Thesis for The Degree of Doctor of Philosophy

Efficiency and Automation in Threat Analysis of Software Systems

Katja Tuma

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2021

Efficiency and Automation in Threat Analysis of Software Systems

Katja Tuma

ISBN 978-91-8009-155-8
ISSN 1652-876X

Technical Report No 191D
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

“It seems to me, Golan, that the advance of civilization is nothing but an exercise in the limiting of privacy.”

- Janov Pelorat in Foundation’s Edge, a novel by Isaac Asimov

Abstract

Context: Security is a growing concern in many organizations. Industries developing software systems plan for security early-on to minimize expensive code refactorings after deployment. In the design phase, teams of experts routinely analyze the system architecture and design to find potential security threats and flaws. After the system is implemented, the source code is often inspected to determine its compliance with the intended functionalities.

Objective: The goal of this thesis is to improve on the performance of security design analysis techniques (in the design and implementation phases) and support practitioners with automation and tool support.

Method: We conducted empirical studies for building an in-depth understanding of existing threat analysis techniques (Systematic Literature Review, controlled experiments). We also conducted empirical case studies with industrial participants to validate our attempt at improving the performance of one technique. Further, we validated our proposal for automating the inspection of security design flaws by organizing workshops with participants (under controlled conditions) and subsequent performance analysis. Finally, we relied on a series of experimental evaluations for assessing the quality of the proposed approach for automating security compliance checks.

Findings: We found that the eSTRIDE approach can help focus the analysis and produce twice as many high-priority threats in the same time frame. We also found that reasoning about security in an automated fashion requires extending the existing notations with more precise security information. In a formal setting, minimal model extensions for doing so include security contracts for system nodes handling sensitive information. The formally-based analysis can to some extent provide completeness guarantees. For a graph-based detection of flaws, minimal required model extensions include data types and security solutions. In such a setting, the automated analysis can help in reducing the number of overlooked security flaws. Finally, we suggested to define a correspondence mapping between the design model elements and implemented constructs. We found that such a mapping is a key enabler for automatically checking the security compliance of the implemented system with the intended design. The key for achieving this is two-fold. First, a heuristics-based search is paramount to limit the manual effort that is required to define the mapping. Second, it is important to analyze implemented data flows and compare them to the data flows stipulated by the design.

Keywords

Secure Software Design, Threat Analysis (Modeling), Automation, Security Compliance

Acknowledgment

In the words of Isaac Newton, *if I have seen further it is by standing on the shoulders of Giants.* Riccardo Scandariato, I owe you my gratitude for your wise guidance, encouragement, and priceless advice. You introduced me to the exciting world of research and on your shoulders I learned countless valuable lessons. For your kind and devoted mentorship, I remain deeply indebted to you.

I am extremely grateful to my co-supervisors Musard Balliu, Gül Çalikli and my examiner Robert Feldt for faithfully following my research and helping me strengthen my work. I would also like to thank Jan Jürjens and the entire RGSE group for welcoming me at the University of Koblenz-Landau. I am very grateful to my co-authors Sven Peldzsuz, Laurens Sion, Daniel Strüber, and Koen Yskout for the memorable debates and pleasant collaborations. Thomas Herpel, Christian Sandberg, Urban Thorsson, and Mathias Widman, thank you for many interesting discussions and your valuable perspective. This thesis would not have been possible without all your help and support.

For the past four years I have been incredibly lucky for having the most fantastic colleagues around me. I would like to especially thank my colleagues Thorsten Berger, Richard Berntsson Svensson, Ivica Crnkovic, Regina Hebig, Rodi Jolak, Eric Knauss, Grischa Leibel, Birgit Penzenstadler, Jan-Philipp Steghöfer, and the whole SE group for embracing me as their own and creating an amazing atmosphere. A special thanks goes to my colleague and pedagogics mentor Christian Berger, who has shown me how fun and fulfilling teaching can be. Richard Torkar, from day one you have made sure that I don't forget my mother tongue! Za tvojo podporo ti bom vedno hvaležna. Thank you Linda Erlenhov, Francisco Gomes de Oliveira, Jennifer Horkoff, Philipp Leitner, Antonio Matrini, Ildiko Pilan, and Joel Scheuner for all the awesome board-game nights! I also want to thank my PhD brothers Mazen Mohamad and Tomasz Kosinski for always being on my team. A huge thanks to my office mates Rebekka Wohlrab, Sergio García and Piergiuseppe Mallozzi for bringing color into the cloudy days.

I want to thank my friends Aura, Carlo, Lydia, Evgenii, Tuğçe, and Giacomo for all the potlucks, summer BBQs, movie nights, and hard-core climbing sessions. I will hold Gothenburg in my dearest memories because of you!

I am eternally grateful to my parents Tanja and Tadej for teaching me right from wrong, supporting my career and loving me no matter what. My dear brother Samo, thank you for putting up with your little sister all those years. Hvala da ste mi dali tako močne korenine. Rada vas imam, moji Tumčki!

Finally, I could never have made it without the most important person behind the scene: my husband, best friend, career advisor, and No. 1 paper reviewer, Marco. Words can not express how grateful I am to have you in my life. With you belaying me, I will fearlessly climb the next rock. Ti amo!

This research was partially supported by the Swedish VINNOVA FFI projects “HoliSec: Holistic Approach to Improve Data Security” and “CyReV: Cyber Resilience for Vehicles - Cybersecurity for Automotive Systems in a Changing Environment” and the Horizon 2020 project “AssureMOSS: Assurance and certification in secure Multi-party Open Software and Services”.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] K. Tuma, G. Calikli, and R. Scandariato.
"Threat Analysis of Software Systems: A Systematic Literature Review"
Journal of Systems and Software (JSS), 2018.

[B] K. Tuma and R. Scandariato.
"Two Architectural Threat Analysis Techniques Compared"
Proceedings of the European Conference on Software Architecture (ECSA), 2018.

[C] K. Tuma, R. Scandariato, M. Widman, and C. Sandberg.
"Towards security threats that matter"
Proceedings of the International Workshop on the Security of Industrial Control Systems and Cyber-Physical Systems (CyberICPS), 2017.

[D] K.Tuma, C. Sandberg, U. Thorsson, M. Widman, T. Herpel, and R. Scandariato.
"Finding Security Threats That Matter: Two Industrial Case Studies"
In submission *to the Journal of Systems and Software (JSS), 2020.*

[E] K. Tuma, M. Balliu, and R. Scandariato.
"Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis"
Proceedings of the International Conference on Software Architecture (ICSA), 2019.

[F] K. Tuma, D. Hosseini, K. Malamas, and R. Scandariato.
"Inspection Guidelines to Identify Security Design Flaws"
Proceedings of the International Workshop on Designing and Measuring CyberSecurity in Software Architecture (DeMeSSA), 2019.

[G] K. Tuma, L. Sion, R. Scandariato, and K. Yskout.
"Automating the Early Detection of Security Design Flaws"
Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), 2020.

[H] S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, and R. Scandariato.
"Security Compliance Checks between Models and Code based on Automated Mappings"
Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), 2019.

[I] K.Tuma, S. Peldszus, R. Scandariato, Daniel Strüber and J. Jürjens.
"Checking Security Compliance between Models and Code"
In submission *to the Journal on Software and Systems Modeling (SoSyM), 2020.*

Other publications

The following publications were published during my PhD studies, but are not appended due to overlapping or unrelated content to the thesis.

(a) S. Jasser, K. Tuma, R. Scandariato, M. Riebisch.
"Back to the Drawing Board"
Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP), 2018.

(b) L. Sion, K. Tuma, R. Scandariato, K. Yskout, and W. Joosen.
"Towards Automated Security Design Flaw Detection"
Proceedings of the International Conference on Automated Software Engineering Workshop (ASEW), 2019.

Research Contribution

I contributed with planning and conducting the systematic literature review (Paper A). In this work I was responsible for selecting the studies, creating the assessment criteria, data extraction and result analysis.

In the empirical study reported in Paper B, I helped conducting the experiments (on-site) in the second year. I also built the base line analysis (ground truth), assessed the reports with respect to the ground truth for both experiments, and drove the result analysis.

For Paper C, I developed the approach during the workshops with our industrial partners and evaluated it with an illustration.

In Paper D, I contributed with an improved analysis procedure (eSTRIDE), prepared the study material, helped to design the case studies, conducted the workshops (on-site), and analyzed the results.

The formalism behind the label extension in Paper E was contributed by my co-author, Musard Balliu. For this work, I implemented the domain-specific language using the Eclipse Plug-in Framework and conducted the evaluation.

In Paper F, I supervised the creation of the catalog of design flaws and I re-evaluated the catalog.

In paper G, I was responsible for designing the empirical study, preparing the study material, and collaboratively implemented the automated detection tool. In addition, I conducted the study with the participants (and one expert) on-site one University campus. Finally, I was driving the tool performance analysis.

In Paper H, I helped to shape the heuristic rules and the mapping, contributed to the implementation of the approach (but was not the main driving force), contributed to the design, and execution of the evaluation (including result analysis).

In Paper I, I contributed to the design of the security compliance checks, the implementation of the checks, the design and execution of two experiments, and the result analysis of one of the experiments.

In all the appended papers (except Paper H), I was the driving force and wrote major parts of the publications.

Contents

Chapter 1

Introduction

Security threats to software systems are becoming a growing concern in many organizations, particularly due to the changes in legislation for handling private user data (GDPR). Previous studies (summarized in [1]) have shown that information security breach announcements result in a financial loss for the breached organization. Notably, last year the British Airways was fined £183 million [2] (1.5% of total yearly revenue) due to a data breach affecting 500,000 customers.

Despite best efforts, cyber-attacks are often successful due to poor security practices. In August 2019, researchers discovered a vulnerability [3] in the database of a biometric security platform (Biostar 2). In particular, they found an unprotected database in the platform, where data was stored in plain text. If exposed, this vulnerability could have allowed the attacker not only to expose biometric information of over one million people, but also to gain access to administrator accounts of the platform, possibly leading to serious criminal activity (such as creating a new account to freely enter high-security facilities).

Software developers can be hindered in building secure solutions by fast-pace development practices, code reuse, and the use of third-party software. Commonly, vulnerabilities are introduced with the incorrect use of third-party APIs. For instance, a recent report [4] revealed that about 24,000 Android apps used insecurely configured Firebase databases, which allowed researchers to gain read and write access to sensitive database records.

To avoid expensive data breaches, security should be considered early-on in the software development life cycle [5]. Practitioners that value security in their products adopt well established best practices, e.g., by applying secure design principles [6] and patterns [7]. Architectural design models are often analyzed to assure the desired properties of the system. Models can be analyzed from different architectural perspectives (e.g., topological view, data view, access control and permissions, functional view, etc.) and on several levels of abstraction.

The goal of this thesis is to improve on the performance of security design analysis techniques (in the design and implementation phases) and support practitioners with automation and tool support. The novelty of the thesis contributions is judged with respect to the related work in Section 1.1. An in-depth study of existing threat analysis techniques (Paper A) has spurred three focused research directions which are presented in Section 1.2.

A recent report [8] shows that only about a third of 130 surveyed organizations analyze software architecture for what concerns security. The **manual**

effort that is today required to perform architectural threat analysis may be a limiting factor for a more wide-spread adoption. Indeed, evidence suggests that performing threat analysis manually (e.g., using STRIDE [9]) results in a large number of discovered threats and can be repetitive [10] (also observed in Paper B). After their discovery, the threats are prioritized based on risk values and low-priority threats are often discarded. This way of working is suboptimal, as effort is wasted on discussing low-priority threats. Further, eliciting threats by considering each architectural element in isolation may be a source of repetitiveness. We provide a solution for performing threat analysis with an enlarged analysis scope (per asset scenario) and focusing on critical parts of the software architecture. Concretely, we developed a model-based technique, which fits particularly well in model-intensive industries, e.g., automotive. We propose a notation extended with risk information (eDFD) accompanied by an *improved analysis procedure* (eSTRIDE) for an efficient discovery of high-priority threats (Paper C). The approach relies on making reductions to the problem and solution space before and during the analysis. We empirically investigate the effect of enlarging the analysis scope on technique performance in the academic (Paper B) and industrial setting (Paper D). Section 1.3 provides summaries of the appended publications.

Manual design analysis and inspection techniques [9, 11] are characterized by a **low recall** (around 50% in Papers B and F), which means that many security flaws can go unnoticed. A possible cause for this effect is that in many existing techniques there is no correctness or completeness guarantees for what concerns the analysis outcomes (as recorded in Paper A). To assure analysis completeness, more automation of design-level security techniques is necessary. To that end, we propose two solutions. First, we introduce a *formally-based detection of confidentiality flaws* (Paper E). In information flow security, the implementation is statically analyzed for a particular set of inputs to determine potential leaks of sensitive information. Initially the inputs are assigned so-called security labels. Typically, a high label refers to a private input and a low label refers to a public input. Similarly, we propose an approach for information flow analysis at design level. The approach is based on a light-weight formal specification language (SecDFD) which we leveraged to propose a technique for an automated analysis of confidentiality flaws with label propagation and a security policy checker. Second, we propose a *graph-based detection of five security design flaws* concerning various security properties (authentication, confidentiality, integrity, and accountability). The five security design flaws were selected from a catalog of security design flaws and their manual inspection rules (presented in Paper F). To model the key security concepts commonly referred to by the inspection rules, we suggest to use a design notation extended with data types and security solutions (Paper G). Further, we developed graph query patterns to automatically detect the presence of the five flaws in concrete design models. We empirically compare the performance of the query patterns over a curated data set of design models. In Section 1.4 we discuss the collective results and answer the main research questions.

Finally, after the implementation phase, architectural design models are rarely revisited. In fact, Hebig et al. [12] have studied 3295 open source projects and found that only 26% ever updated their UML files at least once. Thus, there is a **disconnect** between design models (possibly containing important security information) and their implementation. To address this issue, we introduce a user-in-the-loop approach to establish a mapping between the intended design and the implemented code (Paper H). We also extend the said approach to include *automated security compliance checks* (Paper I). Concretely, we defined a mapping between the DFD model (using the SecDFD presented in Paper E) and the program model [13] which is extracted from the implementation. To limit the required manual effort, we developed a heuristic-based search for possible mappings, which is based on name matching and structural similarities between the two abstractions. Paper I extends this work with two types of static checks, which were used to verify whether implemented programs complied with prescribed security properties in the SecDFD. In addition, using our approach we show that the security information in the intended design can be used to reduce the number of false positives reported by a state-of-the-art data flow analyzer. We present our final remarks and chart a vision for future work in Section 1.5.

1.1 Positioning of Contributions with Respect to the Related Work

This section includes a short background and a positioning of the main thesis contributions with respect to the related literature.

1.1.1 Threat Analysis of Design Models

The first main thesis contribution focuses on improving model-based threat analysis. Threat analysis includes activities which help to identify, analyze and prioritize potential security threats to a software system and the information it handles. A threat analysis technique consists of a systematic analysis of the attacker's profile, vis-a-vis the assets of value to the organization. The main purpose for performing threat analysis is to identify and mitigate potential risks by means of eliciting or refining security requirements. Existing threat analysis techniques are commonly categorized as software-, risk-, and attack-centric.

Software-centric. Software-centric techniques focus the analysis around the software (e.g., architecture design). Their first objective is to establish a good understanding of how the system works before the threats are analyzed. For instance, STRIDE is well-known software-centric technique which is extensively used in practice (e.g., in the automotive industry [14], at Microsoft [9] and some agile organizations [15]). In addition, it has been applied to analyze systems from a variety of domains (such as IoT [16, 17], CI Pipelines [18], MySQL DBs [19], Smart Grids [20], E-health [21], Networks and Protocols [22–24]) across different research communities. STRIDE is well documented and easy to learn, which is witnessed by its popularity. It is using the Data Flow Diagram (DFD) model to represent the architecture of the software under analysis. The analysts manually visit the elements in the diagram and brainstorm for potential security threats. At the end, the list of identified threats is prioritized (based on risk values) to plan for most urgent mitigations. But, with larger DFD models, the number of threats the analysts have to consider explodes, resulting in a high manual effort [10].

Risk-centric. Risk-centric techniques (e.g., CORAS [25], OCTAVE [26–28], PASTA [29]) focus on assets and their value to the organization. Their main objective is to estimate the financial loss for the organization in case of threat occurrence. For instance, CORAS [25] is a methodology comprised of a modeling language and a multi-step procedure of analysis. It provides guidelines and tools (namely, asset, threat, risk, and treatment diagrams) to analyze risk. Risks are analyzed twice in the procedure of CORAS [25]. First after the creation of asset diagrams (step 3), the analysts conduct a high-level risk analysis, where the most important assets (and their threats) are identified. Second, the risks are analyzed using threat diagrams, after which the risk can be accumulated (using risk diagrams). However, an empirical comparison of five risk-centric techniques [30] highlights its slow learning curve and long execution time. Risk-centric techniques (specifically, OCTAVE [27] and PASTA [29], as mentioned in [31]) are more appropriate for finding organizational risks, rather than technological risks. Accordingly, risk-centric techniques require a deeper understanding of the business goals and legal matters [32], which is scarce in organizations. For instance, in some Agile companies [33], the developers that perform threat analysis collect feedback from business experts for what concerns asset and risk related information.

Attack-centric. Finally, attack-centric techniques (Attack trees [34], Misuse Cases (MUC) [35], Problem and Abuse Frames [36–38], to name a few) focus on the hostility of the environment and analyze attacker's motives and behavior. For instance, Attack trees [39] are formally grounded models of all possible attacker actions. The root node (i.e., final goal of the attacker) is refined with a combination of logical gates (e.g., and/or gates) down to leaf nodes. They have been extensively used and adapted in the past [40] to analyze different properties in various domains, including, for instance in the automotive industry (e.g., the EVITA method [14]). Attack trees are often used to support brainstorming threats (e.g., in STRIDE [9], PASTA [29], and LINDDUN [11]). However, creating new attack trees is challenging as it requires both advanced cyber security background and technical knowledge about the domain. Besides the approaches that are based on Problem Frames (e.g., the approach presented in [36]), the outcomes of many attack-centric techniques are not tied to the architectural elements of the system under analysis.

To understand how to reduce the high manual effort, we compare two STRIDE variants in Paper B. Further, to focus the analysis on high-priority threats without sacrificing the quality of outcomes and learnability, we introduce a new (risk-centric) STRIDE variant in Paper C and evaluate it in Paper D.

1.1.2 Automated Security Analysis of Design Models

Many approaches propose to automate the analysis of design models to minimize the resources needed for performing threat analysis in organizations. Often such approaches are able to semi-automate the analysis. That is, they automate parts of the analysis technique, while some parts still require manual effort. Depending on the sophistication of the analysis automation, we continue to describe knowledge-based automation of threat categories, graph-based automation, and formal approaches.

Knowledge-based. The Microsoft Threat Modeling tool (MTM) [41] is a tool developed to support the STRIDE methodology. MTM provides the ability to graphically represent the DFDs. The tool enables the generation of threat categories for individual DFD elements with the use of the STRIDE threat-to-element mapping table. Other works approach threat analysis automation in a similar way. For instance, Sion et al. [42] present an approach which aims to automate the selection of threat mitigations (i.e., matching threat categories (e.g., spoofing) to security solutions). Yet, both approaches automatically generate threat categories (based on the aforementioned mapping table), thus actual attack scenarios still need to be discovered manually.

Graph-based. Design models (e.g., software architecture) can be sometimes represented as graph-like structures. A common method for automating the analysis of design models is by discovering patterns in such graphs. Depending on the analysis focus, the graph patterns can be used to detect threats, vulnerabilities, or security solutions. Seifermann et al. [43] presented an approach for automatically analyzing the security of data-driven architectures. They propose an architectural description language enriched with a data model. The architecture is first transformed to an operation model, which is in turn transformed to a logic program. Finally, logical queries are used to spot security flaws. However, the analysis is demonstrated for unauthorized authentication, while other security design flaws (e.g., insufficient auditing) are not addressed. In addition, the analysis is not conducted alongside the planned security mechanisms. Almorsy et al. [44] proposed an approach for automating the security analysis by capturing vulnerabilities and security metrics. They developed an approach for modeling a system and specifying signatures of vulnerabilities and security metrics with the Object Constraint Language (OCL). Yet, the suggested approach does not provide a way to model data transformations, which affect security properties. In addition, it is not clear whether the approach works for high-level design models (such as the DFDs), as it takes as input a variety of system description models (e.g., UML feature, component, class, and deployment diagrams). Berger et al. [45] develop graph query rules to check for vulnerabilities in extended DFD models and evaluate them with case studies. The query rules are based on the descriptions of existing vulnerability repositories (e.g., CWE, CAPEC). Though the authors provide a way to extend the DFD with asset sensitivity, their approach does not allow modeling of security mechanisms.

