# Towards Ensuring Security by Design in Cyber-Physical Systems Engineering Processes

Johannes Geismann
Paderborn University
Software Engineering
Paderborn, Germany
johannes.geismann@upb.de

Christopher Gerking
Paderborn University
Software Engineering
Paderborn, Germany
christopher.gerking@upb.de

Eric Bodden*
Paderborn University
Software Engineering
Paderborn, Germany
eric.bodden@upb.de

## ABSTRACT

Engineering cyber-physical systems *secure by design* requires engineers to consider security from the ground up. However, current systems engineering processes are not tailored to cyber-physical systems, or lack an integration with security engineering. In this paper, we integrate secure software engineering practices into an engineering process for cyber-physical systems. Thereby, we enable engineers to specify security requirements at the level of systems engineering, and to take effective countermeasures during both platform-independent and platform-specific software engineering. Our key contribution is the integration of threat models for tracing security requirements to countermeasures. We illustrate our approach by an autonomous car with high security requirements.

## CCS CONCEPTS

• **Software and its engineering** → *Software development process management*; • **Security and privacy** → *Software security engineering*; • **Computer systems organization** → *Embedded and cyber-physical systems*;

## KEYWORDS

cyber-physical systems, security by design, systems engineering

**ACM Reference Format:**
Johannes Geismann, Christopher Gerking, and Eric Bodden. 2018. Towards Ensuring Security by Design in Cyber-Physical Systems Engineering Processes. In *ICSSP '18: International Conference on the Software and Systems Process 2018 (ICSSP '18), May 26–27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3202710.3203159

## 1 INTRODUCTION

*Cyber-physical systems* [6] are interconnected embedded devices that exchange information to coordinate their physical interaction. As such, systems are developed collaboratively by engineers from multiple disciplines. This collaboration has coined the term *systems engineering* and requires a discipline-spanning process.

Furthermore, the interconnected nature of cyber-physical systems turns *security* into a core quality attribute [3, 8]. Due to their interaction with environments and humans, the information processed by a cyber-physical system is critical with respect to security properties like confidentiality, integrity, or availability. Thus, development processes need to address security requirements from the ground up in order to prevent vulnerabilities *by design*, i.e., before the systems are deployed to their dedicated computing platforms.

However, security by design of cyber-physical systems is a challenging problem [19]. In particular, the engineering process must enable security requirements to be properly identified, specified, and traced to effective countermeasures. To this end, security engineering practices need to be integrated during process design and consistently applied by the engineers involved in the process.

Current engineering processes are not fit for the purpose of ensuring security by design of cyber-physical systems. Approaches either lack an integration of security engineering practices [20, 29], or they are not tailored to the characteristics of cyber-physical systems such as the discipline-spanning development [5] and the deployment to dedicated computing platforms [2, 13, 24].

In this paper, we present our research in progress on the integration of secure software engineering practices into a discipline-spanning engineering process for cyber-physical systems [11, 12]. This integration enables engineers to specify security requirements at the level of *model-based systems engineering* [21], and to take countermeasures during both platform-independent and platform-specific software engineering. Our key contribution is the integration of dedicated *threat models* [25] for refining security requirements along the process, and tracing them to both application-level countermeasures (e.g., information flow control) and platform-level countermeasures (e.g., cryptography). Thereby, we enable engineers to ensure security by design of the systems under development.

We illustrate our approach by an autonomous car that must not leak confidential data of passengers to a cloud storage provided by the manufacturer. Furthermore, the car's interface for remote diagnostics must not allow to compromise the integrity of information displayed to passengers. Finally, passengers must never be able to accidentally break the availability of the car's engine control.

In summary, our paper contributes

- a secure-by-design process for cyber-physical systems, and
- an integrated threat modeling approach that provides traceability between security requirements and countermeasures.

**Paper Organization:** We give background information on the engineering of cyber-physical systems in Section 2, and propose our secure-by-design process in Section 3. In Section 4, we discuss related work, before concluding in Section 5.

*Also with Fraunhofer Institute for Mechatronic Systems Design (IEM).


