach to Architecture Conformance Checking*. Dissertation, Universität Bern, Bern.

Chen, Y., Boehm, B., and Sheppard, L. (2007). Value driven security threat modeling based on attack path analysis. In *40th Annual Hawaii International Conference on System Sciences*, pages 280a–280a. IEEE.

Darvas, Á., Hähnle, R., and Sands, D. (2005). A theorem proving approach to analysis of secure information flow. In *International Conference on Security in Pervasive Computing*, pages 193–209. Springer.

de Silva, L. (2014). *Towards Controlling Software Architecture Erosion Through Runtime Conformance Monitoring*. Dissertation, University of St. Andrews, St. Andrews.

de Silva, L. and Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Ganesan, D., Keuler, T., and Nishimura, Y. (2008). Architecture compliance checking at runtime: An industry experience report. In *The 8th International Conference on Quality Software*, pages 347–356.

García-Ferreira, I., Laorden, C., Santos, I., and Bringas, P. G. (2014). A survey on static analysis and model checking. In *International Joint Conference SOCO14-CISIS14-ICEUTE14*, pages 443–452. Springer.

Google Chromium Project (2017). Sandbox pattern.

Goulão, M., Amaral, V., and Mernik, M. (2016). Quality in model-driven engineering: a tertiary study. *Software Quality Journal*, 24(3):601–633.

Guo, X., Dutta, R. G., Mishra, P., and Jin, Y. (2016). Scalable soc trust verification using integrated theorem proving and model checking. In *IEEE International Symposium on Hardware Oriented Security and Trust*, pages 124–129. IEEE.

ISO 15408:2009(E) (2009). ISO/IEC common criteria for information security evaluation. Standard, International Organization for Standardization (ISO) and Enternational Elecrotechnical Commission (IEC).

ISO 27000:2009(E) (2009). ISO/IEC Information technology-security techniques-Information security management systems-Overview and Vocabulary. Standard, International Organization for Standardization (ISO) and Enternational Elecrotechnical Commission (IEC).

ISO 27001:2005(E) (2005). ISO/IEC Information technology-security techniques-Information security management systems-Requirements. Standard, International Organization for Standardization (ISO) and Enternational Elecrotechnical Commission (IEC).

Ivkovic, I. and Kontogiannis, K. (2006). A framework for software architecture refactoring using model transformations and semantic annotations. In *10th European Conference on Software Maintenance and Reengineering*, pages 10–pp. IEEE.

Jackson, D. (2011). *Software Abstractions: Logic, Language and Analysis*. MIT Press, Cambridge, 2nd ed. edition.

Jensen, C. S., Møller, A., Raychev, V., Dimitrov, D., and Vechev, M. (2015). Stateless model checking of event-driven applications. In *ACM SIGPLAN Notices*, volume 50, pages 57–73. ACM.

Kim, D.-K., Lu, L., and Lee, B. (2017). Design pattern-based model transformation supported by qvt. *Journal of Systems and Software*, 125:289–308.

Knodel, J. and Naab, M. (2016). *Pragmatic Evaluation of Software Architectures*. The Fraunhofer IESE Series on Software and Systems Engineering. Springer International Publishing, Cham and s.l.

Kordy, B., Piètre-Cambacédès, L., and Schweitzer, P. (2014). Dag-based attack and defense modeling: Dont miss the forest for the attack trees. *Computer science review*, 13:1–38.

Lano, K. and Kolahdouz-Rahimi, S. (2014). Model-transformation design patterns. *IEEE Trans. Software Eng.*, 40(12):1224–1259.

Li, T., Paja, E., Mylopoulos, J., Horkoff, J., and Beckers, K. (2016). Security attack analysis using attack patterns. In *IEEE 10th International Conference on Research Challenges in Information Science*, pages 1–13. IEEE.

Lo, D., Khoo, S.-C., and Liu, C. (2007). Efficient mining of iterative patterns for software specification discovery. In *13th ACM SIGKDD International Conference on Knowledge discovery and data mining*, pages 460–469. ACM.

Lund, M. S., Solhaug, B., and Stølen, K. (2011). A guided tour of the coras method. In *Model-Driven Risk Analysis*, pages 23–43. Springer.