Formal. When more effort for modeling (and analysis of) system design is justified, formal approaches can be adopted. Such approaches typically require the modelers to have a strong background in formal methods and topics alike. The automation of analysis reasoning in a formal setting comes sometimes for free due to the underpinned semantics. Yet, the efficiency and scalability of such approaches is often a challenge. Concretely, a survey on graphical security models [46] reported that there is a lack of efficient generation algorithms for tree-based models. Early work of Sheyner et al. [47] automate the generation of attack graphs, based on the well understood formalism of attack trees. Later-on Ou et al. [48] worked towards increasing the scalability of attack graph generation. On the other hand, Xu et al. [49] approached automating threat analysis with aspect-oriented petri nets. The authors model the intended functions and security threats with Petri nets, whereas they model threat mitigations with Petri net-based aspects. Given the presented semantics, the authors are able to construct a search tree and verify whether certain threat paths are possible in the model. Gerking and Schubert [50] propose an approach for refining and verifying information-flow policies (i.e., non-interference) for cyber-physical architectures. Their compositional verification technique relies of a set of well-formedness rules for architecture refinement and assembly of component diagrams, preserving non-interference. Compared to DFDs, component diagrams are more detailed design models. With regards to semantics of DFDs, the early work of Leavens et al. [51] and Larsen et al. [52] extended the notation with specifications for expressing functional correctness properties. But little work has focused on the security semantics of DFDs.

Since identifying feasible security threat scenarios depends on the knowledge of emerging security attacks within a domain, we do not attempt to automate generation of threat scenarios. Rather, we focus on strengthening the security by automating the detection of security design flaws. To that end, we study how to automate the security design flaw detection on high-level architectural diagrams (i.e., DFDs) in Papers E, F and G. Our aim is to improve the automation of model-based security analysis where the related work falls short. First, we introduce a lightweight security specification of DFDs (Paper E), extended with a simple label model for analyzing confidentiality flaws with some completeness guarantees. Second, we study how to automatically detect five security design flaws (for what concerns several security concerns) by means of graph query patterns, which are executed over DFD models, enriched with data types and security solutions (Papers F and G).

1.1.3 Security Compliance Between Model and Code

Once a design model has been analyzed and its security flaws have been fixed, the results are of limited value if the implementation does not comply with the security properties described in the model. The disconnect between the planned and implemented security has been studied extensively in the domain of Model-Driven Engineering (MDE) [53, 54], where the intended security properties are propagated to code by means of forward engineering. On the other hand, many approaches (summarized in [55]) suggest to solve the problem of disconnect by means of reverse engineering, where often code annotations or intermediate models are used to reconstruct the software architecture. Finally, traceability management approaches study the relations between software artifacts to enable change-impact analyses and support software maintenance. Though traceability recovery approaches may also lean on reverse engineering techniques, we discuss this research area separately. Concretely, we consider approaches that study the refinement traceability (where the level of abstraction of the traced artifacts lowers progressively), rather than variability within a family of models (e.g., in software product lines engineering [56]) and their variations.

Forward engineering security. UML models have been heavily studied in the context of security compliance by means of forward engineering. UMLsec [57] is a security extension of the Unified Modeling Language. It enables developers to express security relevant information in system specification diagrams. It has been widely studied in the industrial context [58–60] and provides mature tool support [61]. Fourneret et al. [62] combine the security analysis using UMLsec with the generation of security tests. The authors specify and verify security properties on UML state machines, which in turn are used to generate tests for the implemented system. Further, Ramadan et al. [63] use model transformations to generate security-annotated UML class models from security-annotated BPMN models. Muntean et al. [64] extend UML statecharts with security properties (e.g., source of a confidential record), generate the code (in C), and then detect data flow violations statically in the implementation. The results of the compliance checks are presented to the user with sequence diagrams. But, the gap between statecharts (or class diagrams) and implementation is much smaller compared to the gap between high-level design models and implementation. Consider that the DFD notation does not allow modeling conditional or sequenced data flows. The IFlow [65] approach presents a UML extension with information flow properties, which is used to generate program skeletons. The generated skeletons are then transformed to a formal model to be proven with a theorem prover. The skeletons have to be manually completed into a final implementation, over which standard information flow properties can be checked using existing analyzers. The downside of relying on code generation, though, is that such approaches can not be used to analyze software implementations which have diverged from the original model, or code that was not generated from a model.

Reverse engineering security. Scoria [66] is a semi-automated approach for extracting and analyzing the Owner Object Graph annotated with security properties (i.e., SecGraph) to find security flaws in the architecture. First, the SecGraph is extracted from an annotated implementation. Second, software architects can refine the SecGraph with additional annotations. Finally, software architects can design queries to analyze the Sec-Graph. Similar to using source code annotations, ArchJava [67] is a language extension for Java, which integrates architectural concepts (i.e., components, connectors, and ports) into the programming language itself. Extending the expressiveness of the programming language with architectural concepts supports compliance analysis. For instance, in [68] the authors extend ArchJava with security annotations and develop architectural constraints to analyze security compliance. Though code annotations (and language extensions) can increase program comprehension and reduce maintenance costs, they also need to be well understood (together with the source code) to be used correctly [69]. Fully comprehending security code annotations is not trivial and may require additional developer training. Jasser [70] recently proposed an approach for analyzing system behavior and detecting its discordance with a set of security rules, expressed with Linear Time Logic (LTL). The system behavior is extracted dynamically using aspect-oriented programming. Before the security rules can be executed, the source-level elements are mapped to the architectural elements. However, this mapping is performed manually. To date, the sole attempt at establishing compliance between DFDs and their implementation was introduced by Abi-Antoun et al. [71] more than a decade ago. The authors automatically extract a DFD (i.e., the source DFD) from the implementation. Next, the user specifies a mapping (using Reflexion models [72]) between a manually created high-level DFD and the source DFD, which is then used to uncover inconsistencies. However, the Reflexion models are created manually. In addition, the security analysis is performed on the level of the DFD, as opposed to the implementation.

Traceability. Most traceability link recovery techniques seek to establish a connection between requirements and code [73]. To this end, the proposed approaches use information retrieval techniques in combination with heuristic-

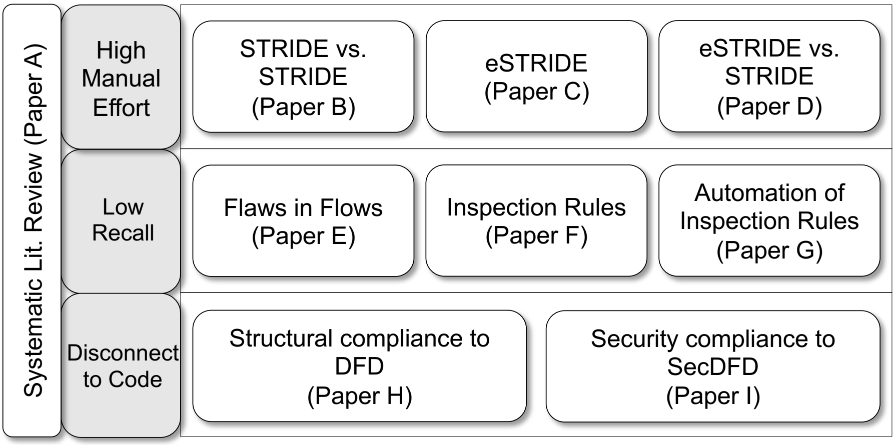

Systematic Lit. Review (Paper A)
High Manual Effort
STRIDE vs. STRIDE (Paper B)
eSTRIDE (Paper C)
eSTRIDE vs. STRIDE (Paper D)
Low Recall
Flaws in Flows (Paper E)
Inspection Rules (Paper F)
Automation of Inspection Rules (Paper G)
Disconnect to Code
Structural compliance to DFD (Paper H)
Security compliance to SecDFD (Paper I)

Figure 1.1: Research Tracks

based analysis of source code representations (e.g., the abstract syntax tree). For instance, Velasco and Aponte [74] recently introduced an approach for creating fine-grained traceability links between program statements (incl. conditionals, assignments, loops, etc.) and critical requirements to ease compliance checking (dictated by regulatory bodies, i.e., HIPAA [75]). First, the requirements and source code files undergo a text processing phase (incl. tokenization, tagging, stop word removal and the like). Next, the authors leverage an information retrieval (IR) technique called Latent Semantic Indexing to identify the most relevant source files for each requirement. Finally, to obtain a ranked list of relevant program slices, predefined criteria (respecting a particular requirement) is used to perform program slicing. Feature location approaches [76] leverage IR techniques in a similar way to determine locations in the source code where a particular functionality is realized. However, most traceability link recovery and feature location techniques rely on analyzing textual similarity. They fail to take into account structural properties of the early software design artifacts (e.g., DFDs), which are essential to capture cross-cutting concerns (such as security) in the source code.

To fill these gaps, we study how to automate the discovery of structural compliance (using both textual similarity and structural heuristic rules) of the implementation to DFDs in Paper H. Our approach does not rely on code annotations and can be applied to existing (Java) projects without any code generation. In addition, we intentionally keep the user in the loop to benefit from domain knowledge and enable a meaningful analysis. In Paper I, we extend the approach with automated security compliance checks of data flow properties by leveraging static code analysis techniques.

1.2 Research Focus

This thesis contributes to solving three problems that were identified in Paper A by means of a systematic literature review (SLR). Accordingly, Figure 1.1 shows the contributions organized into three research tracks. The thesis findings generally progress from left-to-right in this figure. But, some findings of the first research track have steered the work later-on and some research was conducted concurrently. For instance, the SLR was conducted concurrently with Papers B and C. The rest of the appended publications build on top of the previous.

1.2.1 High manual effort

The first research track was oriented towards an industrial collaboration with the automotive industry. With respect to the current state of practice, lack of security expertise is a crucial matter and increasing the efficiency of threat analysis could free valuable resources. For this reason, we studied how to reduce the time spent on analyzing threats without sacrificing the quality of outcomes. Namely, we wondered whether it is acceptable (given the time constraints) to start from an analysis of assets and their risk-related importance and only analyze important threats. Clearly, there is a trade-off between systematicity and focus on important threats. In Paper C, we explore this trade-off and provide a risk-first solution, named eSTRIDE.

In parallel to this study, we worked on building a deeper understanding on how the analysis procedure affects the performance of security analysis. Specifically, we were interested to understand how the procedure of visiting the architecture facilitates designers in identifying threats. To this aim, we looked into the scope of analysis, i.e., the number of elements analyzed at once by the human expert. On the one hand, there exist such techniques that suggest practitioners to find threats to architectural components in isolation (e.g., STRIDE-per-element). Further down the line, some techniques suggest finding threats to a set of components (e.g., STRIDE-per-interaction). Finally, we propose an end-to-end analysis techniques that suggest finding threats to a chain of components (i.e., eSTRIDE). We hypothesized that manual threat analysis performs better when the scope of analysis increases and leads to a more efficient discovery of the most important threats. We conduct an empirical comparison of two existing techniques to test this hypothesis (Paper B). The findings of this study provided inspiration for the definition of an improved manual analysis procedure (eSTRIDE). In Paper D, we continue on this path and conduct two case studies in two organizations, based in two different countries. First, we empirically compare the performance and execution of eSTRIDE to STRIDE-per-element. Second, we question the effect of security expertise on the quality of outcomes. To this end, we empirically compare the performance (and execution) of the less security savvy teams to the teams with security expertise.

The goal of this track is to introduce an efficient manual approach for finding important security threats by enlarging the analysis scope. Collectively, the research conducted in this track (Papers B, C, and D) aims to answer the following research question.

RQ1. What are the effects of broadening the analysis scope on the quality of analysis outcomes?

To answer this research question, we faced three challenges.

RQ1.1. What changes are required in the design model to facilitate a threat analysis focusing on important threats? (Paper C)

RQ1.2. What changes are required for a model-based threat analysis procedure to focus on important threats? (Paper C)

RQ1.3. What is the difference (in terms of performance and execution) between a risk-first and risk-last threat analysis technique? (Paper D)

RQ1.1. Enlarging the analysis scope introduces a challenge for the human expert as a higher cognitive load may harm the quality of analysis outcomes. At the same time, identifying locations in the architecture where important security threats may exist (before actually identifying the said threats), requires the model to be extended with security risk information. Thus, striking the right abstraction (and level of detail) of the model is a crucial step in developing a technique focused on finding important security threats.

RQ1.2. The model extensions provide more security-relevant information to the human expert. However, the extensions alone do not help the analyst during the discovery of security threats. Hence, the analysis procedure needs to be modified. First, the procedure of striving towards systematicity and considering risk at the end did not seem appropriate anymore. Rather, focusing the analysis towards high-priority threats requires leveraging the risk information during the analysis. Second, the strategy of visiting the diagram per element (or interaction) does not take advantage of the model extensions. Accordingly, the second challenge was to reduce the manual effort as much as possible by introducing short-cuts during the analysis.

RQ1.3. Finally, extending the analysis scope and introducing short-cuts during the analysis must not harm the overall technique performance or the quality of the analysis outcomes. The introduced threat analysis technique (eSTRIDE) considers risk information at the very beginning of the analysis. Therefore, it is a *risk-first* technique. In comparison, the STRIDE-per-element suggests to prioritize threats based on risk at the end, and is thus a *risk-last* approach. The final challenge in this track was to gather empirical evidence about sacrificing systematicity for the discovery of important threats.

1.2.2 Low recall

The findings of the first research track have influenced our research agenda in the second research track. The empirical evidence gathered is witness to the limits of tweaking the efficiency of manual threat analysis. For instance, analysis paralysis (i.e., discussing one threat in too much detail) slows the analysis down. Further, sub-optimal team dynamics as well as terminology disagreements may have a negative impact on the quality of outcomes. In addition, many real security threats are overlooked, may it be due to time pressure, lack of information, or simply human error. To overcome such challenges automation of the analysis is an important step.

Architectural security threats exist due to the presence of security design flaws. Therefore, the goal of this research track is to study how security design flaws are inspected, and how they can be detected automatically. The results that emerged in this research track (Papers E, F, and G) collectively aim to answer the second research question.

RQ2. To what extent can security design flaws be automatically detected in DFD-like models?

Concretely, we faced two challenges in our efforts to answer this question.

RQ2.1. What model extensions support an automated security design flaw detection? (Papers E, F)

RQ2.2. What performance can be achieved by an automated technique for security design flaw detection? (Paper G)

RQ2.1. The informal notation of the DFD makes automation difficult. Therefore, we first study the level of formalism that is required in the DFD to automate the detection of confidentiality-related design flaws in Paper E. Formal reasoning about confidentiality (and integrity) is well understood in the area of language-based information flow security [77]. We lean on the theory of information flow analysis, an area of research whose origins date back to the late 70s [78]. To avoid overloading the analysts, we intentionally extend the DFD with light-weight security semantics. Achieving this, together with a formally-based security analysis of the DFD was challenging.

The light-weight extension proposed in Paper E does not support reasoning about other security properties (e.g., authentication). To this aim, we compile a catalog of security design flaws and their inspection rules (introduced in Paper F). We selected five security design flaws from the catalog to study their automated detection. Our next challenge was the design of a sufficient model extension to capture the concepts required to reason about the presence of flaws in the models.

RQ2.2. The second challenge is to understand what levels of performance can be achieved by automating the detection of security design flaws. To this aim, we first translate the inspection rules of five security design flaws into graph query patterns, which we use for the automated detection. We conduct an empirical study comparing the outcomes of the automated technique to a manual inspection (ground truth) performed by human experts. The main challenge we faced was obtaining a data set of publicly available DFD models. In addition, to enable an empirical comparison, we had to conduct an assessment of the collected data set of DFD models with human experts under controlled conditions.

1.2.3 Disconnect between models and code

After performing a manual (or automated) security analysis of design models, there is yet a question that begs for an answer: How do the outcomes of such analyses relate to the implemented program? Much effort is spent on planning the intended security on the level of the design. But, without an explicit relation to the implementation, this effort is not leveraged to its full potential. In addition, model-level analyses do not provide a realistic picture of the implemented security, which diminishes the usefulness of models later-on the in the development life-cycle. The value of the model-level analysis could be increased, if such an explicit relation existed.

The goal of this research track is to study the relation between design and implementation, particularly for what concerns the security compliance. Collectively, the research conducted in this track (Papers H and I) aims to answer the third research question.

RQ3. What security code analysis techniques can be leveraged to discover the security compliance of the implemented system to SecDFD models?

We faced three problems in our effort to automate security compliance checks.

RQ3.1. What relation between the DFD model and an intermediate code representation supports automated security compliance checks? (Paper H)

RQ3.2. What security code analysis techniques can be leveraged to discover security compliance to the node contracts specified in the SecDFD? (Paper I)

RQ3.3. What information from the SecDFD complements existing static code analysis tools? (Paper I)

RQ3.1. First, to enable automated compliance checks, we establish an explicit mapping of high-level elements (e.g., a DFD process) to implemented constructs (e.g., implementation of a method). To this aim, we define rules for mapping element types between two representations. The first representation is the high-level design model (i.e., the SecDFD introduced in Paper E). The second representation is an automatically extracted model of the implementation (i.e., the program model [13]). Finding the appropriate corresponding element types between these two abstractions was our first challenge. In addition, understanding what heuristic rules can help in the discovery of corresponding elements was not trivial.

RQ3.2. The second challenge we faced was understanding what code analysis techniques can be leveraged to detect security compliance, given the explicit mapping between the design model (i.e., SecDFD) and its implementation. The SecDFD allows specifying contracts for the node elements, which precisely define how the confidentiality of an incoming asset(s) changes on the outgoing asset(s). For instance, the encrypt contract bound to one incoming asset and one outgoing asset produces a public (not confidential) output. We were interested to leverage the previously proposed mapping (Paper H) and static code analysis techniques to verify whether the node contracts are implemented as intended.

RQ3.3. Existing data flow analyzers require the user to correctly identify sources and sinks of confidential information. Though some sources and sinks can be extracted from library APIs (e.g., like in [79]), finding project-specific sources and correct sinks still remains a challenge. Besides developing the checks for each node contract in isolation, we were interested to statically analyze security of the entire program. Concretely, we wondered if the outcomes of an analysis on the model-level (e.g., allowing some data to flow into a sink) can be used to complement existing static code analysis tools. We hypothesize that our mapping between the intended design and its implementation may be used to point to locations in the code where secret information is first created, and locations where it is allowed flow. The challenge was to extract this information from the SecDFD in a way that can be useful to existing code analysis tools.

1.3 Paper Summaries

This section includes a summary of the appended papers. We describe our research goals, adopted methods, and main contributions. The reader may refer to the individual papers for a detailed discussion of the related work.

1.3.1 SLR on Threat Analysis (Paper A)

The number of existing threat analysis techniques makes it difficult for practitioners to make informed decisions about selecting the appropriate method for adoption in their organizations. Further, the existing literature on systematizing the knowledge about threat analysis is limited and does not provide a complete list of existing techniques. The primary goal of Paper A is to catalog and characterize the existing threat analysis techniques. The second goal is to provide guidelines for practitioners in selecting techniques for adoption, and to identify knowledge gaps for future research directions. In this study we compare 26 identified methodologies for what concerns their applicability, characteristics of the required input for analysis, characteristics of analysis procedure, characteristics of analysis outcomes, ease of adoption, and their validation. The study was conducted by strictly following the guidelines by Kitchenham et al. [80] and included an elaborate strategy, including backwards snowballing [81] for searching the literature and extracting the data. In addition, we discuss the obstacles for adopting the identified techniques to current trends in software engineering (i.e., Development and Operations, Agile development) and their generalization across domains. Finally, the study provides recommendations to practitioners for technique adoption depending on the amount of planned resource investment.

Contributions. The main findings of the SLR are: (i) Existing threat analysis techniques lack in quality assurance of outcomes, (ii) the use of validation by illustration is predominant, (iii) the tools presented in the primary studies lack maturity and are not always available, (iv) there is a lack of correctness and completeness guarantees for analysis outcomes. The SLR was performed as part of an in-depth study of the state-of-the-art, hence it does not contribute to any of the research questions listed in Section 1.2.

1.3.2 STRIDE-per-el vs STRIDE-per-inter (Paper B)

Among other things, threat analysis techniques may differ in the scope of analysis. We were interested to study the effects of a different analysis scope on the technique performance. To this aim, Paper B rigorously compares two existing techniques with different scopes, namely STRIDE-per-element and STRIDE-per-interaction [9]. In particular, this study measures the respective techniques' performance in terms of their productivity, precision, and recall. The study was conducted in the context of in-vitro experiments with master students. We adopted a standard design for a comparative study [82] of one independent variable with two values, namely, ELEMENT and INTERACTION. The participants were split into two treatment groups, the ELEMENT and INTERACTION treatment group. They were further assigned to teams. The teams were instructed to (i) create a DFD and (ii) perform a threat analysis of a familiar system using the respective technique in a fixed time frame and report the analysis results. We collected the measure of effort (in minutes) spent by each team on both sub-tasks (DFD creation and threat analysis). The final reports were compared to a ground truth analysis to collect the measure of true positives (TP), false positives (FP) and false negatives (FN). On that basis, we collected evidence about statistically significant differences (SSD) between (i) the average productivity (number of TP per hour) of treatments, (ii) the average precision ($TP/(TP+FP)$) of treatments, and (iii) the average recall ($TP/(TP+FN)$) of treatments. Beyond that, the study controlled for any possible discrepancies between the populations of the treatment groups (i.e., with an obligatory entry and exit questionnaire) and gathered subjective feedback on the usability of the techniques.

Contributions. Paper B contributes towards answering **RQ1**. The main contribution of this paper is the gathered empirical evidence about the performance of two threat analysis techniques (with a different analysis scope) in the academic setting. We observed slightly better results for the STRIDE-per-element technique (SSD between the average recall of treatments, ELEMENT : 62% INTERACTION : 49%). We also observed a slightly better average productivity (no SSD, ELEMENT : 4.35 $TP/hour$ INTERACTION : 3.27 $TP/hour$). One possible explanation for the difference in treatment performance is that STRIDE-per-interaction is more difficult to perform for novice analysts [9] (such as our participants). STRIDE-per-interaction requires the consideration of pair-wise interactions of elements, thus increasing the cognitive load for the analyst [83]. Accordingly, we observed that on average the INTERACTION teams produced larger DFDs, indicating that interactions lead to participants constructing a more complex problem space. The increased cognitive load and lack of domain expertise might have affected the performance of the INTERACTION teams. This study concludes that there is no significant difference (in terms of performance) between the two treatments with a slightly different analysis scope.