*ICSSP '18, May 26–27, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-6459-1/18/05…$15.00
https://doi.org/10.1145/3202710.3203159

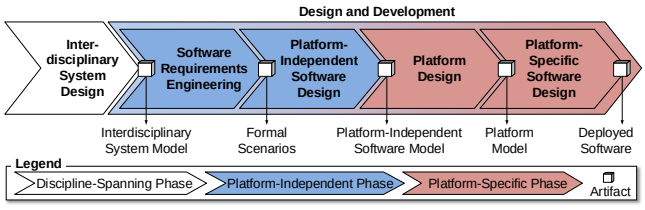



**Figure 1: Cyber-Physical Systems Engineering Process.**

## 2 CYBER-PHYSICAL SYSTEMS ENGINEERING

To account for the interdisciplinary nature, the engineering of cyber-physical systems is guided by the VDI 2206 design methodology [7]. The engineering process consists of (1) an *interdisciplinary system design* phase in which a high-level design is specified in a discipline-spanning fashion, (2) a discipline-specific *design and development* phase in which engineers implement individual parts of the system, and (3) a *system integration* phase that combines the parts from different disciplines into the overall system.

In this paper, we focus on *model-based systems engineering* [21], using models to abstract from the complexity of real systems. In particular, we build on the model-based specification technique Consens [1]. Figure 1 sketches the integration of Consens into the VDI 2206 process [11, 12]. Due to our focus on information security, we restrict ourselves to the discipline of software engineering and the deployment to computing platforms. Accordingly, the system integration with other disciplines is beyond our scope.

In Consens, the *interdisciplinary system design* results in a system model that provides a set of discipline-spanning views on the system under development [1]. In the remainder of this paper, we restrict ourselves to security-relevant views, one of which is an *environment model* that describes interdependencies between the system and its environment. Interdependencies are categorized as information flow, material flow, or energy flow. For example, the environment model in Figure 2a includes information flows between the car and its passengers, the cloud, and remote diagnostics, whereas an energy flow represents the driving force. Another relevant view are *use cases*, describing desired application scenarios. Furthermore, *requirements* that the system must meet to enable certain use cases are specified as another view in tabular form.

The interdisciplinary system model serves as the starting point for the *design and development* (cf. Figure 1). In the *software requirements engineering*, requirements engineers refine use cases to formal scenarios describing how the system communicates with its environment [12]. This *coordination behavior* is driven by software which is designed by software engineers during the *platform-independent software design*. The result is a component-based software model [11] that realizes the coordination behavior by passing messages over ports. In Figure 2b, we depict a component model for the autonomous car with ports to the cloud storage, the remote diagnostics, and passengers. The component contains two subcomponents for the user interface and the engine control. As suggested by Figure 1, the component model is still independent of a computing platform, which is described by platform engineers during the *platform design*. Finally, deployment engineers map the software to the targeted platform during the *platform-specific software design*.

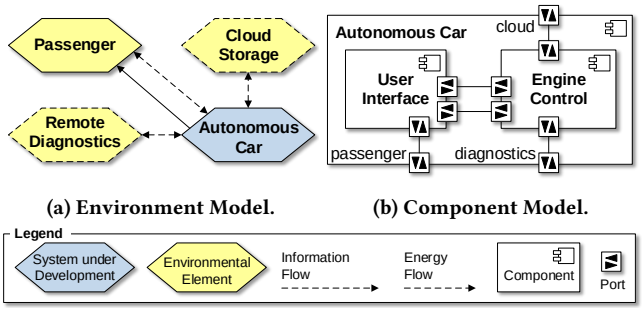



**Figure 2: Autonomous Car Models.**

## 3 SECURE-BY-DESIGN PROCESS

In this section, we propose an integration of secure software engineering practices into the process introduced in Section 2. We describe our extensions to the interdisciplinary system design in Section 3.1, to the platform-independent software engineering in Section 3.2, and to the platform-specific software engineering in Section 3.3.