Marinescu, R., Kaijser, H., Mikučionis, M., Seceleanu, C., Lönn, H., and David, A. (2014). Analyzing industrial architectural models by simulation and model-checking. In *International Workshop on Formal Techniques for Safety-Critical Systems*, pages 189–205. Springer.

Mauw, S. and Oostdijk, M. (2005). Foundations of attack trees. In *ICISC*, volume 3935, pages 186–198. Springer.

Mens, T. (2006). On the use of graph transformations for model refactoring. *Lecture Notes in Computer Science*, 4143:219.

Merson, P. (2007). Using aspect-oriented programming to enforce architecture. Technical Report CMU/SEI-2007-TN-019, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Mouheb, D., Debbabi, M., Pourzandi, M., Wang, L., Nouh, M., Ziarati, R., Alhadidi, D., Talhi, C., and Lima, V. (2015a). *Aspect-Oriented Security Hardening of UML Design Models*. Springer.

Mouheb, D., Debbabi, M., Pourzandi, M., Wang, L., Nouh, M., Ziarati, R., Alhadidi, D., Talhi, C., and Lima, V. (2015b). *Model-Driven Architecture and Model Transformations*, pages 35–45. Springer International Publishing, Cham.

Murphy, G. C., Notkin, D., and Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Software Eng.*, 27(4):364–380.

Nguyen, P. H. (2015). Model-driven security based on a unified system of security design patterns. Technical report, University of Luxembourg.

Nicolaescu, A. and Lichter, H. (2016). Behavior-based architecture reconstruction and conformance checking. In Muccini, H. and Harper, K. E., editors, *WICSA 2016*, pages 152–157, Piscataway, NJ. IEEE.

NIST 800-53 (2013). Security and Privacy Controls for Federal Information Systems and Organizations. Standard, National Institute of Standards and Technology (NIST)).

Opdyke, W. F. (1992). Refactoring object-oriented frameworks.

Oquendo, F. (2008). Dynamic software architectures: Formally modelling structure and behaviour with pi-adl. In *3rd International Conference on Software Engineering Advances*, pages 352–359.

Pene, L., Hamza, L., and Adi, K. (2017). Compliance verification algorithm for computer systems security policies. In *International Conference on E-Technologies*, pages 96–115. Springer.

Saitta, P., Larcom, B., and Eddington, M. (2005). Trike v. 1 methodology document [draft]. *dymaxion. org/trike/Trike_v1_Methodology_Documentdraft.pdf*.

Scandariato, R., Wuyts, K., and Joosen, W. (2015). A descriptive study of microsofts threat modeling technique. *Requirements Engineering*, 20(2):163–180.

Schaad, A. and Borozdin, M. (2012). Tam 2: automated threat analysis. In *27th Annual ACM Symposium on Applied Computing*, pages 1103–1108. ACM.

Sheyner, O., Haines, J., Jha, S., Lippmann, R., and Wing, J. M. (2002). Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and privacy*, pages 273–284. IEEE.

UcedaVelez, T. and Morana, M. M. (2015). *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*. John Wiley & Sons.

van den Berghe, A., Scandariato, R., Yskout, K., and Joosen, W. (2015). Design notations for secure software: a systematic literature review. *Software & Systems Modeling*, pages 1–23.

Wang, C.-M., Huang, C.-C., Chen, H.-M., and Wang, S.-T. (2007). Conformance checking of running programs in dynamic aspect-oriented systems. In *14th Asia-Pacific Software Engineering Conference*, pages 183–190, Los Alamitos, Calif. IEEE Computer Society.

Yskout, K., Scandariato, R., De Win, B., and Joosen, W. (2008). Transforming security requirements into architecture. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 1421–1428. IEEE.

Yskout, K., Scandariato, R., and Joosen, W. (2012). Does organizing security patterns focus architectural choices? In *Proceedings of the 34th International Conference on Software Engineering*, pages 617–627. IEEE Press.

Zuliani, P., Platzer, A., and Clarke, E. M. (2013). Bayesian statistical model checking with application to stateflow/simulink verification. *Formal Methods in System Design*, 43(2):338–367.