1.3.3 Towards Security Threats That Matter (Paper C)

This paper is motivated by the need to increase efficiency of threat analysis techniques in the automotive industry. To this aim, we enlarge the analysis scope and improve the analysis procedure to focus on important assets. The proposal was inspired by STRIDE and comes as a result of numerous workshop sessions with our industrial partners that further highlighted the needs and shortcomings of existing approaches. As a collection of lessons learned, the first author synthesized the approach and validated it with an illustration.

Contributions. This paper contributes to answering **RQ1**. The main contribution of this paper is a novel risk-first threat analysis technique (eSTRIDE) with an enlarged analysis scope. We propose to prioritize threats before they are analyzed based on assets and their priorities. This requires practitioners to enrich the architectural model (i.e., build an extended DFD or eDFD) with assets, their sources, targets, security concerns and priorities, domain assumptions, communication channels, and existing security solutions. The DFD extensions are made to end-to-end user scenarios around highly prioritized assets. During the analysis procedure, such scenarios become the scope of the analysis. Finally, the approach proposes initial guidelines for handling threat explosion by reducing the problem domain before and introducing short-cuts during the analysis. The initial illustration suggests a reduced number of low-priority threats but does not provide sufficient evidence for the potential benefits of the approach.

1.3.4 STRIDE-per-el vs eSTRIDE (Paper D)

This paper is motivated by the lack of empirical evidence about sacrificing systematicity in the procedure of threat analysis for the discovery of high-priority threats. To this end, we conducted two comparative case studies with two different automotive organizations (ORG A and ORG B). The purpose of this study is to gather empirical evidence about the similarities and differences between a risk-last (STRIDE) and a risk-first (eSTRIDE, introduced in Paper C) threat analysis technique in the industrial setting. The case studies were conducted with (in total) 15 industrial practitioners. The participants of the first organization (Org A) were industrial experts, who have been trained in security or self-identify as security experts. On the other hand, the participants of the second organization (Org B) had a deeper knowledge of the system under analysis but self-identify as security novices. This enabled further observations about the effect of security expertise on the overall team performance. Within each organization, we observed and compared two teams analyzing the same system using one of the prescribed techniques (Stride and eStride assigned treatment). The participants were tasked to a) build a DFD (or an eDFD) of the system under analysis and b) analyze the diagram using the procedural guidelines of the prescribed technique. On the first day the teams were given a training session including hands-on exercises of the prescribed technique. On the second and third day the teams worked on their tasks. We measured differences in the quality of analysis outcomes by assessing handed-in reports of the identified threats. Differences in technique execution were measured by analyzing recordings (only allowed in Org A), time-keeping of participant activities, and structured note-taking.

Contributions. Paper D contributes to answering **RQ1**. The main contribution of this paper is the gathered empirical evidence about the performance of two threat analysis techniques (with a different analysis scope) in the industrial setting. We recorded similar levels of productivity between the compared techniques. Possibly, the eStride teams spent more time to extend the diagram, while the Stride teams spent more time to prioritize the threats at the end (this activity is skipped in eStride). Though no evidence suggests an early discovery of high-priority threats, the eStride teams found twice as many high-priority threats (compared to the Stride teams). Only a part of the discovered threats were common threats, therefore we observed that eStride tends to result in a more complete account of high-priority threats. As expected, on the first day all the teams focused on building the diagram, while on the second day they were analyzing the diagram. In Org A, we also observed that domain assumptions played an important role in the analysis (e.g., they used assumptions to justify a threat existence). Finally, we studied the effect of security expertise on the technique outcomes and execution. First, compared to Org A (more security expertise), both teams in Org B made mistakes. However, the achieved precision of the less security expert teams is still quite high (80% and 70%). In addition, the teams in Org B were more productive (about 6 correct threats per hour vs about 3). Clearly, higher productivity does not imply identification of more high-priority threats. In fact, our results show that more experienced analysts identify a bigger percentage of high-priority threats (regardless of the technique used). Regarding the differences in technique execution, we mention that the less experienced teams (Org B) seldom discussed threat feasibility.

1.3.5 Flaws in Flows (Paper E)

Paper E is motivated by the low recall of existing techniques using informal design notations, such as STRIDE [10]. On the one hand, literature describes formalizations of DFDs [84] which often result in a complicated language hindering their usability. On the other hand, several studies propose threat analysis automation (e.g., by means of pattern matching [44, 45]) with no correctness or completeness guarantees of analysis outcomes. Inspired by language-based information flow security [85,86], we propose a formal approach to analyze security information flow policies at the level of the design model.

Contributions. This paper contributes towards answering **RQ2**. The main contributions are two-fold: (i) a light-weight extension of the modeling capabilities of DFDs, and (ii) a tool-supported, formally-based flow analysis technique. The extension of the DFD notation requires the designer to provide the intended security policy for system assets. In addition, the designer is required to specify an abstract input-output security contract for the computational nodes (i.e., DFD processes). The designer also specifies a global security policy for all system assets, based on which the design flaws are identified. The additional information mentioned above is leveraged in the analysis procedure. The second contribution of this work is a formally-based flow analysis technique that propagates security labels across the design model. The approach was implemented and packaged as a publicly available plug-in for Eclipse. We validated the approach using 4 open source applications.

1.3.6 Detection of Security Design Flaws (Papers F & G)

Beyond low-level vulnerability databases (e.g., CVE [87], CWE [88], CAPEC [89]) there is little systematized knowledge on security design flaws and how to find them by inspecting architectural models. In Paper F, we present a catalog of security design flaws and evaluate their manual inspection with master and doctoral students. The catalog contains a list of 19 design flaws related to issues with authentication, access control, authorization, availability of resources, integrity and confidentiality of data. Each catalog entry consists of (i) the name of the design flaw, (ii) a description, and (iii) a series of closed questions that serve as rules for manual inspection. Existing literature already contributes towards automating various security design analyses [43,44,57,65,90–92], yet there is a lack of automated reasoning for detecting security design flaws alongside existing system defenses and security assumptions about assets. Therefore, we select five security design flaws from the catalog (introduced in Paper F) and attempt to automate their detection. To that end, Paper G presents model query patterns which are executed over DFD models enriched with data types and security solutions. The query patterns were developed by translating the manual inspection rules from the catalog to model element types and their relations. Further, we conducted an empirical study (under controlled conditions) with 13 academic researchers on-site two University campuses to obtain a data set of 26 enriched DFD models. The data set was submitted to two expert assessors for a manual application of the inspection rules of the five flaws. We leverage this data set of models (incl. instances of the five flaws) to evaluate the performance of the automated detection technique.

Contributions. This work contributes towards answering **RQ2**. The main contributions of these papers are three-fold: (i) a data set of 26 security enriched models and their flaws, (ii) an automated detection technique of five security design flaws, and (iii) an empirical evaluation of the automated detection using the data set. On average, a model in our data set contains about 17 data flow elements, 5 processes, 3 data stores, 2 external entities, and 8 security solutions. The experts found (on average) about 16 flaw instances on a single model. In our data set, the model size seems to correlate with the average number of identified flaws. The model extensions are leveraged in the automated detection, which was implemented as an Eclipse plug-in. First, the data types are used to identify locations in the model where a flaw could be present. At those location, the automated detection checks for the presence of appropriate security solutions. We compared the expert reports of the identified flaws to the flaws detected by the query patterns. Though the precision of the automated technique is too low to replace expert analyses, it could be used to present a list of issues for the analyst to sieve through.

1.3.7 Structural Compliance (Paper H)

This paper is motivated by the difficulty of establishing and maintaining a software implementation compliant to the intended design. The disconnect between design-level models and their implementation may potentially result in architectural erosion [93]. To address this issue, we present a user-in-the-loop approach for establishing an explicit mapping between DFD models and implemented constructs (concretely, in Java). Our goal is to enable an automated discovery of compliance with minimal user interaction. The proposed approach relies on a set of four correspondence rules between the DFD element types (introduced in Paper E) and the program model element types [13]. These rules are used in a heuristic search for mapping suggestions. The approach includes a user interface and is packaged as a publicly available Eclipse Plug-in. Finally, we experimentally evaluate the precision and recall of the suggested mappings on five open source projects.

Contributions. This paper contributes towards answering **RQ3**. The main contributions of this paper are two-fold: (i) an automated technique for suggesting mappings between DFDs and program models, and (ii) an implementation of the approach as a publicly available Eclipse plug-in, evaluated on five open source Java projects. First, the approach takes as input a completed DFD model. Second, the user needs to extract a program model from the implementation (which is done automatically) [13]. Finally, she can map the DFD to the desired Java project (in the Eclipse workspace). The user can accept, reject, or tolerate the suggested mappings, and execute a new search iteration until she deems the DFD is mapped. She can also manually map source code elements (provided they respect the correspondence rules). Information and error markers are used to provide feedback to the user about the state of compliance. We measured the correctness (in terms of precision and recall) of the suggested mappings and the user impact on the correctness of mappings for each iteration.

1.3.8 Security Compliance (Paper I)

This paper is motivated by the need for automating the verification of implemented programs with respect to the intended security properties in the design. In addition, static analysis tools may report violations which are afterwards labeled by human experts as false alarms [94]. All reported violations have to be manually sieved through, and, more importantly, the actual violations must be distinguished from the false alarms, which is hard for developers with less security expertise [95]. Our goal is to leverage the previously proposed technique (introduced in Paper H) to statically analyze the implemented security properties against the mapped design model. To this end, we propose to extend the approach in Paper H with automated security compliance checks.

Contributions. Paper I contributes towards answering **RQ3**. This work extends the approach in Paper H with two key contributions: (i) two types of static checks to verify security properties (i.e., SecDFD contracts) in the implementation, and (ii) an automated extraction of project-specific sources and sinks of confidential information from the design, which are used to reduce the number of false alarms raised by a state-of-the-art data flow analyzer. In Paper I, we assume an existing SecDFD and its correct mapping (following the steps in Paper H). To verify security properties of the design, we introduce two types of static checks: rule-based checks for the encrypt and decrypt (SecDFD) contracts, and local data flow checks for the forward and join contracts. The second contribution is the extraction of project-specific sources and sinks of confidential information from the design and their localization in the code. For each SecDFD asset, we execute an existing data flow analyzer [96] initialized with the extracted sources and sinks. We experimentally evaluate both contributions with two open source Java projects.

1.4 Discussion

This section summarizes the answers to the main research questions of this thesis. First, we discuss the main findings of the study of the state-of-the-art (Paper A).

Our assessments in Paper A show that existing threat analysis techniques are mainly applicable on the level of requirements, architecture and design. This is not very surprising considering that one of the main purposes for performing threat analysis is to elicit security requirements. Further, most of the studied techniques use architectural design models and requirements (usually in textual form) as inputs to the analysis procedure, which is in line with the first finding. Interestingly, the precision of most threat analysis procedures is based on templates and examples, such as textual descriptions of example threats. About half of the studied techniques consider risk to prioritize the analysis outcomes. The analysis outcomes of the studied techniques in turn are mostly threats. Yet, half of the techniques also produce security mitigations or requirements. Finally, we find that not many of the studied techniques have a way to assure the quality of analysis outcomes.

Paper A also investigated the ease of adoption for the studied techniques. About half of the studied techniques do not provide any tool support. The target audience for most of the studied techniques are security experts and security trained engineers. We contemplate which characteristics are important for adopting the techniques in practice and provide the following guidelines for technique selection:

(a) If the organization plans to make *small* investments into adopting a threat analysis technique and security is *not prioritized* by management, we recommend selecting a technique that can be used without further modifications. Important criteria: Tool availability and maturity, sufficient documentation, low target audience and a light-weight analysis procedure.

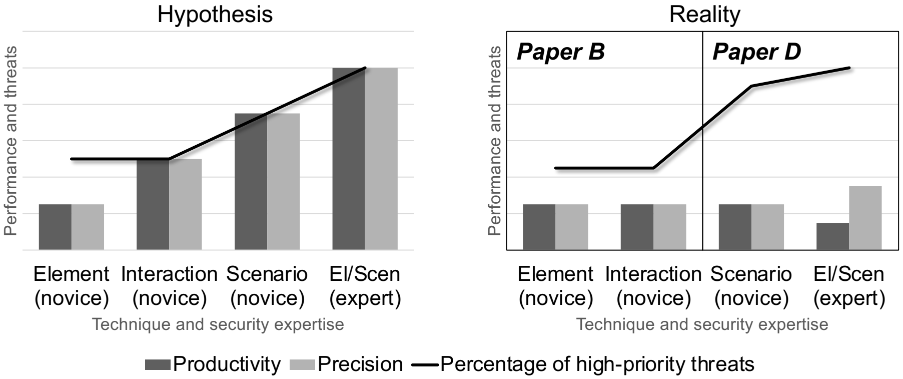


Figure 1.2: Illustration of our hypothesis about technique performance in relation to the cognitive load required for brainstorming threats and security expertise (left) and the reality for three specific techniques (right)

(b) If the organization plans to make *small* investments into adopting a threat analysis technique and security is *prioritized* by management, we recommend selecting a technique that is systematic. Important criteria: Systematic analysis procedure, expert-based and preferably semi-automated.

(c) If the organization plans to make *large* investments into adopting a threat analysis technique, we recommend developing an "in-house" adaption of a promising technique. Important criteria: Systematic analysis procedure, potential for improvement (e.g., technology improvement).

RQ1. What are the effects of broadening the analysis scope on the quality of analysis outcomes?

Among other, Papers B and D investigate the effects of enlarging the analysis scope on technique performance. Figure 1.2 illustrates our hypothesis and observed reality about the linear dependency between technique performance and the analysis scope. In addition, it shows our expectations (left) and observations (right) about performance differences between less experienced groups in security and groups with security experts.

Empirical evidence shows that the productivity, precision, and number of high-priority threats found are not significantly different for the per-element and per-interaction variants of STRIDE. Thus, in the context of the controlled experiments reported in Paper B, the analysis scope does not have a significant effect on the overall technique performance. Further, in the context of the case studies reported in Paper D, we find that enlarging the analysis scope to a chain of elements (like in eSTRIDE) does not affect the overall technique performance either, therefore similar levels of outcomes quality can be assumed. However, the eSTRIDE technique leads to finding twice as many high-priority threats (compared to STRIDE-per-element). Contrary to our intuition, the productivity of expert analysts is lower (about 3 TP per hour) compared to novice teams (about 6 TP per hour). However, our results show that more experienced analysts identify a bigger percentage of high-priority threats (regardless of the technique used). In addition, the precision of the expert groups is still higher, compared to novice groups.

RQ1.1. What changes are required in the design model to facilitate a threat analysis focusing on important threats?

Reasoning about risk early-on requires a good understanding of the assets and their whereabouts in the system. During the asset analysis, the assets first need to be identified in the model (incl. asset source, target(s)). The importance of assets can only be deduced by discussing their security objectives (i.e., confidentiality, integrity, availability, accountability) and the priorities of those (high, medium, low). The annotated assets are required in the model to indicate where the model should be further extended. By focusing on highly prioritized assets, the analysis is performed on parts of the architecture. This is how the problem space is reduced *before* the analysis begins. Domain assumptions, communication channels, and existing security solutions are notation extensions that are used to make reductions during the analysis.

RQ1.2. What changes are required for a model-based threat analysis procedure to focus on important threats?

In Paper C, we provide guidelines for handling threat explosion before (see RQ1.1) and during the analysis. To reduce the effort *during* threat analysis, we propose a slight departure from the analysis procedure suggested by STRIDE. First, eSTRIDE analysis is performed using eDFDs. Second, threats are only elicited for scenarios containing high-priority assets. Third, the scope of the analysis is an end-to-end scenario of an asset with important security objectives. This means that a chain of elements is considered during threat elicitation, rather than single elements (or their interactions). Further, only threats that directly threaten a highly prioritized security objective are considered. For instance, tampering threats compromise the integrity objective. Finally, only threats that are possible despite annotated domain assumptions and existing security solutions are considered.

RQ1.3. What is the difference (in terms of performance and execution) between a risk-first and risk-last threat analysis technique?

In Paper D, we observe similar productivity levels across the two treatments (Stride vs eStride). One possible explanation is that, instead of spending time on prioritizing threats at the end (in Stride), the analysts of the eStride teams had to spend time on extending the diagram. This is reflected in the productivity of eStride teams. However, in reality these sessions can span over weeks, therefore the additional security information in the model could help to reboot the discussions after more time has passed (though this was not measured in Paper D). We also observe that many high-priority threats are found around trust boundaries. Trust boundaries illustrate locations in the diagram where entities with different privileges interact [9]. It would be interesting to observe the timeliness of discovering high-priority threats if these boundaries are analyzed first. In addition, we find that discussing feasibility of threats is time-consuming, but is required for a precise analysis. Indeed, the less experienced teams seldom discussed threat feasibility in detail, were more productive, but performed a less precise analysis. Regardless of the security expertise, our teams were able to quickly learn and effectively apply both techniques. Therefore, we postulate that security expertise may be traded for a higher paced and less precise analysis under resource-constrained conditions.

The main findings regarding RQ1 are summarized in what follows.

RQ1. Summary of main findings

☞ The domain and security knowledge of the team has an impact on the quality of outcomes and needs to be present in the architectural model *before* the analysis begins. (Paper C)

☞ The complexity of the architectural model needs to be managed by making model abstractions wherever possible while enrichments are made only around assets with highest priorities. (Paper C)

☞ During the analysis, only threats to scenarios with high-priority assets should be elicited. In addition, only threats (that exist despite domain assumptions and security solutions) to high-priority objectives of assets should be considered. (Paper C)

☞ Similar performance (in terms of productivity and precision) is measured for the risk-first and risk-last threat analysis technique. (Paper D)

☞ Compared to a risk-last technique, the risk-first technique tends to discover twice as many high-priority threats. (Paper D)

☞ In the industrial setting, security expertise may be traded for a faster-paced and less precise threat analysis. (Paper D)

RQ2. To what extent can security design flaws be automatically detected in DFD-like models?

We propose two extensions of the regular DFD notation to automate the detection of the security design flaws. The first is a semantically enriched specification language (SecDFD) coupled with a formally-grounded model

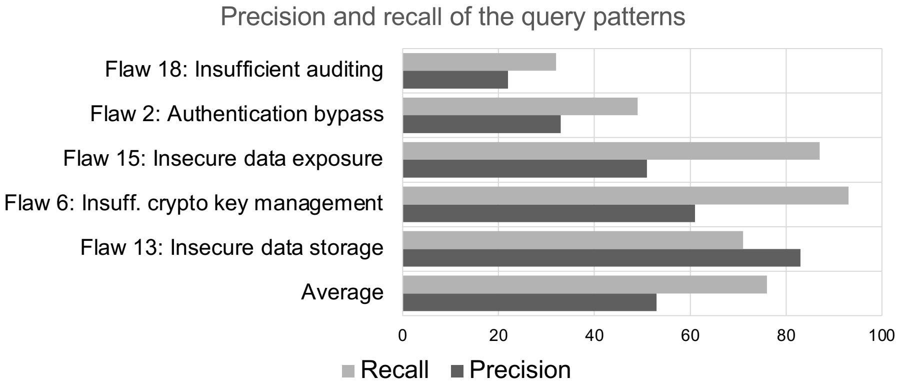


Figure 1.3: Precision and recall of the automated detection of five security design flaws (Paper G)

analysis (Paper E). Preliminary evaluation on four open source applications suggests that automating the detection of data leaks between confidential sources and public sinks can be automated without false positives. In the second proposal, we extend the notation with data types and leverage an existing extension to model security solutions [42]. Figure 1.3 shows the average precision and recall of the developed query patterns. Our key take-away is that it is very hard to attain good performance when automating the inspection rules of the catalog, though the higher recall (compared to precision) is still encouraging.

RQ2.1. What model extensions support an automated security design flaw detection?

In order to reason about confidentiality design flaws, we introduce the Security Data Flow Diagram (in short, SecDFD) specification language. First, the regular DFD notation is extended with confidentiality labels of assets, their sources and sinks. In addition, the notation is extended with attacker zones. Attacker zones are sets of elements where the attacker may be able to observe assets. The above extensions enable the definition of a global security policy, i.e., the model is considered secure if and only if there is no sensitive asset flowing into observable model locations. Second, the security properties of assets are subject to change in the diagram. To capture this, one must consider how each process affects the confidentiality of a traveling asset. To this aim, the regular DFD notation is enriched with a formal label model with propagation functions (or security contracts). The label model defines the semantics of four different security contracts (i.e., forward, join, encrypt, and decrypt), depending on the process (or node) type. Given a SecDFD model, we are able to propagate the confidentiality labels by visiting each node (in a depth-first manner) and spot locations where data may leak.