### 3.1 Interdisciplinary System Design

In Figure 3, we give an overview on the interdisciplinary system design phase [1]. Initially, during the *Analyze Environment* phase, systems engineers specify the environment model introduced in Section 2. We extend this phase with a step *Analyze Threats* to support the identification of valuable assets and potential attack vectors of the system under development. On the basis of this threat analysis, we propose to derive an *information flow policy* [15] that extends the environment model with illegitimate information flows that the system must avoid. For example, the remote diagnostics could potentially be exploited as an attack vector to compromise the information that the car displays to its passengers. Thus, a flow of information from the remote diagnostics to passengers is illegitimate. In this paper, we leave open the concrete methodology used for threat analysis, but refer to general approaches such as STRIDE [25] which provide a practical mnemonic for the identification of common threats.

In phase *Define Use Cases* (cf. Figure 3), systems engineers define the desired use cases of the system. We extend this phase by a definition of *misuse cases* [26], describing insecure behavior that is either maliciously exploited by attackers or accidentally triggered through incorrect use. Here, previously identified threats are refined into step-by-step scenarios. For example, a passenger could accidentally disable the engine control by misusing the user interface.

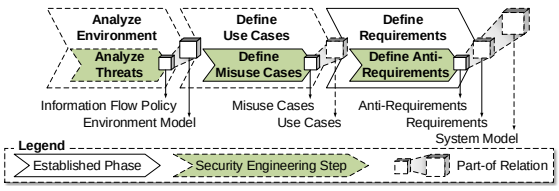
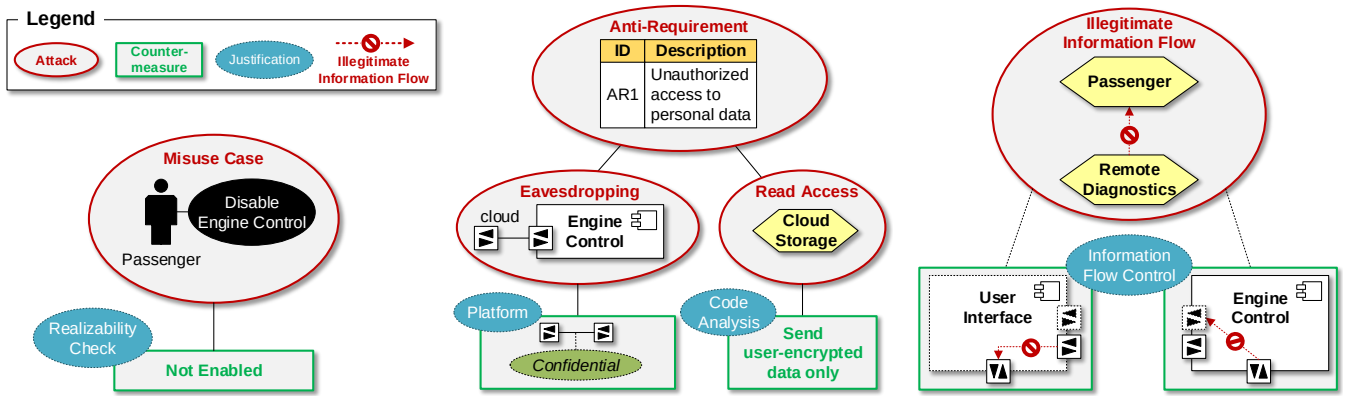



**Figure 3: Interdisciplinary System Design.**




(a) Misuse Case. (b) Anti-Requirement. (c) Information Flow Policy.

**Figure 4: Exemplary Attack-Defense Graphs.**

Next, we extend the phase *Define Requirements* (cf. Figure 3) with a step *Define Anti-Requirements* [4], representing malicious requirements that the system must avoid to mitigate threats. Referring to our example, personal data from passengers must not be stored in the manufacturer's cloud. Hence, systems engineers could specify the anti-requirement *"Unauthorized access to personal data"*.

In summary, our process extends the interdisciplinary system model with a security policy expressed at different abstraction levels, including misuse cases, anti-requirements, and an information flow policy. In the following, we represent these artifacts inside a threat model (called *application threat model*) that allows engineers to trace the security policy to countermeasures taken during the *design and development* phase. To this end, we propose to use *attack-defense graphs* [14] as threat model because they provide a structured way for stepwise refinement of threats along with the ongoing process. An attack-defense graph refers to an *attack node* as its root, describing a possible attack by referring to an element of the security policy. For example, the root nodes of the attack-defense graphs in Figure 4 refer to the misuse case *Disable Engine Control* (Figure 4a), the anti-requirement of unauthorized access to personal data (Figure 4b), and an illegitimate information flow from the remote diagnostics to passengers (Figure 4c).