In Papers F and G, we study the detection of security design flaws concerning various security properties. Compared to Paper E, our aim is to develop the detection of several flaw types with less formal guarantees. The inspection rules of five security design flaws (from the catalog introduced in Paper F) instruct the analyst to identify sensitive information in the model, and to evaluate the existence of security mechanisms. Therefore, the regular DFD notation is extended with a data model, which enables representing different types of sensitive data (such as key material, session token, encrypted data, etc.). In addition, our data model allows specifying data transformations (e.g., encryption of a sensitive asset). To analyze security flaws in the context of existing system defenses, we leverage an existing DFD modeling extension by Sion et al. [42]. The benefit of this extension is that the meta-model allows specifying customized solutions and types of threats (e.g., spoofing) they mitigate. Instances of data types and security solutions are bound to concrete DFD model elements. These extensions enable finding weak spots with respect to existing defenses and assets of value by querying the graph-like model for problematic patterns.

RQ2.2. What performance can be achieved by an automated technique for security design flaw detection?

Paper G empirically compares the outcomes of the developed technique for automated flaw detection to a manual inspection (performed by human experts). The overall average precision of the automated technique is about 50% and the average recall is about 75% (see Figure 1.3). In the context of our performance evaluation, the precision of our automated detection is not good enough to replace manual expert analyses. Among other reasons, falsely detected flaws have two origins: (i) over-approximated asset sensitivity levels, and (ii) ambiguously modeled solutions. The expert assessors disregarded incorrectly modeled sensitive assets (i.e., if the experts considered the asset not sensitive, they did not report a flaw despite the incorrect model). Further, in case of minor mistakes or ambiguities the experts took modeler intention into account and did not report a flaw. In comparison, the automated detection technique assumes that the model is correct, therefore in such situations the flaws are still detected by the tool. The higher value of recall (compared to the precision) is still encouraging, as the automated technique could generate a list of issues for the expert to sieve through. The second take home message is that some rules seem to be more promising than others for automation. For example, the query patterns for design flaws 13 and 15 (design flaws only affecting the confidentiality of assets) perform somewhat better.

Below we summarize the main findings for what concerns RQ2.

RQ2. Summary of main findings

- ☞ The design notation needs to be extended with node types to specify the operations that the process elements perform over the assets. (Paper E)
- ☞ The global security policy (i.e., the security condition in Paper E) and the attacker model need to be defined to reason about analysis completeness. (Paper E)
- ☞ The semantics of the node types need to be defined for a formally-based analysis (e.g., security contracts of node types). (Paper E)
- ☞ Graph-based queries seem promising for automating the detection of security design flaw inspection rules. (Papers F & G)
- ☞ It is hard to attain good performance (precision and recall) when automating the rules for manually inspecting security design flaws. (Paper G)
- ☞ Some security design flaws are more amenable to automation, and the overall higher recall (compared to precision) of the automated detection is still encouraging. (Paper G)

RQ3. What security code analysis techniques can be leveraged to discover the security compliance of the implemented system to SecDFD models?

Figure 1.4 shows the steps (from the user perspective) of the proposed approach for analyzing the security compliance between the intended design and its implementation. First, to automate structural compliance checks, we propose (Paper H) to establish a mapping between the high-level model and the implementation. The user is intentionally kept in the loop to make the compliance analysis meaningful. Our evaluation shows that the precision and recall of the automated mappings suggestion progresses with every iteration,

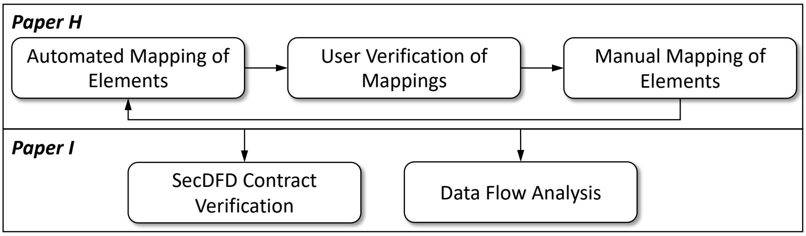

Paper H
Automated Mapping of Elements
User Verification of Mappings
Manual Mapping of Elements
Paper I
SecDFD Contract Verification
Data Flow Analysis

Figure 1.4: The steps of the iterative approach for analyzing structural compliance (Paper H) and the security compliance analysis steps (Paper I)

demonstrating that (i) our heuristics are able to provide useful suggestions for mappings, and (ii) the search for mappings takes user input (e.g., rejecting a mapping or manually adding a mapping) into account. Given the technique proposed in Paper H, static code analysis can be used to develop security compliance checks. In particular, we present rule-based checks and local data flow checks to verify whether the SecDFD security contracts are implemented as intended. In addition, we show that the security information in the intended (and mapped) design can be leveraged to improve code-level analysis tools by reducing the number of reported false positives.

RQ3.1. What relation between the DFD model and an intermediate code representation supports automated security compliance checks?

In Paper H, we present an iterative, user-in-the-loop approach for analyzing the compliance between the intended design and implementation. The approach is based on establishing mappings between a design-level model and the program model (extracted from the implemented system in Java [13]). A set of four rules is used to pin-down the corresponding elements between the two abstractions. For instance, assets can be mapped to Java types (e.g., a class or a primitive type). The rationale for this rule is that assets hold data, which (in the implementation) is typically transmitted using parameters and return values. The only property of assets which rarely changes in the implementation is their type.

Our approach consists of three steps: the automated suggestion of mappings, user mappings verification, and manual mappings creation (see Figure 1.4). To find and present meaningful mappings to the user, our algorithm heuristically assigns scores to all possible mappings. We implemented simple name matching heuristics (using the Levenshtein distance) and structural heuristics. The Levenshtein distance [97] is a measure of the minimal number of characters which have to be removed, added or flipped to change one word into another. For instance, the Levenshtein distance is used to score similarity between DFD process names and method names. In one of the structural heuristics, we score concrete method signatures by comparing incoming parameter types and return types to incoming and outgoing DFD assets (of a process to be mapped). In the second step, the user verifies suggested mappings via the tool interface by accepting, rejecting, or tolerating them. Finally, the user is able to manually add mappings. After the user has finished defining the mappings, static checks can be used to determine structural compliance. All accepted or manually added (but not rejected) mappings are allowed and are thus convergences. Elements present in the code, but not specified in the DFD represent a divergence. It is possible to display flows from process-mapped members to other members which have not been mapped to this process. If the target of such a flow has not been mapped to any process, there seems to be a divergence. A divergence also arises if there is a flow between two processes in the code that has not been specified on the DFD. Finally, if a DFD element has not been mapped to any program model element, the user can discover an absence of the specified functionality in the code.

Though the precision of the first round of suggested mappings is on average about 50%, the last automated suggestion phase reaches an acceptable precision of almost 90%. Similarly, the recall progresses with every iteration, which suggests that the search for mappings takes user input (i.e., rejecting or manually adding a mapping) into account.

RQ3.2. What security code analysis techniques can be leveraged to discover security compliance to the node contracts specified in the SecDFD?

In Paper I, we build on top of our work on structural compliance and study the security compliance between the intended security and implemented security. First, we introduce rule-based checks to verify that the implementation complies with the indented cryptographic process contracts (i.e., the SecDFD encrypt and decrypt contracts). In essence, for each SecDFD process with such a contract, the check will inspect the mapped source code, and verify whether there exists a call to at least one method with a method signature predefined to be used for cryptographic operations. We also develop checks to verify that the implementation complies with the intended data processing contracts (i.e., the SecDFD forward and join contracts). On the level of the program model, implemented data flows can be traced trough incoming parameter and return flows. For each SecDFD process with such a contract, we extract the relevant implemented flows from the program model and compare them to the expected flows (according to the SecDFD) to find a potential match. In addition, we leverage the security information from the design model to initialize and execute a state-of-the-art data flow analyzer for Java programs (i.e., FlowDroid [96]).

We consider the security compliance to converge when a planned security contract (of the SecDFD process) is implemented at the correct location and no leaks have been detected by the data flow analyzer. Instead, divergence is identified if (i) there exists an implemented data flow which does not comply with the security contracts (of the SecDFD process), or (ii) a leak has been detected by the data flow analyzer. Finally, absence is identified when a SecDFD contract is not implemented.

From our evaluation we conclude that the two developed types of security compliance checks are relatively precise (average precision is 79.6% and 100%) but may still overlook some implemented information flows (average recall is 65.5% and 94.5%) due to the large gap between the design and implementation.

RQ3.3. What information from the SecDFD complements existing static code analysis tools?

We study how security information present in the design models can be used to complement code-level analysis. First, we use our mappings to extract the locations of confidential sources in the code. For instance, if the asset source is an external entity and it is mapped to method definitions, their signatures are collected as sources. We maintain the list of source method signatures for each confidential asset (as they may differ across assets). Second, we use a baseline list of sinks [79], which we modify before executing the analyzer. Similar to source extraction, for each confidential asset we are able to identify sinks where the asset is allowed to flow (by design). The allowed sinks are then removed from the baseline list of sinks [79]. Finally, mapped method signatures of elements contained in attacker zones (in the SecDFD) are added to the list of sinks.

The key takeaway from our evaluation is that using this approach we were able to extract project-specific sources and allowed sinks of confidential data, and reduce the number of false alarms (up to 62 %) raised by the state-of-the-art data flow analyzer.

The main findings regarding RQ3 are summarized below.

RQ3. Summary of main findings

☞ A semi-automated, user-in-the-loop approach is promising for establishing the mappings between a design model and its implementation. (Paper H)

☞ The performance of the heuristic search for mappings is less optimal with no user input (i.e., in the first iteration), however both precision and recall increase in the following iterations, reaching fairly good levels (e.g., on average the precision of the final automated phase is 87.2% and recall is 92%). (Paper H)

☞ Given an existing mapping, static analysis techniques can be used to develop security compliance checks with a fairly good precision (e.g., average precision for the two type of developed checks is 79.6% and 100%). (Paper I)

☞ Our approach is able to extract additional project-specific sources and allowed sinks of confidential information in the code. (Paper I)

☞ The security information in the intended (and mapped) design can be leveraged to help code-level analysis tools by reducing the number of reported false positives. (Paper I)

1.5 Conclusion and Future Work

This thesis addresses three research problems, which were identified by conducting a systematic analysis of the state-of-the-art in threat analysis of software systems. To address the issue of high manual effort, we propose a notation extended with security-relevant information (eDFD) and an improved analysis procedure (eSTRIDE). Second, we study how to raise the recall of model-based security analysis techniques. To this aim, we introduce two approaches for automatically detecting security design flaws: the SecDFD and a graph-based automated detection. Finally, we suggest an approach for automating the security compliance checks of the implemented programs with respect to the intended design (represented with SecDFDs). We envision two future directions.

Extensions to privacy threat modeling. Fueled by changes in the legislation (GDPR), privacy threat modeling has been receiving more attention in academia and industry. But the gap between actual system behavior and the high-level notions of the GDPR is immense. To overcome this issue, design-level analyses could be adopted. Recent work by Antignac et al. [98] introduces a set of privacy preserving transformations to statically identify and mitigate so called "privacy hotspots" in DFDs. For instance, personal data flowing into a third-party component (external entity) represents information disclosure, and thus a potential breach of privacy. The privacy transformations modify such interactions by inserting a pattern of new DFD elements to ensure that the necessary steps will be taken at the time of implementation. But, GDPR requires a more fine-grained tracking of data processing operations. We are curious to study how our formally-based approach for detecting confidentiality flaws (Paper E) can be extended with a privacy analysis. In particular, we are eager to understand what data processing operations can be expressed for DFD processes, and how these operations affect privacy properties of data classes.

Applications to the Internet of things (IoT) domain. In the domain of IoT, security and privacy properties are hard to enforce due to hardware constraints in the devices, and their access to private data. We are working on applying the formally-based analysis of confidentiality (and integrity) flaws (Paper E) in the context of IoT applications. In particular, we are interested to leverage static code analysis techniques to verify implemented security properties. Analyzing the source code statically (for every possible input) can be resource demanding. Therefore, we are looking into the possibility to leverage the compositionality property of the SecDFD specification language. First, we intend to extract DFD-like graphs from existing IoT applications. Intuitively, static code analysis could be performed over the implementation of local application nodes to extract the implemented data flows. Next, the global security policy could be verified by leveraging our label propagation model. To validate our approach, we are studying a flow-based programming platform (i.e., NodeRED [99]) and the accompanying repository of IoT applications.

Chapter 2

Paper A

This chapter is based on
Threat Analysis of Software Systems: A Systematic Literature Review,

written by
K. Tuma, G. Calikli, and R. Scandariato,

published in
Journal of Systems and Software (2018), 2018.

Abstract

Architectural threat analysis has become an important cornerstone for organizations concerned with developing secure software. Due to the large number of existing techniques it is becoming more challenging for practitioners to select an appropriate threat analysis technique. Therefore, we conducted a systematic literature review (SLR) of the existing techniques for threat analysis. In our study we compare 26 methodologies for what concerns their applicability, characteristics of the required input for analysis, characteristics of analysis procedure, characteristics of analysis outcomes and ease of adoption. We also provide insight into the obstacles for adopting the existing approaches and discuss the current state of their adoption in software engineering trends (e.g. Agile, DevOps, etc.). As a summary of our findings we have observed that: the analysis procedure is not precisely defined, there is a lack of quality assurance of analysis outcomes and tool support and validation are limited.

2.1 Introduction

After decades of research the issue of integrating security early-on in the Software Development Life-cycle (SDL) has received more attention and is becoming a corner stone in software development. In this respect, architectural threat analysis plays a major role in holistically addressing security issues in software development. Threat analysis includes activities which help to identify, analyze and prioritize potential security and privacy threats to a software system and the information it handles. A threat analysis technique consists of a systematic analysis of the attacker's profile, vis-a-vis the assets of value to the organization. Such activities often take place in the design phase and are repeated later on during the product life-cycle, if necessary. The main purpose for performing threat analysis is to identify and mitigate potential risks by means of eliciting or refining security requirements. Threat analysis is particularly important, since many security vulnerabilities are caused due to architectural design flaws. A failure to consider security early-on can be a cause for so-called Architectural Technical Debt (ATD) [100]. Furthermore, fixing such vulnerabilities after implementation is very costly and requires workarounds which sometimes increase the attack surface.

Building Security In Maturity Model (BSIMM)[1] collects statistics from 95 companies and gauge their level of adoption with respect to several secure software development practices. According to this technical report [101], security-specific code analysis techniques have successfully found their way into the industrial practice, as two thirds of the surveyed companies routinely adopt them. However, it is a bit discouraging to find that only one third of the companies adopt architectural threat analysis. One possible explanation for that is the lack of automation support of threat analysis, since available tools require extensive human interaction for efficient use [44, 45, 102]. Another possible explanation is the lack of an industry-standard technique for threat analysis. In comparison with safety analysis techniques (failure analysis), threat analysis have yet to mature in this area [103]. This paper attempts to understand the potential road blocks to a wider adoption of threat analysis techniques by systematically studying the existing methods.

Recently a limited and compendiary review of threat analysis techniques has emerged [104] in a form of a short technical report, yet this review only describes a handful of approaches. To the best of our knowledge, this is the first systematic and complete review of the state of the art. We have analyzed 38 primary studies for a total of 26 threat analysis techniques. With this study we aim at providing information to the practitioners about the extent the existing threats analysis techniques are applicable to their needs. Providing such information to practitioners might facilitate active usage of the aforementioned techniques and in the long-term cause techniques to mature. The contributions of this work are threefold:

(i) We systematically analyze the existing literature and identify gaps for future research,

(ii) we provide insight into the obstacles for adopting the existing approaches in practice and how these obstacles could be overcome,

[1] https://www.bsimm.com

(iii) we provide insight into the adoption of the threat analysis techniques in software engineering trends (i.e. DevOps, Agile development, IoT and automotive).

The rest of the paper is organized as follows. Section 2.2 describes the research methodology, including the research questions and data extraction strategy. Section 2.3 presents the results, while Section 2.4 discusses them. The threats to validity are listed in Section 2.5. Section 2.6 discusses the related work and Section 2.7 presents the concluding remarks.

2.2 Research methodology

We conducted our research by adopting the systematic literature review method. By following the steps introduced by Kitchenham et al. [80], we collected and analyzed the literature. According to the guidelines, our study consisted of three main steps: planning, conducting and documenting the review. The SLR was motivated by the need to strengthen security engineering practices in the SDL, desired both by academia and industry. We searched for similar studies in the ACM, IEEE, Google Scholar and Scopus digital libraries (November 2016), to establish whether an SLR about threat analysis techniques was previously conducted. None of the mentioned digital libraries contained an SLR about threat analysis techniques, reaching the same goals and objectives.

2.2.1 Research questions

The initial goal of this study is to catalog and characterize the existing threat analysis techniques. Thereafter, the second goal of our work is to provide future directions and to address how the techniques can be used by practitioners including their adoption to the latest software engineering trends. To this end, a critical analysis of the selected literature was performed answering three main research questions, which are reflected in the assessment criteria, presented in Tables 2.2, 2.4 and 2.5.

RQ1: What are the main characteristics of the identified techniques? We have organized the first research question into four inquiries (refer to Table 2.2 for more details).

Applicability (RQ1.1). What level of abstraction is the threat analysis technique applicable to? Threat analysis can be conducted on projects, where little is known about the actual system in the early design stages. However, systems are sometimes also analyzed for threats later-on in the SDL. For instance, integration of new units in a code-base may require a threat analysis of the effected components. Therefore, such an analysis might be performed on a low-level of abstraction (e.g. static code analysis).

Input (RQ1.2). What information do the identified techniques require as input? This question refers to the information about the system that is required in order to execute the analysis. In particular, it aims at identifying the *type* and the *representation* of required information for executing the analysis. This information helps researchers and practitioners to determine which approaches can be adopted according to the available software artefacts.

Procedure (RQ1.3). What kind of activities are part of the analysis procedure of the identified techniques? This research question aims at determining how the input is transformed to obtain the desired outcomes of the analysis. Most threat analysis techniques, such as STRIDE [9,105], CLASP [106], OCTAVE [107], etc. require expert knowledge for execution. However, some methods are supported by catalogs of security threats, which aid the identification of threats by providing contextual examples. This study considers such techniques as knowledge-based. Furthermore, we observe the level of precision of threat analysis procedures. A higher precision may increase the quality of the analysis and provide opportunities for security compliance. Commonly, the technique documentation includes descriptive guidelines for analysis execution, yet no clear definition is given for when the procedure ends. As part of this research question we also investigate how the proposed techniques determine when the analysis should stop (i.e. Definition of Done).

Finally, we also observe which security concerns are accounted for and to what extent is risk assessment present in the analysis procedure.

Outcomes (RQ1.4). What information is gained by the outcomes of the identified techniques? This question intends to qualify the added value of adopting a technique. The main purpose for investigating the outcomes is to indicate what kind of results can be expected from the studied approaches. Among others, we assess the granularity of outcomes as well as the available quality assurance of outcomes.

RQ2: What is the ease of adoption of the identified techniques? Our second research question is motivated from a more practical perspective (see Table 2.4). It aims to determine the challenges of adopting the studied approaches in practice. This work refers to *ease of adoption* as a broader term compared to "usability". First, tool availability is a strong indicator of technique maturity. Unfortunately, fully stand alone tools are less common compared to prototype tools or tool extensions. Second, practitioners benefit from a complementary guidance for execution. The guidance could provide fine- or coarse-grained instructions for using the proposed tool in combination with the theoretical concepts of the approach. Third, tools are typically accompanied by tool documentation. We also investigate whether there are other sources of technique documentation available (e.g. demonstrations). Finally, the ease of adopting the studied approaches is also dependent on the required knowledge and skill set of the analyst. For instance, approaches that require extensive education in formal methods will be difficult to use for software engineers without additional training. Likewise, manual approaches typically require domain knowledge and knowledge about security attacks and countermeasures. To this aim, the second research question aims to determine the target audience of the proposed approaches.

RQ3: What evidence exists that threat analysis techniques work in practice? The purpose of this question is to identify the extent of validation conducted for a technique. In addition to previously mentioned characteristics, providing evidence about a realistic application of an approach is very important to practitioners as well as academics. In the scientific community, validation has to be extensive and reproducible. Unfortunately validation is sometimes under-prioritized (as summarized in Table 2.5). First, this research question aims to determine the type of validation method used to evaluate the proposed approaches (e.g. case studies). Second, we aim to determine who performed the validation (e.g. a third party). The reader should note that we do not attempt to undermine the validation efforts contributed by the authors of the techniques. Third, this research question aims to identify the domain of validation (e.g. automotive, web based systems, etc). Validation across different domains further enables the generalizability of results. In general, extensive validation includes different validation methods across domains, preferably also performed by validators with no conflict of interest.

2.2.2 Search strategy

The search strategy included an automatic search of digital libraries using a search string validated by experts. According to the SLR guidelines and lessons learned [108], the search string is comprised of keywords grouped into four categories.

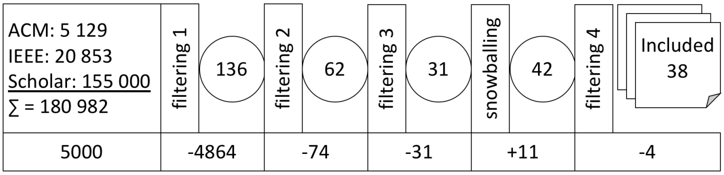
```
ACM and Scholar
(1) (secur* OR privacy) AND
(2) (abuse OR misuse OR risk OR threat* OR attack* OR flaw*) AND
(3) (analysis OR assess* OR model* OR management OR elicit*) AND
(4) (system OR software OR application)
IEEE
(1) (secur* OR privacy) AND
(2) (abuse OR misuse OR risk OR threat* OR attack* OR flaw) AND
(3) (analysis OR assess* OR model* OR elicitation) AND
(4) (system OR software OR application)
```

We conducted pilot searches in order to refine the search string. While doing so, we excluded the keywords that did not produce additional search results. Furthermore, due to additional constraints imposed by the digital libraries (IEEE Xplore), we restricted the number of keywords and "wildcard" characters (*). To this aim, we have decided to use a second, similar search string used to search within keywords, title, abstract and full text of the publications. Keywords related to security and privacy are in the first group of terms. Keywords limiting the search results to black-hat (type) of techniques are in the second group of terms. The third group of keywords specifies the activity of the target techniques. Finally, the fourth group of keywords limits the scope to software, systems and applications.

Figure 2.1 shows the adopted search method of this study. We adopt two techniques to search the existing literature in this study: (i) automatic search of digital libraries and (ii) backwards snowballing.

Digital libraries. We have obtained studies from the digital libraries using the search string. In January 2017, ACM returned 5129 titles, IEEE Xplore 20853 and Google Scholar 155000 search results. The search results were ordered by relevance and cut to top 2000 for ACM and IEEE and to top 1000 for Google Scholar, resulting in a total of 5000 search results. We then proceeded to filter the search results in several steps as shown in Figure 2.1.

The CORE [109] ranking portal was used for assessing the rank for both conference and journal venues. The portal provides two separate search interfaces,



filtering 1	Considering venue rank and applying the I/E criteria to title and keywords.
filtering 2	Applying the I/E criteria to the abstract and conclusions.
filtering 3	Applying the I/E criteria to the entire paper.
filtering 4	Applying the I/E criteria to the entire paper (obtained from snowballing).

Figure 2.1: Search method used in this study. The digital libraries were queried in January 2017.

one for conference venues [2] and one for journal venues .[3] Journal venues are ranked based on the ERA ranking process [110]. The ranking assigns conference and journal venues into the following categories: (i) *A** - leading venues in a discipline area, (ii) *A* - highly respected venues, (iii) *B* - good venues, (iv) *C* - venues meeting the minimum standards, and (v) *Unranked* - insufficient quality data has been provided to determine the ranking. In the first filtering step (filtering 1) the publications presented at a venue with CORE rank below B were excluded. The publications that were presented at an unranked venue required further investigation for exclusion. The inclusion and exclusion criteria (Table 2.1) were manually applied to the title and keywords. After this step, the amount of search results considerably decreased to 136.

In the second filtering step (filtering 2) the inclusion and exclusion criteria (Table 2.1) were applied to the abstract and conclusion sections of the 136 remaining publications. After this step, the amount of search results decreased to 62.

Finally, 62 papers were read entirely. In the third filtering step (filtering 3) the inclusion and exclusion criteria were applied to the entire paper, which resulted in the exclusion of 31 papers. After this step, the amount of search results decreased to 31.

Snowballing. We have also performed the backward snowballing search technique [81]. Essentially, this involves repeating the entire search strategy on the referenced work of a final set of papers. In our case, snowballing was performed on 31 papers. In the fourth filtering step (filtering 4) the inclusion and exclusion criteria were applied to the entire paper obtained by backwards snowballing. After this step, the amount of search results increased to 38, leading to the final primary studies.

[2] http://portal.core.edu.au/conf-ranks/
[3] http://portal.core.edu.au/jnl-ranks/

Table 2.1: Inclusion and exclusion criteria.

	Inclusion criteria
1.	Primary studies
2.	Studies (i.e. papers) that address methodologies, methods or techniques for identifying, prioritizing and analyzing security threats to a system including a software component.
3.	Studies that relate to software design.
4.	Studies that relate to security or privacy of software related systems.
	Exclusion criteria
1.	Studies written in any language other than the English language.
2.	Short publications and posters (< 3 pages).
3.	Publications at venues with a CORE rank below B (explained in Section 2.2.2).
4.	Publications that were unavailable through the search engine.
5.	Studies that focus on concrete mitigation strategies, security solutions, taxonomies of security threats and security analysis of systems.
6.	Studies that focus on anomaly detection and intrusion detection systems.
7.	Publications about safety-hazard analysis and detection methods and studies investigating the relationships between safety and security requirements.

2.2.3 Inclusion and exclusion criteria

Table 2.1 presents the summarized inclusion and exclusion criteria. We restricted the review to work published at any time before January 2017 that present a contribution in the area of threat analysis throughout the Software Development Life-cycle. The first five exclusion criteria in Table 2.1 are self-explanatory. We have noticed that a large amount of search results focused on anomaly detection and intrusion detection systems. Furthermore, the search results contained a lot of work published on safety-hazard analysis and relationships between safety and security requirements. For these reasons we added the last two exclusion criteria (6 and 7 in Table 2.1).

2.2.4 Data extraction

Table 2.2: Assessment criteria corresponding to research question RQ1.

Characterization (RQ1)		
Applicability	Level of abstraction	Requirements level Architectural level Design level Implementation level
Input	Type	Goals Requirements Attacker behavior Security assumptions Architectural design Source code
	Representation	Textual description Model-based Other
Procedure	Knowledge based	No Yes
	Level of precision	None Based on examples Based on templates Semi-automated Very precise

	Security objectives	Confidentiality Integrity Availability Accountability Not applicable
	Risk	Not considered Internal part of technique Externally considered
	Stopping condition	Present Not present
Outcomes	Type	Mitigations Threats Security requirements
	Representation	Structured text Model-based Other
	Assurance of quality	Explicit Present Not present
	Granularity	High-level Low level

Table 2.4: Assessment criteria corresponding to research question RQ2.

Ease of adoption (RQ2)	
Tool support	None Prototype tool Tool
Guidance for execution	Coarse grained phrases Fine grained steps No structure
Documentation	Publication Tutorial Presentations Tool documentation Demonstration
Target audience	Engineer Security trained engineer Security expert Researcher

Table 2.5: Assessment criteria corresponding to research question RQ3.

Validation (RQ3)	
Type	Case study Experiment Illustration
Validator	None Author 3rd party both
Domain	Automotive IS SOA SCADA ...

Tables 2.2, 2.4 and 2.5 depict the assessment criteria used to record the information that was needed to answer the research questions. We have extracted the information from 38 publications by building a database of the

identified techniques and corresponding assessments. In this section, we provide the rationale behind some of our choices for criteria levels.

The *types of input* were determined by choosing the most commonly required information for threat analysis to start. This includes requirements (functional or non-functional), attacker behavior, security assumptions, architectural design, source code and goals. The term "goal" is often used as a general term, yet this work makes a distinction between requirements (i.e. goal refinements) and goals. Threat analysis of a system requires at least: (i) the knowledge of what the system is (architecture, source code, functional requirements) and (ii) what it should be protected from (security assumptions, attacker behavior).

Studies have shown (e.g. Yuan et al. [111], Wang et al. [112], Williams et al. [113]) that including *knowledge base* (e.g. taxonomies, catalogs of misuse and abuse cases, attack scenarios and trees, etc.) helps the analyst to identify and analyze threats. Therefore we were interested to record which existing techniques provide a knowledge-base. We have assessed the techniques as knowledge based if the they are supported by some external source of information which helps raise the quality of outcomes. For instance, some techniques provide a catalog of example threats (e.g. STRIDE [9, 105]), templates (e.g. misuse cases) or even use one of the existing databases (such as CAPEC,[4] CWE,[5] CVE[6]) to compute threat suggestions.

In addition to knowledge base, we are interested in observing the *precision* of the analysis techniques. By "precision", we refer to the repeatability or reproducibility of the obtained results. In other words, "precision" is the degree to which repeated measurements under unchanged conditions show the same results [114]. In our case, it would not be an ideal situation every time an expert makes security threat analysis on the same software artefact (e.g., software design, architecture, requirements, source code, etc.) under the same conditions and comes up with a different set of security threats as a result of this analysis. However, as it was shown by cognitive psychologists, humans are incorrigibly inconsistent in making a summary of judgments of complex information [115–117]. As a result, humans frequently give different answers when asked to evaluate the same information twice and this leads to the lack of precision in expert judgments in "low validity environments", which are environments that are not sufficiently regular to be predictable [117, 118]. Figuring out potential security threats for a software system by analyzing its high level design, more fine grained architectural design, its requirements documents or source code also corresponds to making predictions in a low validity environment, as there are many parameters one cannot think of in advance such as the configurations of the final software when it is deployed on site. It was shown that the existence of formulas and algorithms as a backup to expert judgment improves precision [119]. Therefore, we assessed the precision of each technique by observing whether the procedure of analysis is: (i) supported by a formal framework (very precise), (ii) supported by tools that semi-automate the analysis, (iii) based on templates, (iv) based on example threats or (v) not accounting for precision (none).

Since *risk assessment* plays an important role in threat prioritization,

[4] https://capec.mitre.org/

[5] https://cwe.mitre.org/

[6] https://cve.mitre.org/

we have investigated to what extent the techniques consider risk. Namely, some studies focus on associating risk levels to identified threats, while others consider risk externally, e.g. by combining the technique with an external risk management framework.

Notice that in addition to assessing the type and representation of analysis outcomes, this work also investigates the *quality assurance of outcomes*. We assess the techniques on this criterion by observing whether the quality assurance of outcomes is explicit, present or absent (none). Analysis techniques that explicitly assure the outcomes for quality define this activity as part of the analysis procedure. For instance, if the outcomes are represented with models, the technique may perform model verification as part of the analysis procedure (as presented by Dianxiang Xu and K. E. Nygard [49]). However, explicit quality assurance of outcomes is not always present in the studied approaches. If the techniques provide informal guidelines for assessing the quality of outcomes (such as a checklist of most common threats), this study still considers that a form of quality assurance is present. Finally, this work investigates the *granularity of outcomes*. We assess the granularity of outcomes with two levels: high-level and low-level outcome. For example, the analysis technique presented by Almorsy et al. [44] projects the outcomes on models, which can be transformed into source code. Therefore, we have assessed that this technique produces a low-level outcome. On the other hand, Haley et al. [120] present so called "threat descriptions", which are descriptive phrases of the form: performing action "X" on/to asset "Y" could cause harm "Z". Therefore, we have assessed that this technique produces a high-level outcome.

As per RQ2 (Table 2.4), this study also assesses the available support for executing a threat analysis technique. Coarse grained *guidance for execution* include high-level overview of the technique with less detailed descriptions (only using key verbs, for instance). For instance, describing the threat identification as "brainstorming threats with participants" is considered as a coarse-grained guideline. E.g. Whittle et al. [102] provide a recommended process for developing and testing executable misuse cases. Yet, the authors do not further explain how the attack scenarios are identified or how the mitigations are supposed to be re-designed in case the simulation ends in a successful attack. On the other hand, Chen et al. [121] exemplify how to use the supporting tool by describing one instance run. Guidelines are considered to be fine-grained if they include precise instructions for analysis execution.

The *target audience* for the techniques was assessed to understand the minimum knowledge and skills required in order to execute each analysis technique. We identified four levels of competency, three of which are aligned to the Software Assurance Competency Model presented in [122]. Table 2.6 shows the mapping between the competency levels and the target audience considered in our work. An engineer is considered to posses the knowledge and skills of the competency level L1. A security trained engineer is considered to possess an active knowledge of security related concepts and has an engineering degree (BSc and/or MSc), which corresponds to levels L2-L3. Finally, a security expert corresponds to the levels L4-L5. In this work we consider researches to possess an active knowledge of practical as well as theoretical concepts in the field of security in software engineering.

As per RQ3 (Table 2.5) we have developed the assessment criteria to un-

Table 2.6: Target audience considered in this work in relation to the competency levels in [122].

Target audience	Level	Major tasks	Exemplary title
Engineer	L1	Tool support, low-level implementation, testing, and maintenance	Junior Software Developer, Acceptance tester, Junior Security Engineer, Software Assurance Technician
Security trained engineer	L2-L3	Requirements fundamentals and analysis, architectural design, implementation, risk analysis and assessment	Security Analyst, Release Engineer, Information Assurance Analyst, Maintenance Engineer, Senior Software Developer, Software Architect
Security expert	L4-L5	Assurance assessment, assurance management, risk management across the SDL, advancing in the field by developing, modifying, and creating methods, practices, and principles at the organizational level or higher	Project Manager, Senior Software Architect, Chief Information Assurance Engineer, Chief Software Engineer
Researcher	-	Remain in touch with the current research and publish own research in the discipline of security in software engineering	PhD student, Post Doctoral candidate, Assistant Professor, Senior lecturer, etc.

derstand how each technique was validated. We have assessed the validation of each technique with three levels: case study, experiment and illustration. A *case study* is sometimes a rather loosely used term in software engineering. According to Runeson and Höst [123] the presented case studies in software engineering range from very ambitious and well organized studies in the field, to small toy examples that claim to be case studies. A case study is a research methodology used to study a real phenomena of *exploratory, descriptive, explanatory* and *improving purpose*, the later being most popular in software engineering [123]. It requires rigorous planning, data collection and triangulation, data analysis, a discussion on threats to validity and evidence based conclusions. This work considers all applications of the proposed approach to a real world problem as case studies, in spite of only a handful (if any) conforming to the previous definition. In addition to case studies our assessment criteria includes two other forms of validation, namely illustrations and experiments. In contrast to the lightweight illustrations, experiments measure the effects of manipulating dependent variable(s) on an independent (response) variable. Experiments identified within this study were mostly experiments in empirical software engineering (e.g. comparative experiment of two techniques).

2.2.5 Quality assurance in this study

The selection of primary studies and data extraction was performed by a single researcher (first author). In order to circumvent the effects of potential bias, the following quality assurance plan was implemented.

Random assessment of included/excluded publications. We have randomly selected 10% of search results and the second author has applied the inclusion and exclusion criteria independently (filtering 1 in Figure 2.1). The outcome has been compared to the results of the first author. The few disagreements (2 papers) of plausible exclusion were discussed between the the two researchers until an agreement was reached. A summary of this discussion was submitted

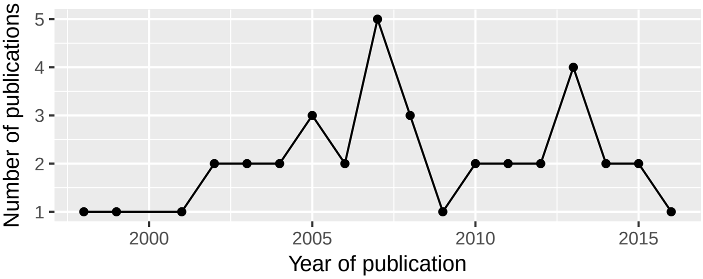


Figure 2.2: Year of publication for the selected techniques.

to the third author of this study for further assurance. In summary, we are confident that the inclusion and exclusion criteria was crisp enough to minimize any selection bias.

Random quality check of data extraction. A second random quality check was performed by the second author with regards to the data extraction. A random sample of the included publications (5 publications, roughly 10%) was independently re-assessed. The outcomes of this quality check were again compared to the outcomes obtained by the first author. This comparison yielded to a few discrepancies due to the perceived definition of certain criterion levels (6 out of 67 criterion levels). The first and second author revisited the precise differences between: (i) tool and prototype tool, (ii) engineer with security background and security expert, (iii) goal and requirement, (iv) design and architectural level of abstraction, (v) external and internally considered risk, and (vi) case study and illustration. After a consensus was reached, the first author of this study manually examined the rest of the publications (90%) to assure that the assessments were correct. To conclude, the authors are confident that the data was extracted correctly.

Continuous soundboarding. Informally, several sessions were held with all authors to maintain the quality of the review. For instance, the list of publications obtained from the initial pilot review were discussed. Further, the inclusion, exclusion and assessment criteria were refined during such sessions. These sessions were held continuously as sanity checks for the first author.

2.3 Results

In this section, we first overview the techniques and then present the answers to the research questions. We conclude this section with recommendations for practitioners, offering insight to the reader for when to use certain techniques.

2.3.1 Overview of threat analysis techniques

Figure 2.2 shows a time-line of the 38 publication included in this SLR. Overall, the interest in the area of threat analysis approaches seems to be rather constant with an average of 2 publications per year. Despite a the slight progression of publications observed from early 2000 until 2007, it is difficult draw tendencies due to the small numbers.

Table 2.7: Threat analysis techniques. Note that, some publications were grouped by leading authors, sometimes resulting in observing separate techniques rather that fully fledged methodologies.

Methodology	**Ref**	**Technique**
Abe et al.	[124]	Threat patterns, negative scenarios
Almorsy et al.	[44]	Attack scenarios
Attack and Defense Trees	[34, 125]	Attack trees, defense trees
Beckers et al.	[126]	MUC
Berger et al.	[45]	DFDs, rule-based graph matching
CORAS	[127]	Threat, risk, treatment diagrams and descriptions
Chen et al.	[121]	Attack paths
Dianxiang Xu and K. E. Nygard	[49]	Petri-nets
El Ariss and Xu	[128]	State charts
Encina et al.	[129]	Misuse patterns
Extended i*	[130–133]	Attacker agents with goals
Haley et al.	[120, 134]	Threat tuple-descriptions with rebuttals to claims
Halkidis et al.	[135]	STRIDE, Fault tree analysis
Hatebur, Heisel et al.	[36–38]	Problem frames
J. McDermott et al.	[136, 137]	Abuse cases
KAOS	[138–140]	Threat graphs rooted in anti-goals, anti-models, threat trees
Karpati et al.	[141, 142]	MUC maps, MUC, attack trees
LINDDUN	[11]	Threat to (DFD) element mapping, threat tree patterns, MUC scenarios
Liu et al.	[143]	Attacker agents with goals
P.A.S.T.A.	[29]	Threat scenarios with associated risk and countermeasures
STRIDE	[9, 105]	Threat to (DFD) element mapping
Sheyner et al.	[47]	Attack graphs
Sindre and Opdahl	[35]	MUC
Tong Li et al.	[144]	Automated generation of attack trees
Tøndel et al.	[145]	MUC, attack trees
Whittle et al.	[102]	MUC

Table 2.7 depicts threat analysis techniques included in this SLR. Most commonly used techniques in the presented body of knowledge were misuse cases, attack trees, problem frames and several software-centric approaches.

Misuse cases (MUC) are derived from use cases in requirements engineering. In the form of templates, they are used to capture textual descriptions of threat paths, alternative paths, mitigations, triggers, preconditions, assumptions, attacker profiles, etc. The literature also mentions abuse cases, MUC maps and MUC scenarios. The difference between misuse and abuse cases is subtle and the two terms are sometimes used interchangeably. Strictly speaking, abuse is misuse with malicious intent. MUC maps and scenarios both focus on representing chained attacks, from start to the end of vulnerability exploitation.

Another way of identifying alternative paths of attack is by using *attack (or threat) trees*, where the root node is refined into leaves representing all possible attacker actions. Therefore an attack path is a single path starting at leaf node leading to the root node. Attack trees are commonly adopted in a combination with other techniques. For instance, LINDDUN [11] proposes a combined analysis by first mapping the threats to (DFD) elements, using threat tree patterns and usage scenarios in order to identify MUC scenarios.

Much like threat patterns, *problem frames* are used to describe problems in software engineering. They define an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirements (M. Jackson [146]). As such problem frames are rather general in scope, therefore conceptualized security problem frames were soon introduced (Hatebur, Heisel et al. [37] [36] [38]).

Goal-oriented requirements engineering (GORE) perceives systems as a set of agents communicating in order to achieve goals. In GORE goals (or anti-goals) are refined until finally requirements (or anti-requirements) are achieved.

Finally, several software-centric techniques are well recognized in the software engineering community, particularly in the industrial space, such as STRIDE [9, 105], CORAS [127], P.A.S.T.A [29], DREAD [147], Trike [148], to name a few.

Table 2.8 shows the analysis techniques, their respective domains of validation and tool support. It is generally acceptable to group threat analysis techniques into risk-centric, attack-centric and software-centric techniques.

Risk-centric threat analysis techniques focus on assets and their value to the organization. They aim at assessing the risk and finding the appropriate mitigations in order to minimize the residual risk. Their main objective is to estimate the financial loss for the organization in case of threat occurrence (e.g. CORAS [127]). Therefore, when risk-centric techniques are used assets dictate the priority of elicited security requirements.

On the other hand, *attack-centric* threat analysis techniques focus the analysis around the hostility of the environment. They put emphasis on identifying attacker profiles and attack complexity for exploiting any system vulnerability (e.g. Attack trees [34]). Their main objective is to achieve high threat coverage and identify appropriate threat migitations.

Finally, the literature also mentions so-called *software-centric* threat analysis techniques. This group includes techniques that focus the analysis around the software under analysis. For example, in STRIDE [9] [105] the analysis is per-

Table 2.8: The selected analysis techniques.