## 3.2 Platform-Independent Software Engineering

In our process, we propose to refine the attack-defense graphs along with the platform-independent software engineering. To keep track of the security policy, our graphs introduce dedicated *countermeasure nodes* that are used as child nodes to represent countermeasures against the attacks in their parent nodes (cf. Figure 4). Each countermeasure has to be justified to ensure that it has really been implemented. This *justification* could be the result of a formal analysis, or created manually by an engineer (e.g., by signing that a specific behavior is guaranteed). In Figure 4, we depict the integration of such *justifications* into the attack-defense graphs. In the following, we describe the refinement of the attack-defense graphs along with the software requirements engineering in Section 3.2.1, and the platform-independent software design in Section 3.2.2.

*3.2.1 Software Requirements Engineering.* The formal scenarios resulting from this phase enable requirements engineers to analyse the consistency of the specified coordination behavior [9]. In this paper, we propose to extend the same approach towards undesired behavior by formalizing misuse cases as forbidden scenarios. Thereby, requirements engineers can identify inconsistencies like use cases which enable specific misuse cases. By identifying and eliminating such flaws, engineers ensure the *realizability* [9] of the specified coordination behavior. For example, in Figure 4a, we illustrate the refinement of the misuse case *Disable Engine Control*. A countermeasure node describes that the misuse case is never enabled by the coordination behavior, which is justified by means of an automatic realizability check.

*3.2.2 Platform-Independent Software Design.* As depicted in Figure 5, software engineers start this phase by deriving a software component model from the system model [11]. During this *Derive Component Model* phase, we propose to add a step *Refine Security Policy* in which the attack-defense graphs are refined along with the component model. For example, in Figure 4b, we refine our anti-requirement by two attack nodes, describing that personal data may be accessed either by eavesdropping the communication between engine control and cloud, or by read access to the cloud. As another example for the refinement of the security policy, the information flow policy is decomposed into local flow policies restricting individual components such that their message passing behavior over certain ports must not depend on other ports. For example, in Figure 4c, we decompose the illegitimate information flow into two local flow policies for the user interface and the engine control.

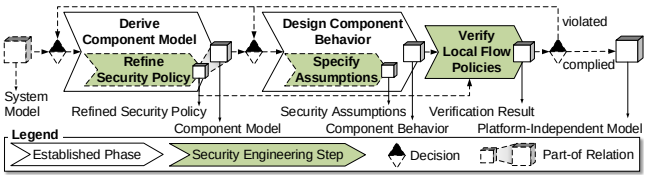



**Figure 5: Platform-Independent Software Design.**

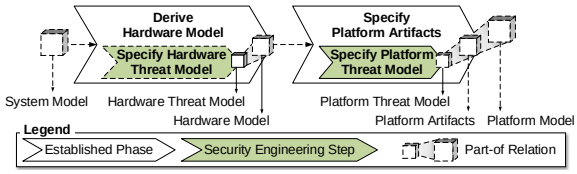



**Figure 6: Platform Design.**

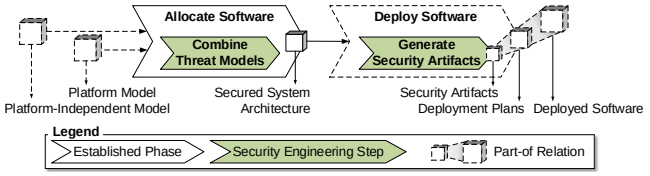



**Figure 7: Platform-Specific Software Design.**

Next, in the *Design Component Behavior* phase, software engineers create models for the coordination behavior of components. We extend this phase with a step to *Specify Assumptions* about countermeasures provided by the targeted computing platform. For example, in Figure 4b, we show a countermeasure against *Eavesdropping*, assuming that the platform ensures confidential communication. Another countermeasure against *Read Access* assumes that only user-encrypted data is sent to the cloud.

In the step *Verify Local Flow Policies*, we propose to use techniques from the area of *information flow control* [10] to verify components against their local flow policies. In case of a violation, software engineers track back the error either by correcting the message passing behavior, or by adjusting the flow policy. If a flow policy is complied, the information flow control serves as justification (cf. Figure 4c). A crucial condition for this verification approach is *compositionality* [15], i.e., if all local flow policies are complied, the global information flow policy must also be complied by definition. Finally, the component model and the behavioral models constitute integral parts of the *platform-independent model*.

## 3.3 Platform-Specific Software Engineering

Finally, we address the deployment of the application software to the hardware/software computing platform. We describe our extensions to the platform design in Section 3.3.1 and to the platform-specific software design in Section 3.3.2.

*3.3.1 Platform Design.* The final result of this phase is a *platform model* (cf. Figure 6) that describes hardware and platform software such as the operating system. During the phase *Derive Hardware Model*, platform engineers specify the available hardware, hardware composition, and network topologies. We propose to extend this phase with a step *Specify Hardware Threat Model* to represent physical threats like, e.g., influenced car sensors, access to the internal bus system, or eavesdropping over-the-air communication [23].

Next, in the phase *Specify Platform Artifacts*, platform engineers describe software artifacts that serve as a platform for the application software. This might contain non-security artifacts like math libraries but also security-related artifacts like encryption libraries (e.g., to encrypt user data before sending them to the cloud) or secure communication channels (e.g., between car and cloud storage). Such platform artifacts serve as security solutions for assumptions made in the platform-independent software design. In step *Specify Platform Threat Model*, we propose to represent such security solutions as another threat model in the form of *defense-attack graphs*. Essentially, these graphs correspond to the attack-defense graphs but use countermeasures like encryption as root nodes. Platform engineers can refine these graphs with possible attacks to the security solutions, e.g., breaking an encryption algorithm.

*3.3.2 Platform-Specific Software Design.* Finally, the software allocation and deployment takes place, bringing the platform-independent model together with the platform model (cf. Figure 7). First, the application software is mapped to the execution nodes of the platform. During this *Allocate Software* phase, deployment engineers need to resolve allocation constraints, e.g., with respect to performance or real-time scheduling but also security. For example, confidential communication must be mapped to nodes that communicate over secure channels. In parallel, during the step *Combine Threat Models*, deployment engineers combine the application threat model with the platform threat model by mapping assumptions to security solutions which results in a *secured system architecture* justifying all assumed countermeasures.

In the final *Deploy Software* phase, the actual deployment is prepared by creating detailed *deployment plans* that describe all necessary actions for a secure deployment. In addition, implementations of software artifacts can be generated, e.g., source code from the behavioral models, or architectural code from the component model. The attack-defense graph composed during the *Allocate Software* phase describes threats and countermeasures of all layers. Thus, in phase *Generate Security Artifacts*, this graph can be used to generate additional security-related artifacts such as penetration test cases, secure configurations (e.g., for security managers), or policies for code analyses if binaries or handwritten code need to be integrated. Such generated artifacts can be used to create justifications for the countermeasure nodes in the attack-defense graph. For example, for the countermeasure node in Figure 4b justified by a code analysis, specifications for this code analysis can be generated. Finally, also non-software artifacts can be derived from the threat model, e.g., installation plans or development guidelines. Such informal artifacts can be used to guide deployment engineers when justifying remaining countermeasures in the attack-defense graph. After a successful execution of an appropriate analysis, deployment engineers can mark this node as justified. Before the real system is deployed, all leaf nodes of the attack-defense graph must be either justified countermeasures, or attacks with acceptable risk.

# 4 RELATED WORK

Current systems engineering approaches [20, 29] lack an integration of security engineering practices. In contrast, approaches integrating security engineering into the software development lifecycle are surveyed in [5, 17]. However, comparatively few approaches consider security consistently across the entire lifecycle [17], as we propose in this paper. Furthermore, the aforementioned works focus on the secure development of information systems, and are not tailored to the specific characteristics of cyber-physical systems. In particular, they do not take the discipline-spanning systems

engineering into account, and therefore fail to integrate software engineering with other engineering disciplines involved in the development of cyber-physical systems.