Methodology	**Ref**	**Approach**	**Domain**	**Tool**	**Validation**
Abe et al.	[124]	Attack-centric	IS	none	CS
Almorsy et al.	[44]	Attack-centric	ERP, Web, E-commerce	none	EXP
Attack and Defense Trees	[34, 125]	Attack-centric	IS, other, ATM	tool	CS,EXP,ILU
Beckers et al.	[126]	Privacy	Cloud computing, E-bank	none	ILU
Berger et al.	[45]	Attack-centric	Logistic application	tool	CS
CORAS	[127]	Risk-centric	Telecom, SCADE, IS	tool	CS,EXP
Chen et al.	[121]	Attack-centric	IT, COST	prototype	CS
Dianxiang Xu and K. E. Nygard	[49]	Attack-centric	Web store	none	CS
Encina et al.	[129]	Attack-centric	Cloud services	none	CS
Extended i*	[130–133]	GORE	Web-IS	tool	ILU
Haley et al.	[120, 134]	SRE	Air traffic, HR, IS	none	CS,ILU
Halkidis et al.	[135]	Risk/Attack-centric	E-commerce	tool	EXP
Hatebur, Heisel et al.	[36–38]	Risk/Attack-centric	E-commerce	tool	ILU ,CS
J. McDermott et al.	[136, 137]	SRE	IS	none	CS
KAOS	[138–140]	GORE	E-commerce, Web store, Ambulance system	tool	CS
Karpati et al.	[141, 142]	Attack-centric	Banking system, IS, Web-based IS	tool	CS,EXP
LINDDUN	[11]	Privacy, GORE	Social network, E-health application	none	CS,EXP
Liu et al.	[143]	SRE, GORE	IS	tool	CS
P.A.S.T.A.	[29]	Risk-centric	Web-bankig application	none	ILU
STRIDE	[9, 105]	Software-centric	IS, automotive, other	tool	CS,ILU,EXP
Sheyner et al.	[47]	Attack-centric	System Network	tool	ILU
Sindre and Opdahl	[35]	SRE	E-store, Telemedicine	tool	CS,EXP
Tong Li et al.	[144]	Attack-centric	Smart grid	prototype	CS
Tøndel et al.	[145]	Attack-centric	IS	none	ILU
El Ariss and Xu	[128]	Attack-centric	Web store	none	CS
Whittle et al.	[102]	SRE	E-voting, CPS	tool	CS

Table 2.9: The characteristics of the applicability and input of the selected techniques.

		Applicability Abstraction				**Input Type**						**Repre-senta-tion**		
Methodology	**Ref**	Requirements	Architectural	Design	Implementation	Requirements	Attacker behavior	Security assumptions	Architectural design	Source code	Goals	Textual description	Model-based	Other
Abe et al.	[124]		●			●							●	
Almorsy et al.	[44]		●	●	●	●		●	●	●			●	●
Attack and Defense Trees	[34, 125]	●	●								●			●
Beckers et al.	[126]	●							●				●	
Berger et al.	[45]		●	●					●				●	
CORAS	[127]		●						●			●		
Chen et al.	[121]			●	●		●	●				●		
Dianxiang Xu and K. E. Nygard	[49]			●		●	●						●	
El Ariss and Xu	[128]		●								●	●		
Encina et al.	[129]	●					●					●		
Extended i*	[130–133]	●									●	●		
Haley et al.	[120, 134]	●	●			●		●	●			●	●	
Halkidis et al.	[135]		●	●					●			●	●	
Hatebur, Heisel et al.	[36–38]	●	●	●		●	●	●	●			●	●	
J. McDermott et al.	[136, 137]	●				●						●		
KAOS	[138–140]	●									●	●		
Karpati et al.	[141, 142]		●	●			●		●			●		●
LINDDUN	[11]	●	●					●	●			●	●	
Liu et al.	[143]	●									●	●		
P.A.S.T.A.	[29]		●	●		●		●	●			●	●	●
STRIDE	[9, 105]		●	●				●	●				●	
Sheyner et al.	[47]			●					●				●	
Sindre and Opdahl	[35]	●				●						●	●	
Tong Li et al.	[144]	●				●	●					●		●
Tøndel et al.	[145]	●	●	●		●							●	
Whittle et al.	[102]	●				●							●	

formed on DFDs, which provide a high-level architectural view of the software.

However, not all threat analysis techniques can be categorized in the aforementioned three groups. For instance, in GORE the main goal could be "stealing the GPS coordinates of a vehicle fleet". In this case, the analysis would clearly evolve around that particular asset and could be therefore considered as risk-centric. Yet, the main goal could also be "malicious access to a DNS server". In this case, the discussions and the analysis can be considered as attack-centric. For this reason, we categorize the techniques also as "GORE", "SRE" and "Privacy", as shown in Table 2.8.

Overall, a majority of techniques are attack-centric (≈ 45%) and requirements engineering approaches (GORE ≈ 20%, Security Requirements Engineering (SRE) ≈ 15%). We continue to present the results for the research questions in the subsequent sections.

2.3.2 RQ1: Characteristics

Table 2.9 shows the characteristics of the applicability and input of the selected techniques.

Applicability (RQ1.1). In general, threat analysis can be performed iteratively at several stages of the software development. In this study, we differentiate between abstraction levels according to the input information required for analysis execution. We have assessed each technique for applicability at the level of requirements, architecture, design and implementation. For instance, in order to create and manually analyze attack trees the analyst only needs high level goals (or anti-goals). Therefore, the most basic form of attack trees are applicable at the level of requirements and architecture. In this study we make a distinction between the design and architectural level of abstraction.

On the *architectural level of abstraction* requirements are used in order to construct the architecture. Software architecture is a set of principal design decisions made about the system (as defined by N. Taylor et al. [149]). The outcomes of this level of abstraction are high-levels diagrams (such as DFDs), sequence diagrams, flow-charts etc. The word "principal" here indicates that not all design decisions are architectural. In fact many design decisions are related to the domain, algorithms, programming languages or are based on preference.

Therefore, *designing* the intended architecture includes committing to a set of architectural styles and patterns, which are further refined until a detailed design is evaluated against the system requirements. The outcomes of this level of abstraction include most (or all) the design decisions made about the system (e.g. component diagrams, connector types and interfaces, deployment diagrams, etc).

Only two techniques are applicable at the level of implementation, where a concrete system is taken into account. First, Almorsy et al. [44] describe a semi-automated Model-Driven Engineering (MDE) approach for a partial architecture reconstruction, followed by a risk-centric threat analysis. Second, Chen et al. [121] presents a quantitative threat analysis approach based on attack-path analysis of COTS systems. Predominantly, the techniques are applicable at the level of requirements (14 techniques), architecture (14 techniques) and design (11 techniques).

Input (RQ1.2). The input of a threat analysis technique is all the information required in order to begin with threat identification. In order to understand the input information for each threat analysis technique, we have observed input type and representation. The *input type* can vary from high-level goals, requirements, attacker behavior, security assumptions, architectural design to source code of the system under analysis. For instance, the root node in an attack tree, typically referred to as an anti-goal, is decomposed into hierarchical leaves of possible attacker actions. Despite the domain knowledge and security expertise needed to find anti-goals and possible attack actions, the analyst does not require more than a high-level description of the system (e.g. in terms of its business functionality). The *input representation* was assessed with three levels: textual description, model-based and other.

One third of the analyzed techniques require as input architectural design (12 techniques) and one third requirements (11 techniques). Some techniques (6) consider the attacker behavior as input. Security assumptions are required

Table 2.10: The characteristics of threat analysis procedure. KB = Knowledge Base.

		KB		**Precision**					**Objectives**					**Risk**		
Methodology	**Ref**	Yes	No	None	Based on examples	Based on templates	Semi-automated	Very precise	Confidentiality	Integrity	Availability	Accountability	Not applicable	Not considered	Internal part	Externally considered
Abe et al.	[124]	●				●			●	●	●	●		●		
Almorsy et al.	[44]	●			●		●		●	●	●	●			●	
Attack and Defense Trees	[34, 125]	●						●					●	●		
Beckers et al.	[126]	●				●			●	●	●	●				●
Berger et al.	[45]	●			●		●		●	●	●	●		●		
CORAS	[127]		●	●					●	●	●	●			●	
Chen et al.	[121]	●				●	●		●	●	●				●	
Dianxiang Xu and K. E. Nygard	[49]		●					●	●	●	●	●		●		
El Ariss and Xu	[128]		●	●									●	●		
Encina et al.	[129]		●	●					●	●	●	●		●		
Extended i*	[130–133]		●			●			●	●	●	●				●
Haley et al.	[120, 134]		●					●	●	●	●	●		●		
Halkidis et al.	[135]	●				●							●		●	
Hatebur, Heisel et al.	[36–38]		●			●			●	●	●	●		●		
J. McDermott et al.	[136, 137]		●	●									●	●		
KAOS	[138–140]	●					●	●	●	●	●	●				●
Karpati et al.	[141, 142]		●	●									●	●		
LINDDUN	[11]	●				●			●			●				●
Liu et al.	[143]		●	●									●	●		
P.A.S.T.A.	[29]		●	●					●	●	●	●			●	
STRIDE	[9, 105]	●			●				●	●	●	●				●
Sheyner et al.	[47]		●				●						●		●	
Sindre and Opdahl	[35]		●			●							●			●
Tong Li et al.	[144]	●			●	●			●	●	●	●				●
Tøndel et al.	[145]	●					●						●		●	
Whittle et al.	[102]		●				●						●	●		

for analysis in less than 25% of techniques (7). Only one technique takes source code into account as input to the analysis. Almorsy et al. [44] present a technique where source code is an optional input to the analysis. Finally, in some techniques (5) high-level goals were used as input to the analysis. The required input is commonly represented either with textual descriptions (16 techniques), models (15 techniques) or both (6 techniques).

Procedure (RQ1.3). Threat analysis procedure includes all required actions and tasks that the analyst needs to perform in order to obtain the desired outcomes. As depicted in Table 2.10, we assess the characteristics of the procedure that takes place during each analysis technique. To this aim we observe traces of knowledge base, precision, security objectives and risk in the procedure of each analysis technique.

On average about half of the techniques include a knowledge base of some kind. As previously mentioned, knowledge base (domain or security knowledge) helps the analyst to identify threats in the context of the system in question. Yet, we have found that most approaches take advantage of the existing knowledge base, rather than contribute with innovative examples (e.g. Hatebur et al. [37]). For instance, Almorsy et al. [44], Berger et al. [45], Chen et al. [121] and Tondel et al. [145] present formalized rules to extract knowledge from public repositories of threats and vulnerabilities namely Common Weakness Enummeration (CWE) [150] , Common Attack Pattern Enummeration and Classification (CAPEC) [89], Open Web Application Security Project (OWASP) [151, 152].

In general, the precision of the technique procedures is on the level of templates and examples (about half of the publications). Four techniques (namely, Attack and defense trees [34] [125], Dianxiang Xu and K. E. Nygard [49], Haley et al. [120] [134] and KAOS [138–140]) formally approach the analysis and are therefore very precise. Finally, six techniques (Almorsy et al. [44], Berger et al [45], Chen et al. [121], KAOS [138–140], Sheyner et al. [47] [153], Tøndel et al. [145], Whittle et al. [102]) introduce a semi-automated approach using tools (or prototype tools). According to our assessment, about a quarter of techniques (7) describe the analysis procedure with no regards towards the precision of the analysis.

A majority of techniques address security objectives explicitly in the presented approach. Some studies specifically mention only one security objective, yet in our assessment we include other security objectives that could be directly applied in the proposed approach. For instance, Hatebur, Heisel et al. [36–38] describe problem frames by introducing the authentication frame, therefore they consider confidentiality and integrity. However, the authors do not initialize possible problem frames for all security objectives. Ultimately, we do not see significant obstacles to introduce problem frames for other security objectives.

About half of the techniques (12) do not include risk assessment as part of the threat analysis technique. The rest of the studies are either risk-centric (7 techniques) or consider risk as an external activity (7 techniques).

Outcomes (RQ1.4). As previously mentioned, we have observed the type and representation of outcomes. Table 2.11 shows the outcome characteristics of the selected techniques. We have monitored three *types of outcomes*: threats, mitigations and security requirements. All techniques present approaches that produce threats as main outcomes. Threat mitigations are security countermeasures planned for lowering the residual risk. Design-level security countermeasures are further on refined into implementable security requirements. Beyond threats as main outcomes, about half of the techniques also produce threat mitigations (15) and security requirements (12) as outcomes. In fact about a third of the techniques (8) produce all three types of outcomes (namely Dianxiang Xu and K.E. Nygard [49], Extended i* [130–133], Haley et al. [120, 134], J. McDermott et al. [136, 137], KAOS [138–140], LINDDUN [11], Sindre and Opdahl [35], Whittle et al. [102]).

In addition, we have observed the *representation of outcomes*. Most techniques result in outcomes represented with either a structured text (16), model-based form (16), or both (6). For instance, Whittle et al. [102] introduce an aspect-oriented approach that results in finite state machines (model-based), misuse cases (model-based) and elicited security requirements (structured text).

Next to the type and representation, we have observed the *quality assurance of outcomes* for each analysis technique. Only a handful of techniques (3) have an explicit way of assuring the quality of outcomes (namely, Dianxiang Xu and K.E. Nygard [49], Haley et al. [120,134], KAOS [138–140]). For example, Haley et al. [120, 134] include an activity for constructing satisfaction arguments as part of the procedure. The satisfaction arguments are used in order to verify whether the primary and secondary goals are satisfied with the resulting security

Table 2.11: The outcomes characteristics of the selected techniques.

		Type			**Representation**		**Quality assurance**			**Granularity**	
Methodology	**Ref**	Mitigations	Threats	Security requirements	Structured text	Model-based	Explicit	Present	Not present	High-level	Low-level
Abe et al.	[124]		•			•			•	•	
Almorsy et al.	[44]		•		•				•	•	•
Attack and Defense Trees	[34, 125]	•	•			•		•		•	
Beckers et al.	[126]		•	•	•			•		•	
Berger et al.	[45]	•	•		•				•	•	
CORAS	[127]	•	•			•			•	•	
Chen et al.	[121]	•	•		•				•		•
Dianxiang Xu and K. E. Nygard	[49]	•	•	•		•	•			•	
El Ariss and Xu	[128]		•			•			•	•	
Encina et al.	[129]	•	•		•				•	•	
Extended i*	[130–133]	•	•	•		•			•	•	
Haley et al.	[120, 134]	•	•	•	•		•			•	
Halkidis et al.	[135]		•		•				•	•	
Hatebur, Heisel et al.	[36–38]	•	•		•	•			•	•	
J. McDermott et al.	[136, 137]	•	•	•	•	•		•		•	
KAOS	[138–140]	•	•	•	•	•	•			•	
Karpati et al.	[141, 142]		•	•		•			•	•	
LINDDUN	[11]	•	•	•	•				•	•	
Liu et al.	[143]	•	•		•	•		•		•	
P.A.S.T.A.	[29]		•			•			•	•	
STRIDE	[9, 105]		•		•				•	•	
Sheyner et al.	[47]		•			•			•	•	
Sindre and Opdahl	[35]	•	•	•	•	•			•	•	
Tong Li et al.	[144]		•	•	•				•	•	
Tøndel et al.	[145]		•	•		•		•		•	•
Whittle et al.	[102]	•	•	•	•	•		•		•	

requirements. The rest of the techniques do not have an explicit activity for quality assurance of outcomes. E.g., Beckers et al. [126] present a method for information security management system for cloud IS that includes threat analysis based on patterns. Their structured approach is aligned with ISO 271001 security standard and includes guidelines for assuring the quality of outcomes. Therefore, some form of quality assurance of outcomes is present, yet not explicitly defined. We have observed a presence of some kind of quality assurance of outcomes in 6 techniques. Still, the quality assurance of outcomes is predominantly not present in the techniques. For instance, Abe et al. [124] propose an interesting approach for threat pattern detection and negative scenario generation, using transformation rules on sequence diagrams. However, the authors do not evaluate or measure the quality of the generated negative scenarios.

Regarding the *granularity of outcomes*, only three of the techniques (namely Almorsy et al. [44], Chen et al. [121], and Tøndel et al. [145]) produce low-level outcomes (e.g. source code). Almost all of the techniques (25) result in outcomes of high-level abstraction, which is in line with the results obtained from observing the applicability of techniques (RQ1.1).

2.3.3 RQ2: Ease of adoption

As shown in Table 2.12, about a third of the techniques (9) do not include any tool support. The rest are supported by tools (13 techniques) or present a prototype tool (4 techniques). In general, the majority of studies include coarse-grained guidelines for execution, which could be inferred from the publication. Six techniques provide fine-grained guidelines, yet three of them are not supported by a tool. Furthermore, most approaches together with their tools are only documented in the respective publications. Only a handful of techniques provide a tool with precise guidelines on how to use it.

The target audience of the techniques are most commonly security experts (9) and security trained engineers (10). Most of the techniques describe approaches that do not require extensive knowledge of any research field. According to our assessment, only two techniques are targeted more towards researchers, namely Dianxiang Xu and K. E. Nygard [49] and Halkidis et al. [135]. In general, knowledge base, automation and tool support can decrease the level of required expertise. Despite that, important steps in the analysis are still required by experts (namely, threat identification and prioritization). Two techniques are thoroughly documented and two of them could be used by engineers without further training (STRIDE [9, 105] and KAOS [138–140]). They are both knowledge-based (they provide example threats) and are well documented (they provide tutorials, presentations, etc.) Evidently, the better the approach is documented, the easier it is to apply in practice.

2.3.4 RQ3: Validation

As shown in Table 2.8, the majority of techniques (19) were validated with a case study. Despite of the recently increasing quantity of empirical studies in software engineering and the long history of advocating empirical research in software engineering, there is still room for improvement [154]. About 20% of techniques (8) were validated also with an experiment, reflecting the immatu-

Table 2.12: The ease of adoption for techniques.

		Tool support			**Execution**			**Documentation**					**Target audience**			
Methodology	**Ref**	None	Tool	Prototype	No structure	Coarse-grained	Fine-grained	Publication	Tutorial	Presentations	Tool docum.	Demonstration	Engineer	Sec trained Engineer	Security expert	Researcher
Abe et al.	[124]	●				●		●						●		
Almorsy et al.	[44]			●		●		●							●	
Attack and Defense Trees	[34, 125]		●			●		●	●	●	●	●			●	
Beckers et al.	[126]		●			●		●						●		
Berger et al.	[45]		●				●	●						●		
CORAS	[127]		●			●		●							●	
Chen et al.	[121]			●			●	●		●			●			
Dianxiang Xu and K. E. Nygard	[49]	●					●	●								●
El Ariss and Xu	[128]	●				●		●					●			
Encina et al.	[129]	●			●			●					●			
Extended i*	[130–133]		●			●		●					●			
Haley et al.	[120, 134]	●					●	●							●	
Halkidis et al.	[135]	●				●		●							●	●
Hatebur, Heisel et al.	[36–38]		●			●		●						●		
J. McDermott et al.	[136, 137]	●				●		●					●			
KAOS	[138–140]		●				●	●	●	●	●				●	
Karpati et al.	[141, 142]		●			●		●			●			●		
LINDDUN	[11]	●					●	●	●					●		
Liu et al.	[143]		●			●		●						●		
P.A.S.T.A.	[29]	●				●		●		●				●		
STRIDE	[9, 105]		●			●		●	●	●	●	●	●	●		
Sheyner et al.	[47]		●			●		●							●	
Sindre and Opdahl	[35]		●			●		●		●			●			
Tong Li et al.	[144]			●		●		●							●	
Tøndel et al.	[145]			●		●		●						●		
Whittle et al.	[102]		●			●		●							●	

rity of empirical research in the software engineering community. In addition, extensive validation (case studies and experiments) was often applied only in the domain of Web systems (E-commerce, Web store, E-bank, Social network, E-health, Web bank, E-store, E-voting). About one third of the techniques (9) were validated by illustrations.

2.3.5 Recommendations for practitioners

Previously reported results have not considered any preferences between the levels of our assessment criteria (e.g. model-based is preferred over textual input). To complement the objectively reported results, we include a more subjective reflection, where we aim at providing insights to the reader about the benefits of adopting certain threat analysis techniques. To this aim we have considered the amount of resource investment needed for adopting a technique. In order to simplify the discussion we categorize the resource investments into "small" and "large".

If the planned resource investment is *"small"*, the organization is likely to prefer using a technique as is, without any improvements. Additionally, if *security is not prioritized*, the allocated budget might be sufficient only for recruiting a security trained engineer (e.g. requirements engineer). In this case, the time spent on threat analysis is limited. In addition, the target audience of the technique should be engineers with or without security training. Therefore, tool availability (and maturity), documentation, low target audience and a lightweight procedure (i.e. level of precision is none, based on examples or templates) are the most valued criteria for technique selection. According to the results of this study, techniques originating in requirements engineering fit this description. In our opinion, the tier techniques that could be adopted in this kind of organizations are CORAS [127], Problem Frames (Hatebur, Heisel et al.) [36–38], MUC (Sindre and Opdahl) [35] and Abuse cases (J McDermott et al.) [136, 137], Extended i* [130–133], KAOS [138–140]. These techniques seem to require less effort to use as they are less systematic and thorough. They are more intuitive and are supported by toolkits such as RE-Tools.[7]

The aforementioned techniques lack guarantees for analysis correctness (i.e. quality assurance of outcomes). In organizations where *security is prioritized*, quality assurance of outcomes also becomes important. In this case, more effort for threat analysis is justified. Therefore, the existing budget for security is bigger, sufficient for recruiting security experts. The preferred techniques should not only have good tool support and documentation, but also be systematic, thorough, expert-based and possibly semi-automated. In our opinion STRIDE [9, 105] and LINDDUN [11] are the tier techniques that fit this description. STRIDE (similarly LINDDUN) is a systematic approach that visits each element in the DFD and is therefore subjected to the so called "threat explosion" problem. In order to counter the explosion problem, some automation (namely threat category generation) is already available by the MTM.[8]

The previously mentioned techniques require little additional effort since quality assurance of outcomes is not prioritized. However, if the planned resource investment is *"large"*, the organization is likely prepared for improving

[7] `http://www.utdallas.edu/~supakkul/tools/RE-Tools/`

[8] `https://www.microsoft.com/en-us/download/details.aspx?id=49168`

an existing technique to obtain an “in-house” adapted version. We also consider academic researchers looking for a starting point in their research to be prepared for a “large” investment of resources. These techniques should be systematic by construction (e.g. formal) but most importantly show potential for improvement (e.g. technology improvement). In our opinion, two such techniques stand out. First, the work of Berger et al. [45] presents an interesting semi-automated technique for extracting threats from graphs based on rules matching certain CAPEC and CWE entries. The authors argue that the existing notation for DFDs needs to be extended with more security semantics. To this aim, Berger et al. extend the notation by annotating flows with assets, security objectives and type of communication (e.g. manual input). A more formal definition of security semantics might assure the quality of outcomes explicitly, which is a promising research direction. Further, querying graphs could be implemented using a different set of technologies. Therefore we believe that their approach is with some effort adaptable to the needs of the organization. Second, Almorsy et al. [44] have used Object Constraint Language (OCL) to define attack scenarios and security metrics. The authors developed an Eclipse plug-in that is able to perform a trade-off analysis for different applications based on their signature evaluator. Minimizing the architectural design space with such a semi-automated trade-off analysis could indeed benefit organizations. For organizations that already practice MDE, this approach could be tailored to the models they use. Yet OCL constraints can only be as accurate as the model instances, therefore it might be promising to pursue this research outside the space of MDE.