In general, cyber-physical systems fall into the category of distributed systems. Approaches towards security for such systems are surveyed by Uzunov et al. [27]. In this area, model-based approaches are particularly promising for provable and traceable security due to their use of formal methods [18]. On the one hand, there are model-based approaches aiming at security for information systems in general, e.g., UMLsec [13], Model-driven Security [2], and OpenPMF [24]. These approaches can be used to specify security policies as well as corresponding analyses and enforcements. However, in contrast to our work, they do not focus on the secure deployment of cyber-physical systems to their computing platforms. On the other hand, further model-based approaches aim at cyber-physical systems in particular, e.g., SEED [28], SecureMDD [16], or SysML-Sec [22], which consider the platform of the system explicitly, and use formal methods to enforce or analyse security policies. However, in contrast to our work, both SEED and SecureMDD do not enable a dedicated threat modeling. As the most closely related work, SysML-Sec [22] provides a dedicated development process integrating techniques to enforce security. Among others, the authors also use attack graphs to specify and analyse attacks at the beginning of the development. In contrast to our approach, the attack graphs are not integrated with the design models in the subsequent steps, and therefore do not provide traceability to countermeasures.

## 5 CONCLUSIONS

The emergent need for security in cyber-physical systems requires appropriate engineering techniques to make systems secure by design. In this paper, we illustrate how secure software engineering practices can be integrated into an engineering process for cyber-physical systems. We describe how security requirements can be identified and specified at systems engineering level. In addition, we describe how these security requirements can be addressed systematically by taking appropriate countermeasures during software engineering. For this purpose, we use attack-defense graphs as threat model for tracing security requirements to both application-level countermeasures (e.g., information flow control) and platform-level countermeasures (e.g., cryptographic libraries). Thereby, we seek to increase the overall security of systems because requirements, threats, and countermeasures are made explicit and are traceable across the whole development lifecycle.

By extending a state-of-the-art development process, we seek to help engineers apply our approach in systems engineering practice. In future work, we plan to further strengthen the applicability of our approach by providing tool support for specification, analysis, and traceability. On this basis, we also plan to empirically validate the security benefits of our work and to investigate the integration of our approach with agile development methodologies.

## ACKNOWLEDGMENTS

Johannes Geismann is member of the Ph.D. program "Design of Flexible Work Environments: Human-Centric Use of Cyber-Physical Systems in Industry 4.0", supported by the German federal state of North Rhine-Westphalia.

## REFERENCES

[1] H. Anacker, C. Brenner, R. Dorociak, R. Dumitrescu, J. Gausemeier, P. Iwanek, W. Schäfer, and M. Vaßholz. 2014. Methods for the Domain-Spanning Conceptual Design. In *Design Methodology for Intelligent Technical Systems: Develop Intelligent Technical Systems of the Future*. Springer-Verlag, 117–182.
[2] D. A. Basin, J. Doser, and T. Lodderstedt. 2006. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology* 15, 1 (2006), 39–91.
[3] A. Chattopadhyay, A. Prakash, and M. Shafique. 2017. Secure Cyber-Physical Systems: Current trends, tools and open research problems. In *DATE 2017*. IEEE, 1104–1109.
[4] R. Crook, D. C. Ince, L. Lin, and B. Nuseibeh. 2002. Security Requirements Engineering: When Anti-Requirements Hit the Fan. In *RE 2002*. IEEE, 203–205.
[5] B. de Win, R. Scandariato, K. Buyens, J. Grégoire, and W. Joosen. 2009. On the secure software development process: CLASP, SDL and Touchpoints compared. *Information & Software Technology* 51, 7 (2009), 1152–1171.
[6] J. S. Fitzgerald, P. Gorm Larsen, and M. Verhoef. 2014. From Embedded to Cyber-Physical Systems: Challenges and Future Directions. In *Collaborative Design for Embedded Systems*. Springer-Verlag, 293–303.
[7] J. Gausemeier and S. Moehringer. 2002. VDI 2206 - A New Guideline for the Design of Mechatronic Systems. *IFAC Proceedings Volumes* 35, 2 (2002), 785 – 790.
[8] J. Giraldo, E. Sarkar, A. Cárdenas, M. Maniatakos, and M. Kantarcioglu. 2017. Security and Privacy in Cyber-Physical Systems: A Survey of Surveys. *IEEE Design & Test* 34, 4 (2017), 7–17.
[9] J. Greenyer and T. Gutjahr. 2017. Symbolic Execution for Realizability-Checking of Scenario-Based Specifications. In *MoDELS 2017*. IEEE, 312–322.
[10] D. Hedin and A. Sabelfeld. 2012. A Perspective on Information-Flow Control. In *Software Safety and Security*. IOS Press, 319–347.
[11] C. Heinzemann, O. Sudmann, W. Schäfer, and M. Tichy. 2013. A discipline-spanning development process for self-adaptive mechatronic systems. In *ICSSP 2013*. ACM, 36–45.
[12] J. Holtmann, R. Bernijazov, M. Meyer, D. Schmelter, and C. Tschirner. 2016. Integrated and iterative systems engineering and software requirements engineering for technical systems. *Journal of Software: Evolution and Process* 28, 9 (2016), 722–743.
[13] J. Jürjens. 2005. *Secure systems development with UML*. Springer-Verlag.
[14] B. Kordy, L. Piétre-Cambacédès, and P. Schweitzer. 2014. DAG-based attack and defense modeling. *Computer Science Review* 13-14 (2014), 1 – 38.
[15] H. Mantel. 2002. On the Composition of Secure Systems. In *IEEE S&P*. IEEE, 88–101.
[16] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. 2009. SecureMDD: A model-driven development method for secure smart card applications. In *ARES 2009*. IEEE, 841–846.
[17] N. M. Mohammed, M. Niazi, M. Alshayeb, and S. Mahmood. 2017. Exploring software security approaches in software development lifecycle: A systematic mapping study. *Computer Standards & Interfaces* 50 (2017), 107–115.
[18] P. Nguyen, M. Kramer, J. Klein, and Y. Le Traon. 2015. An extensive systematic review on the Model-Driven Development of secure systems. *Information and Software Technology* 68 (2015), 62–81.
[19] S. Peisert, J. Margulies, D. M. Nicol, H. Khurana, and C. Sawall. 2014. Designed-in Security for Cyber-Physical Systems. *IEEE Security & Privacy* 12, 5 (2014), 9–12.
[20] K. Pohl, M. Broy, H. Daemkes, and H. Hönninger. 2016. *Advanced Model-Based Engineering of Embedded Systems*. Springer-Verlag.
[21] A. L. Ramos, J. Vasconcelos Ferreira, and J. Barceló. 2012. Model-Based Systems Engineering: An Emerging Approach for Modern Systems. *IEEE Transactions on Systems, Man, and Cybernetics* 42, 1 (2012), 101–111.
[22] Y. Roudier and L. Apvrille. 2015. SysML-Sec: A model driven approach for designing safe and secure systems. In *MODELSWARD 2015*. IEEE, 655–664.
[23] F. Sagstetter, M. Lukasiewycz, S. Steinhorst, M. Wolf, A. Bouard, W. R. Harris, S. Jha, T. Peyrin, A. Poschmann, and S. Chakraborty. 2013. Security challenges in automotive hardware/software architecture design. In *DATE 2013*. IEEE, 458–463.
[24] R. Schreiner and U. Lang. 2004. OpenPMF: A Model-Driven Security Framework for Distributed Systems. In *ISSE 2004*. Vieweg, 138–147.
[25] A. Shostack. 2014. *Threat Modeling: Designing for Security*. John Wiley and Sons.
[26] G. Sindre and A. L. Opdahl. 2005. Eliciting security requirements with misuse cases. *Requirements Engineering* 10, 1 (2005), 34–44.
[27] A. V. Uzunov, E. B. Fernández, and K. Falkner. 2012. Engineering Security into Distributed Systems: A Survey of Methodologies. *Journal of Universal Computer Science* 18, 20 (2012), 2920–3006.
[28] M. Vasilevskaya, L. A. Gunawan, S. Nadjm-Tehrani, and P. Herrmann. 2014. Integrating security mechanisms into embedded systems by domain-specific modelling. *Security and Communication Networks* 7, 12 (2014), 2815–2832.
[29] T. Weilkiens. 2016. *SYSMOD - The Systems Modeling Toolbox*.