2.4 Discussion

In this section we first discuss the applicability of threat analysis techniques in current trends in the software engineering community. We then proceed to discuss the main findings of this study. In summary, the main findings are the following:

(i) There is potential for improving threat analysis techniques in order to be applicable in the context of current trends in software engineering,

(ii) there is a lack of Definition of Done in the threat analysis procedures,

(iii) there is a lack of quality assurance of outcomes,

(iv) the use of validation by illustration is predominant which is worrisome,

(v) the tools presented in the primary studies lack maturity and are not always available.

2.4.1 Potential for improvement along current trends

Development and Operations (DevOps) is a software engineering practice that aims at unifying software development and operations by means of higher automation, measurement, sharing and promoting a specific culture in the organization. Commonly adopted activities, such as continuous integration and deployment cause the SDL to shorten considerably. Such organizations face significant challenges in providing the required security of the product under the rapid rate of software changes. Despite the immaturity of research in integrating security into DevOps, some efforts are summarized by Mohan et al. [155]. According to a recent survey performed with practitioners [156], the majority of participants believe that other security practices are prevalent in DevOps organizations (i.e. security policies, manual security tests, security configuration). To the best of our knowledge, threat analysis techniques have not been applied in the context of DevOps. In our opinion, there are three areas where existing techniques could be improved in order to cater to the needs of DevOps.

First, it is important that the information that was gained from threat analysis is automatically propagated to source code level (and vice-versa). It might be beneficial to assure the *traceability* between the threats and corresponding security requirements at the level of implementation. This might facilitate a more efficient reuse of analysis outcomes in the fast changing code base. Establishing a traceable link between architectural design and implementation can be achieved with a “top-down” or “bottom-up” approach. In a “top-down” approach, the architectural design decisions need to be annotated in the source code (e.g. as presented by Abi-Antoun and Barnes [157]). Such annotations may have to be added manually by developers themselves, which could render the technique unreliable. Therefore, there are existing approaches to extract the architecture from the code base (i.e. Software Architecture Reconstruction (SAR)) by employing dynamic and/or static reverse engineering techniques (e.g. as presented by Granchelli et al. [158]). To the best of our knowledge, the existing tools supporting SAR have limitations and are not commonly applied to practice. From a usability perspective, practices such as continuous deployment cause uncertainty in the security implications of modified code base. For instance, it would be beneficial for developers to get instant feedback on how their contribution impacts the security of the code base (e.g. one threat is mitigated).

Second, the existing techniques would benefit from guidelines of how to *compose the analysis* outcomes. In practice, the software systems under analysis are too large and complex to be analyzed at once. Therefore, organizations are forced to scope the system into sub-systems and assign the analysis to several teams of experts to be analyzed simultaneously. As a results, border elements are either analyzed multiple times, or overlooked. One possible solution could be to scope the system according to assets. In this case, elements handling certain assets would be analyzed together in an end-to-end manner. To facilitate the composability of analysis outcomes, a level of formalism could be beneficial. For example, taint analysis has been used to analyze applications in order to present potentially malicious data flows to the human analyst. The analyst (or automated malware detection tool) is able to decide whether particular flows constitute a policy violation. For instance, Arzt et al. [96] present a flow-sensitive taint analysis tool for Android applications. One possible research direction could be in using hybrid taint analysis techniques on architectural models.

Third, the analysis performed for one subsystem is related to security assumptions, which may not be in line with the security assumptions of another subsystem. Further, threats with high impacts to the organization are typically prioritized. Threat prioritization is commonly still performed manually, which demands a lot of resources. Therefore, existing analysis techniques need to invest in *impact analysis* automation.

The literature states that some *Agile* practices such as Extreme Programming (XP) are not suitable for high-reliability requirements [159]. Similarly to DevOps, agile development practices require highly automated threat analysis techniques due to short sprints. Incidentally, start ups and Agile organizations adopting novel software engineering practices with less supervision are facing similar challenges.

To conclude, in light of DevOps and Agile, where software development is driven by change, there are three important aspect where existing analysis techniques have yet to mature: (i) traceability of analysis in the code base, (ii) composability of analysis outcomes and (iii) threat impact analysis automation.

2.4.2 Definition of Done (DoD)

Threat analysis is typically performed on a certain level of abstraction. The level of abstraction is determined during the first session by system architects and security experts. However, the analysts will typically also consider threats on a lower level of abstraction, depending on their feasibility. It is up the experts to determine which parts of the system should be analyzed in detail (on a low abstraction level). Analysts are also faced with the challenge of deciding how many identified threats (and at what level of abstraction) are enough for a "good" analysis of a particular sub-system. In the Agile community, a so called Definition of Done (DoD) is used for guiding Scrum teams to program more efficiently and minimize technical debt. For example, checklists of test cases can be used to define when a planed release is finished. We borrow this term from the Agile community in order to depict the lack of similar practices in existing threat analysis techniques. Defining when threat analysis can be concluded is still an open question, which is today handled by practitioners in an ad-hoc manner. Future work in this direction could have a large impact on the IoT domain, where systems are composed of a large number of middle-ware components and devices.

2.4.3 Lack of precise guidelines

We have found that there is a lack of precisely defined rules for the analysis. Some techniques operationalize rules for discovering threats and apply them on a graph representing the architectural model (e.g. Almorsy et al. [44]). Yet, the guidelines provided by authors for using their plug-in tool are vague and informal. Moreover, El Ariss and Xu [128] refer to the process of constructing attack trees as goal refinements that continue until the desired level of abstraction is reached. Such guidelines are not precise and, for instance, do not elaborate on how to identify AND/OR gates of attack trees. The lack of precise guidelines effects the techniques' ability to assure the quality of outcomes. We have rarely found that the techniques have an explicit way of determining how well the analysis was performed. While some approaches check for the number of threats found in comparison to a base-line analysis, only a handful do so systematically and automatically.

2.4.4 Generalization across domains

As shown in Table 2.8 the domains of validation vary, yet the majority are still applied to Web-based systems. However, traditional threat analysis techniques appear to be used in some form independently of the domain. In particular, we have discussed the commonly used varieties and combinations of (i) STRIDE [9, 105], (ii) attack trees [34], graphs and paths [121], (iii) MUCs [35]

(iv) problem frames [37] and threat patterns [124]. We argue that the aforementioned techniques are more general in nature and are therefore easily applicable across domains. Unfortunately, we have found that most approaches are poorly validated (using illustrations on toy examples) and the limited tool support typically only aids the graphical representation of threats, rather than the analysis of threats. The lack of validation across different domains questions the applicability of analysis techniques to current trends in software engineering. Internet of Things (IoT) and Cyber-Physical Systems (in particular automotive) have recently been attracting a lot of attention.

IoT systems typically consist of a large amount of relatively small devices and sensors with limited capabilities functioning as individual agents to achieve goals. These interconnected devices are commonly analyzed individually, thus their vulnerabilities are well known. Yet new vulnerabilities may arise once the devices are connected. Therefore, a knowledge-base of threats and mitigations to the known vulnerabilities could aid in automating threat analysis for IoT devices and in maintaining the quality of analysis outcomes. Recent efforts have proposed analysis approaches in the domain of IoT formalizing the cyber-physical interactions including the malicious perspective. For instance, Mohsin et al. [160] introduced a formal risk analysis framework based on probabilistic model checking. Their framework is able to generate system threat models, which are used to formally compute the likelihood and cost of attacks. Further, Agadakos et al. [161] have introduced an approach for modeling cyber-physical attack paths in IoT using Alloy. Their approach also ultimately generates potential threats. Non-formal approaches supporting aspects of threat analysis in IoT have also been proposed. For instance, Geneiatakis et al. [162] have built an attacker model covering security and privacy threats in a typical IoT system. Regarding usability Mavropoulos et al. [163] presented a tool that supports security analysis of IoT systems. Rather than aiding the analysis procedure, the tool helps to visualize assets, threats and mitigations.

In the *automotive* domain, Threat Analysis and Risk Assessment (TARA) approaches are summarized by Macher et. al. [14]. TARAs summarized in this review use traditional threat analysis approaches (such as CORAS [127] and Attack trees [34]) as well as approaches tailored for the automotive (e.g. HEAVENS [14] and SAHARA [164] are adaptations of STRIDE [9], EVITA [165] is based on Attack trees [34], etc). Threat analysis of novel autonomous vehicles is extremely lengthy and complex due to heavy safety and security requirements and compliance to standards (e.g. ISO 26262 [166]). The automotive industry to this day relies predominantly on threat analysis performed manually by experts. Yet there is a need to semi-automate threat analysis procedures due to scarce resources (i.e. security experts). A risk-centric light-weight threat analysis technique could facilitate the identification of the most important threats in only a few sessions. In order to ensure compliance to safety and security standards, the problematic parts of the system still need to undergo a systematic threats analysis.

In summary, significant effort has been invested in researching failure analysis in the domains of Cyber-Physical Systems (e.g. Martins et al. [167]), IoT and automotive due to the required compliance to safety standards. Therefore, mature hazard analysis (safety) techniques have already been established (e.g. failure mode and effects analysis (FMEA) [168] and fault tree analysis

(FTA) [169]). On the other hand, there seems to have been less focus on threat analysis techniques, particularly in Agile development and DevOps, where security is often not a business priority.

2.4.5 Ease of adoption

In the space of threat analysis approaches, tools have been used for three main purposes: i) partially automating the analysis procedure, ii) graphically representing threats to the system and (iii) facilitating the analysis execution (i.e. helping the analyst to follow the procedure).

Semi-automated approaches utilize tools for the purpose of *automating* a part of the analysis procedure (Such as Berger et al. [45], Almorsy et al. [44] and Whittle et al. [102]). For instance, Whittle et al. [102] extended an existing tool in order to automatically weave mitigation scenarios into a set of core behavior scenarios. The authors are able to then generate a new set of finite state machines including both the initial behavior and the behavior including the mitigations. Finally, they execute the attack behavior on the new set of finite state machines to determine the success of the attack.

Manual threat analysis approaches are supported by tools for the purpose of retaining the *structure of the analysis* technique. For example, MUC (and MUC maps [141]) are a form of templates used in the process of analysis. The tools supporting threat analysis with MUCs only provide the required elements to model the misuse, such as graphical elements to represent attackers with an empty template for defining their abilities. Meanwhile threat identification is not supported by tools and is considered a brainstorming task. Similarly, Microsoft Threat Modeling Tool[9] provides the visual elements (e.g. boxes, arrows, ellipses, etc.) needed to create DFDs. To some extent, this tool also facilitates the proper execution of the analysis, as it generates categories of threats for each DFD element. The generated categories guide the analyst through the analysis procedure of the technique. However, threat categories generated based on the threat-to-element mapping table only provide a hint of what type of threats could be identified. Similarly, the open source SeaSponge[10] threat modeling tool primarily serves as a graphical aid to represent threats on a system model. Some primary studies present tools whose purpose is both to aid automation of analysis and provide graphical representation. For instance, Sheyner et al. [47,153] present a tool for generating and analyzing attack graphs.

Tools serving the sole purpose of *graphical representation* are fairly straight forward to use just by drag-and-dropping elements on an empty canvas. Anyone with basic computer skills could easily use them. However, such graphical tools do not support threat identification and prioritization. The correctness and completeness of the results submitted by an engineer using such tools is not assured. One could argue that more expertise is required for the proper execution of the analysis using tools that only aid the graphical representation of threats. Our assessment suggests that tools supported by knowledge-base could to some extent leverage the security (and domain) expertise required for threat analysis. Further, introducing quality assurance features is very important for a novice analyst. Finally, partial automation could help speed up the analysis

[9] https://www.microsoft.com/en-us/download/details.aspx?id=49168

[10] https://github.com/mozilla/seasponge

to facilitate efficient training of junior analysts. Several primary studies have the potential to be extended with tool support also targeting engineers, namely Berger et al., Almorsy et al. [44], Chen et al. [121], KAOS [138–140], Halkidis et al. [135], LINDDUN [11], STRIDE [9, 105]. In summary, tool support seems to be a common trend in the primary studies, yet tool proposals are preliminary with limited validation.

2.5 Threats to validity

We consider *internal* and *external* threats to validity, as defined in [170]. Considering that substantial work was done by a single researcher, we consider a risk of subjectivity as an internal threat to the validity of this study. The bias introduced by the first author was mitigated by including random quality controls into the review process, particularly during the selection of primary studies and data extraction.

Furthermore, in this work we restrict our search of the literature by considering only top venues available in the digital libraries mentioned in Section 2.2.2. Consequently, we raise the risk of considering a non-representative subset of the relevant existing literature, thus harming the validity of our conclusions. However, as per focusing the search on top venues, we are confident that the selected papers represent the most influential work done in the area of secure design in software engineering.

In general, the validity of results of systematic literature reviews depend heavily on the external validity of the selected studies. We attempted to mitigate this issue by adopting a conservative exclusion criteria, disregarding grey literature papers, position papers and short papers (< 3 pages). Finally, due to resource limitations, not all aspects could be extracted from the data. For instance, further investigations could have been made regarding the learnability in relation to tool support of the identified threat analysis techniques.

2.6 Related work

To the best of our knowledge this is the only systematic literature review on threat analysis techniques. However, recently Cheung [104] has contributed with a brief literature review of 8 threat analysis techniques. The main purpose of this work was to identify the added value and impact of adopting threat analysis techniques to cyber-physical systems of public water infrastructures. The author summarizes a subset of the primary studies analyzed in this work.

Threat analysis is used for the main purpose of security requirement elicitation or refinement. Hence, security requirements engineering approaches may include aspects of threat analysis. We continue to address the related work in the areas of security requirements engineering and risk analysis and assessment.

2.6.1 Security requirements engineering

Mellado et al. [171] performed a systematic literature review concerning security requirements engineering methodologies, processes, frameworks and techniques. The authors selected 51 primary studies to investigate. Some of these studies are overlapping with our selection of primary studies (namely [172], [134], [35] and [102]). Among assessing the selected studies based on a smaller set of criteria, the authors additionally present the integration of primary studies with security standards. Our work could also be extended to include the integration of primary studies with security standards, which would further aid practitioners.

Similarly, Salini et al. [173] have published a survey on security requirements engineering approaches. The authors present and compare SRE issues and methods. Additionally, the authors stress the importance of threat analysis in the early stages of software development. Yet, this survey focused on reviewing SRE frameworks and processes (e.g. Security Quality Requirements Engineering methodology (SQUARE) and Security Requirements Engineering Process (SREP)).

Further, Fabian et al. [174] contributed with a conceptual framework for SRE with a strong focus on security requirements elicitation and analysis. The authors use the proposed framework to compare several SRE approaches. Similar to our study, Fabian et al. also investigate problem frames and other UML-based modeling approaches. Additionally, the authors also assess the quality of outcomes for the selected studies. In contrast to this study, the authors perform an unsystematic comparison of SRE methods, as opposed to a systematic comparison of threat analysis techniques.

Munante et al. [175] have performed a review of SRE methods with a focus on risk analysis and model-driven engineering (MDE). The purpose of their work was to identify which SRE methods are compatible with existing risk analysis and MDE approaches. To this aim, Munante et al. have analyzed the existing work and concluded that KAOS and Secure i* are the most compatible SRE methods with a model-driven approach. They also concluded that extending them with risk analysis concepts is feasible. Despite the overlap in primary studies of this work, Munante et al. have based their analysis on a smaller set of assessment criteria and have done so unsystematically.

Daramola et al. [176] have published a comparative review on i*-based and use case-based security modeling approaches. Their main findings show that both categories of approaches show conceptual similarities in the modeling aspects and method process. They also found several differences between both categories of approaches (namely, representational, supported activities and techniques, quality of outcomes and tool support).

Kriaa et al. [177] have performed a survey of approaches combining safety and security for industrial control systems. The authors contribute with highlighting the main commonalities, differences and interconnections between safety and security in industrial control systems. A subset of the reviewed approaches overlap with our primary studies (namely, CORAS, MUC). Their review also considers Failure Mode and Effects Analysis (FMEA), Failure Mode, Vulnerabilities and Effect Analysis (FMVEA), Fault Tree Analysis (FTA), which are based on attack trees. In contrast to their work, this study only investigates threat analysis techniques from the security perspective.

2.6.2 Risk analysis and assessment

Latif et al. [178] present a systematic literature review in the field of cloud computing with a focus on risk assessment. The purpose of their work was to categorize the existing approaches and explore which areas need further investigation. The authors selected 31 primary studies and have looked into the existing risks in cloud computing from the perspective of a customer and a provider. Their main finding is that topics such as data security and privacy are widely investigated, whereas physical and organizational security, have received less attention. However, their literature review is narrowly scoped only to one domain and does not assess the characteristics of the selected works.

Cherdantseva et al. [179] present a state-of-the-art review of the literature on cyber security risk assessment methods for SCADA systems. The authors have selected and examined 24 risk assessment methods. They provide descriptions of the methods and assess them with an elaborated criteria. Among other methods, the review also includes attack trees, petri net analysis, attack and defense modeling and CORAS. Interestingly, the authors propose several challenges for future work, some of which are in line with the findings of this work, namely (i) need for improving the validation of methods (ii) overcoming the attack-failure orientation (in this work referred to as determining the stopping condition), (iii) lack of tool support. Yet, their review has a strong focus on risk assessment methods in the context of SCADA systems.

Dubois et al. [180] have contributed with a systematic approach to define a domain model of information system, which is used to compare, select of improve security risk management methods. The authors provide a literature review as part of their study, which also include threat analysis approaches CORAS, OCTAVE and Common Criteria. Yet, this study contributes with an ontology, rather then systematically reviewing the literature.

Raspotnig et al. [103] have compared risk identification techniques for safety and security requirements. The purpose of their work was to investigate whether and how the techniques can mutually strengthen each other. Among other methods, the authors also assess attack trees, MUC and KAOS. Similar to this study, the authors also look into stakeholders (in this study target audience) of the selected methods. One of their main findings was that security techniques can be strengthened by including better stakeholders and communication descriptions, while the safety techniques can benefit from a tighter integration between the risk identification and development processes. However, this study does not look into safety or the interaction between safety and security.

2.7 Conclusions and future work

In this study, we have performed a systematic review of 26 threat analysis approaches for secure software design. We have developed detailed assessment criteria reflecting our research questions, presented in Section 2.3. Our search strategy included an automatic search of three digital libraries and snowballing. The data was extracted from the primary studies according to the assessment criteria. The main findings of this study show that the existing techniques lack in quality assurance of outcomes. Furthermore, the techniques lack maturity, validation and tool support. Finally, they lack a clear definition of when the analysis procedure is done.

As per the results discussed in Section 2.4, we identify three possible directions for future work. First, a connection (feedback) between the intended and actual architecture might aid in understanding the reality of analysis outcomes. The quality of outcomes might only provide insightful speculations without a clear link to the actual architecture. Further, other architectural design decisions might have lead to architectural decay [181], causing a disconnection to the “as-planned” security. To this aim, a formal language for design-level threat analysis may aid in establishing the link to the extracted architecture (e.g. by means of adapting dynamic and/or static reverse engineering approaches). Regarding the Definition of Done, we believe that further investigations are needed to understand the effects of composing analysis outcomes of subsystems. To this aim, the assets play an important role as border elements between subsystems (e.g. middle-ware). Further, a semi-automated way of composing analysis outcomes might facilitate analysis reuse for products in different stages of the SDL. Finally, in the context of DevOps and Agile, we believe that analysis velocity is preferred over analysis systematicity. Therefore, an analysis approach focusing on most important assets might be more appropriate for such organizations. To this aim, we are evaluating a risk-first lightweight approach for finding the most important threats sooner in the analysis procedure [182].

Chapter 3

Paper B

This chapter is based on
Two Architectural Threat Analysis Techniques Compared,

written by
K. Tuma and R. Scandariato,

published in
Proceedings of the 12th European Conference on Software Architecture (ECSA 2018), 2018.

Abstract

In an initial attempt to systematize the research field of architectural threat analysis, this paper presents a comparative study of two threat analysis techniques. In particular, the controlled experiment presented here compares two variants of Microsoft's STRIDE. The two variants differ in the way the analysis is performed. In one case, each component of the software system is considered in isolation and scrutinized for potential security threats. In the other case, the analysis has a wider scope and considers the security threats that might occur in a pair of interacting software components. The study compares the techniques with respect to their effectiveness in finding security threats (benefits) as well as the time that it takes to perform the analysis (cost). We also look into other human aspects which are important for industrial adoption, like, for instance, the perceived difficulty in learning and applying the techniques as well as the overall preference of our experimental participants.

3.1 Introduction

After decades of research and knowledge transfer in the field of "security by design", the software-intensive industries have absorbed the idea that security needs to be addressed throughout the software development lifecycle. Building Security In Maturity Model (BSIMM) [183] collects statistics from 95 companies and gauges their level of adoption with respect to several secure software development techniques. According to the report, security-specific code analysis techniques have successfully found their way into the industrial practice, as two thirds of the surveyed companies adopt them routinely. In this respect, the availability of well-known automated tools has helped significantly. *Architectural threat analysis* is another important pillar of building more secure software and the above-mentioned BSIMM report mentions that about one third of the surveyed companies use architectural threat analysis techniques, like Microsoft's STRIDE [9], attack trees [39], Trike [148], CORAS [127], PASTA [29], threat patterns [124], to cite a few.

Working in collaboration with our industrial partners from the automotive industry, we noticed that Microsoft's STRIDE is well-known and often used. In particular, our partners use the so-called STRIDE-per-element version. In this version, each component of the software system is considered in isolation and scrutinized for potential security threats. However, practitioners advocate for a threat analysis technique that allows them to analyze end-to-end scenarios where several components interact (e.g., to provide a given functionality). In this respect, the STRIDE-per-interaction variant could be more appropriate, as in this variant the analysis has a slightly wider scope and considers the security threats that might occur in a pair of interacting software components. On the other hand, there are also truly end-to-end analysis techniques, like for instance the one proposed by Tuma et al. [182]. From our perspective, it is interesting to study how these alternative techniques differ across the spectrum (analysis of isolated components vs analysis of pair-wise interactions vs analysis of end-to-end scenarios) in terms of performance. In essence, which approach to threat analysis produces more results in a faster way? Consequently, this study focuses on the differences between the analysis of isolated components and the analysis of pair-wise interactions. In current work, we are also comparing the analysis of isolated components with the analysis of end-to-end scenarios.

In the latest publication by Shostack [9] describing Microsoft's STRIDE, the author describes two variants that are dubbed 'STRIDE per element' (analysis of isolated components) and 'STRIDE per interaction' (analysis of pair-wise interactions). A more detailed description of the two is provided in Section 3.2. In our study, we divide our participants (110 master students) into two treatment groups (Element vs Interaction), each using one of the two variants of STRIDE to analyze the architectural design of an Internet-of-Things system. For replication purposes of this study, we have created a *companion web-site [184]*, where all the material used during the experiment is available. The study analyzes and compares the effectiveness of the two variants in unearthing security treats (benefits) as well as the time that it takes to perform the analysis (cost). We also look into other human aspects which are important to adoption, like the perceived difficulty in learning and applying the techniques as well as the overall preference of our participants.

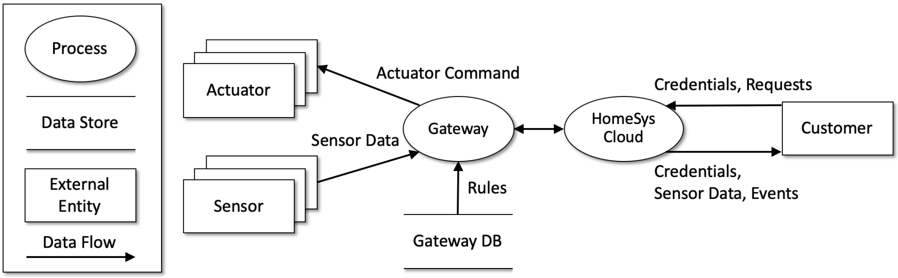


Figure 3.1: A high-level DFD of the experimental object.

The rest of the paper is organized as follows. Section 3.2 provides a primer on the STRIDE variants. Section 3.3 describes the experiment and states the research hypotheses. Section 3.4 presents the results, while Section 3.5 discusses them. The threats to validity are listed in Section 3.6. Section 3.7 discusses the related work and Section 3.8 presents the concluding remarks.

3.2 Treatments

STRIDE is a threat analysis approach developed to help people identify the types of attacks their software systems are exposed to, especially because of design-level flaws. The name itself is an acronym that stands for the threat categories of Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege. For the definition of threat categories, we refer the reader to the documentation of STRIDE [9].

The analysis is based on a graphical representation of the system architecture as a Data Flow Diagram (DFD). As shown in Figure 3.1, a DFD represents how information moves around in a software-based system. The diagram consists of processes (active entities), data flows (exchanged info), external entities (e.g., users or 3rd parties), data stores (e.g., file system) and trust boundaries.

The first step in applying the STRIDE methodology is to create a DFD using the available system documentation. The second step is a systematic exploration of the DFD graph in order to identify the threats. The two STRIDE variants differ in how this exploration is carried out.

STRIDE per element. Using this approach, the analyst visits every element in the diagram (e.g., starting in the top-left corner). For each element type, STRIDE advises looking into a subset of threat categories. To this aim, STRIDE provides a table mapping element types to threat categories. For instance, the Sensor in Figure 3.1 is an external entity, therefore according to STRIDE, the analyst should look into Spoofing and Repudiation threats. For each pair of element type and threat category STRIDE also provides a catalog of example threats that can be used for inspiration by the analyst. With reference to the Sensor, one provided example threat for spoofing a hardware device is IP spoofing.

STRIDE per interaction. This technique adopts an approach of systematically visiting the interactions in a DFD. Interactions are patterns of DFD elements connected via data flows. The analyst has to first identify the interactions. For instance, “Sensor sends sensor data to Gateway” is a match for the above-mentioned interaction. For each type of interaction STRIDE again advises looking into a subset of threat categories and provides a catalog of example threats. For example, when an external entity is passing input to a process, the analyst is advised to look into Spoofing and Repudiation threats. If there is no logging in place, the Gateway is able to deny having received sensitive information from the Sensor.

3.3 The experiment

This section presents the design of a controlled experiment, conducted with participants in an academic setting.

3.3.1 Experimental object

As depicted in Figure 3.1, the Home Monitoring System (HomeSys) is a system for remote monitoring of residential homes. The purpose of this system is to provide necessary tools for customers to automatically receive and manage notifications about critical events in their homes. The system consists of a smart home gateway which communicates with sensors and actuators and a cloud system which collects data from the gateways and offers a dashboard to the customers. Sensors are analog or digital hardware devices that produce measurements and send them to the gateway. This system includes sensors that detect temperature, humidity, smoke, etc. Actuators are hardware devices that can receive commands from the gateway, like for instance, taking a picture, activating a buzzer or flicking a switch. The gateway is a hardware device which relays measurements to the cloud (via a 3G or WiFi network) and manages the actuators in the residency. The HomeSys cloud is a software system that communicates with the gateways and provides services for the customers, as well the operators of the system.

The system documentation (about 30 pages) includes (1) the description of the problem domain with scenarios, (2) the requirements of the system and (3) a hierarchical architectural description documented in UML. For instance, the documentation includes a UML deployment diagram. The complete description of the system is available with the experimental material [184]. The participants worked on the HomeSys system throughout the entire course before entering the experiment. Therefore, they were very familiar with the object of the experiment and had enough knowledge about the system to understand the problem and complete their task.

3.3.2 Participants

The participants of this study are 110 first-year master students of Software Engineering, attending a course on “Advanced Software Architecture”, taught by the experimenters. In order to gather sufficient data, we have repeated the experiment for two consecutive academic years (2016 and 2017). The participants performed the assigned tasks in teams. Each year, the participants were randomly grouped into teams of about 4 students and assigned one analysis

Table 3.1: Answers to the entry questionnaire.

Questions and answers [%]			
1. Do you have any working experience in software development outside the university?			
Yes (63)	No (37)		
2. How would you describe your working experience outside the university?			
Profit (39)	Non-profit (8)	Both (27)	NA (26)
3. How would you rate your level of expertise as a programmer?			
Very insufficient (1)	Limited (17)	**Adequate (58)**	Advanced (24)
4. How would you rate your level of familiarity with software design, including the use of UML?			
Very insufficient (2)	Limited (33)	**Adequate (63)**	Advanced (2)
5. How would you rate your level of expertise in security?			
Very insufficient (16)	**Limited (65)**	Adequate (16)	Advanced (3)

techniques (i.e., treatment groups). In total, we have assigned 14 teams to the Element and 13 teams to the Interaction treatment.

We have collected information with a short questionnaire before the study took place to investigate the background of the participants. As shown in Table 3.1, it included questions about participants' work experience and perceived familiarity with task-related concepts. Most participants have had previous experience in software development outside the university and consider to have adequate knowledge about software design and programming. A large majority of the participants are able to use UML, which is relevant as the study object is documented in such language. The course does not require background knowledge of information security, hence the participants consider to have limited expertise in this area, as expected.

3.3.3 Task

The teams were presented with the same task on the same experimental object. The task was divided into two sub-tasks: participants were asked to (1) build a DFD based on the provided architectural documentation and (2) analyze the DFD according to the assigned technique.

During the training, participants were provided with guidelines for creating the DFD. First, they had to create a DFD by mapping the nodes from a given deployment diagram into DFD elements. Second, the participants had to use the rest of the documentation (e.g., component and sequence diagrams) to refine the DFD and identify the data flows. The details of training are available online [184].

The second sub-task required a systematic analysis of the DFD according to the techniques described in Section 3.2. The analysis results had to be documented in a report and submitted electronically. The report had to contain a list of identified threats and corresponding descriptions. Threat descriptions were made according to a provided template (available online [184]). The purpose of the template is to simplify and standardize the analysis of the reports. Note that the task has been performed during a supervised lab session. In the lab, the teams were instructed to keep an informal log of the identified threats (lab notes). The preparation of the official report had to be done after the supervised lab. However, we have monitored that the reports did not contain more threats with respect to the work done in the lab (e.g., by taking snapshots in the lab). The snapshots taken during the lab were compared with the final report to capture any threats identified outside the supervised lab. We have not recorded any discrepancies between the snapshots and the reported threats.

Finally, we asked the teams to keep track of the time they spent. To this aim, the teams were instructed to use an online time-tracking tool (`www.toggl.com`) and submitted their time-sheets electronically at the end of the supervised lab.

3.3.4 Execution of the study

The experiment is positioned at the end of a course on software architecture. The topic covered in the experiment aligns with the course content. Participation in the study contributes to the teaching objectives of the course, hence participants were highly motivated. For more details about the experimental material please refer to the companion web site [184].

Entry questionnaire. A few weeks before the beginning of the study, the participants have been asked to fill in a brief questionnaire about their knowledge and background (see Section 3.3.2).

Training. As part of training for the experiment, the participants attended 2 lectures (mandatory 4 hours of training). In the first lecture (2 hours) the participants got an introduction to secure design and the use of the DFD notation. The lecture also included a practical exercise on how to build a DFD for a system of similar size as HomeSys. For the second lecture (2 hours) participants were split according to their assigned treatment group. Each group received a dedicated lecture explaining the philosophy of STRIDE specific to their treatment group. In addition, participants received only documentation specific to their treatment group. This was done in order to limit the problem of treatment diffusion. An overview of the HomeSys documentation was also given in the second lecture. The students were more than familiar with the system, but the experimenters wanted to be sure that they would be able to navigate the documentation without hiccups. Finally, in a lab session preceding the experiment, the participants were familiarized with the time-keeping tool.

Supervised lab. The experiment took place in one lab session of 4 hours. The session was supervised by the authors and two teaching assistants. At the beginning of the lab, the authors explained the experimental protocol to the participants, e.g., by summarizing the task, mentioning all the provided material, and reminding the participants about the time-tracking tool. Each team was provided with a printed copy of task description, the relevant book chapters, and the documentation of HomeSys. The teams performed the assigned task and kept track of their work by writing lab notes.

Report. The participants were given about a week to write a report documenting the threats they had found during the lab. To this aim, they used their lab notes. Each report contained a figure of the DFD and a list of identified threats, where each threat was documented according to a provided template [184]. In particular, each threat is described with a title, a position in the DFD where the threat is located, a threat category (STRIDE), required attacker capabilities, and a detailed description of the threat itself. The participants were also asked to document their assumptions about the system.

Exit questionnaire. At the end of the lab session, the participants were asked to fill in an exit questionnaire. Access to the questionnaire was open for a week after the lab session had finished, during which time a few reminders were sent by email. As discussed later, this questionnaire is meant to validate some experimental assumptions (e.g., the participants understood the task and were adequately prepared to carry it out) and to collect additional information about the treatments (e.g., the perceived difficulty of the tasks).

3.3.5 Measures

We have collected the measure of effort (in minutes) spent by each team on both sub-tasks (DFD creation and threat analysis).

We have also collected the measure of true positives (TP), false positives (FP) and false negatives (FN). True positives are reported threats that are assessed as correct by the experimenters in light of the analyzed DFD and the security assumptions made by the team. False positives are wrong or unrealistic threats reported by the team. Finally, false negatives are threats that are present in the analyzed system but had gone unnoticed by the team.

In order to record the correct threats a "ground truth" has been created by the first author. Incidentally, we decided to let the teams produce their DFD as this activity is an integral part of threat analysis in practice. Ergo, the teams have analyzed slightly different DFDs. A ground truth was built *for each team* to ensure a correct evaluation. For each report, the ground truth was used to identify the correct, incorrect and overlooked threats. In particular, a threat is considered correct if (1) it is identified at the correct location, (2) it is correctly categorized, (3) it has some impact on system assets, (4) the description of the threat agent is correct and (5) the description provided by the team is realistic from a security perspective and does not contradict their assumptions. Oftentimes the teams reported the same threat more than once, using a different title. A threat (either correct or incorrect) that is identified more than once is marked as a duplicate. Note that duplicated threats are intentionally not considered as TP or FP.

3.3.6 Hypothesis

We have adopted a standard design for a comparative study of one independent variable with two values, i.e., the two treatments of Element and Interaction. Our study investigates three dependent variables: productivity, precision, and recall. In this study we define the *productivity* ($Prod$) of a team as the number of correct threats (TP) per time unit. For each team, *precision* (P) is the percentage of correctly identified threats out of the total number of reported threats ($TP/(TP + FP)$). *Recall* (R) is the percentage of correctly identified threats out of the total number of existing threats ($TP/(TP+FN)$).

We use the Wilcoxon statistical test to determine whether there is a statistical difference in the three dependent variables across the two treatment groups. Accordingly, the null hypotheses are as follows:

H_0^{Prod} : *There is no statistically significant location shift between the average productivity of the two treatment groups.*

H_0^{P} : *There is no statistically significant location shift between the average precision of the two treatment groups.*

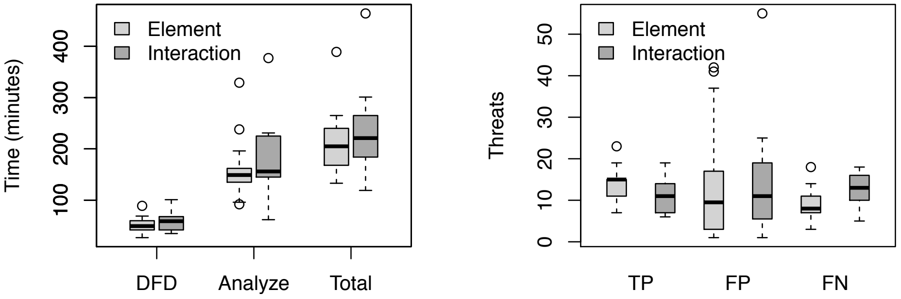


Figure 3.2: Average time per sub-task and total time of treatment groups (left) and true positives, false positives, and false negatives (right).

H_0^R : *There is no statistically significant location shift between the average recall of the two treatment groups.*

3.4 Results

In this section, we present the results of the study and answer to research questions. All statistical tests have been performed using the two-sample Wilcoxon test of independence with a level of significance equal to 0.05.

3.4.1 True positives, false positives, and false negatives

Figure 3.2 reports the number of TP, FP and FN per treatment group. We have observed slightly better averages for the Element group, compared to the Interaction. Namely, the TP is higher (Element: 14.3, Interaction: 11.6), the average number of FP is lower (Element: 14.2, Interaction: 15) and the average number of FN is also lower (Element: 8.8, Interaction: 11.8). However, the analysis shows that there are no significant differences between the amount of TP, FP and FN across treatments groups.

The average number of TP, FP and FN per threat category is depicted in Figure 3.3. Overall, both treatment groups visibly focused less on Repudiation and Elevation of Privilege threats compared to other threat categories. A statistical analysis shows that there are significant differences between the TP of the Denial of Service (p-value = 00.02) and Tampering (p-value = 00.007) threat categories across treatments. For the Denial of Service threat category, the Element treatment group identified on average more TP (statistically significant, p-value = 00.02) and less FN (not significant). For the Tampering threat category, the Element treatment group identified on average less FN, more TP (statistically significant, p-value = 00.007), and less FP. This might be due to the two methods providing different mapping tables (from DFD to threat categories [9]). The Interaction has a lower rate of mappings to the Denial of Service threat category (8/72 = 11% vs 3/20 = 15%). We have also computed the recall when the Denial of Service threats are removed. The results stay similar to what is reported in Figure 3.4, i.e., the median recall is not "driven" by the Denial of Service category. Incidentally, there is a similar situation in the mappings for the Tampering category (3/72 = 4%

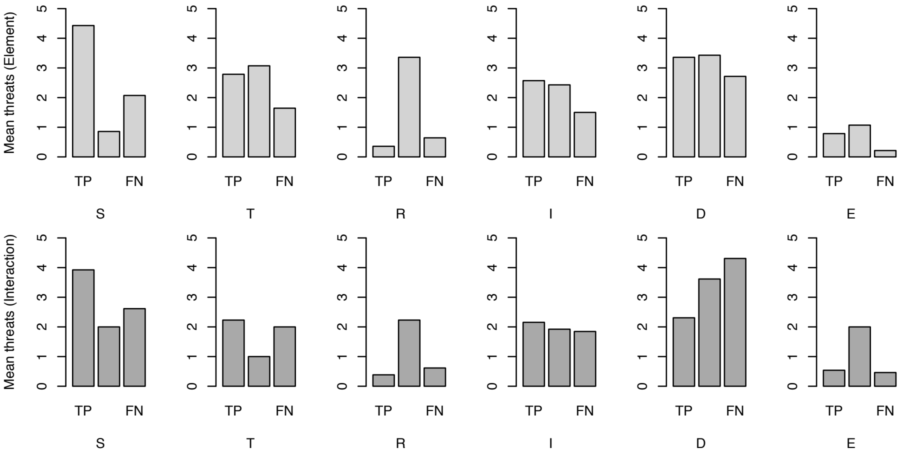


Figure 3.3: The mean number of identified threats for per element (top) and per interaction treatment (bottom).

vs 3/20 = 15%). Identified threats from other categories do not differ across treatment groups.

3.4.2 RQ1: Productivity

As shown in Figure 3.2, the average time spent by the teams performing STRIDE-per-element is 3.5 hours, whereas the average time spent per teams performing STRIDE-per-interaction is 3.95 hours (not statistically significant). There is a noticeable difference between time spent on the sub-tasks, with the analysis time being dominant. When looking at differences across the treatments, the ELEMENT group was on average faster in performing both sub-tasks (not statistically significant). It is interesting to notice, that even though both treatments followed the same guidelines for DFD creation, the INTERACTION group created on average DFDs with more elements (discussed in Section 3.5).

The overall productivity of a technique depends on the amount of correctly identified threats (TP). The average productivity of the ELEMENT group is **4.35** TP/h.[1] The INTERACTION group turned out to be less productive (**3.27** TP/h). However, the difference is not statistically significant and, hence, the null hypothesis H_0^{Prod} cannot be discarded.

As a possible explanation for lower productivity of the INTERACTION treatment, we highlight that the documentation of the STRIDE-per-interaction variant is more complex with regard to mapping threats to interactions. As mentioned by Shostack, "STRIDE-per-interaction is too complex to use without a reference chart handy" [9]. Such a reference chart was available to the participants, yet the complexity might still have been too high.

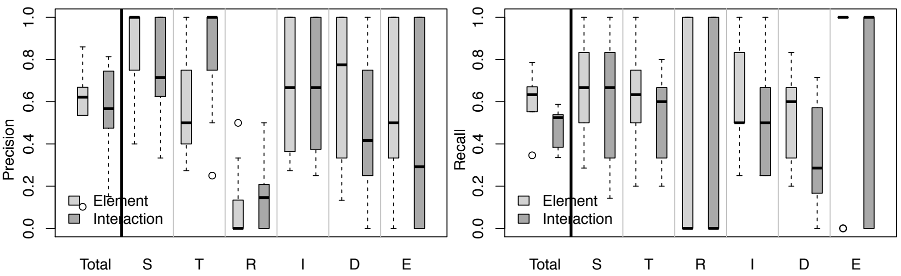


Figure 3.4: Precision (left) and recall (right) aggregated across threat categories.

3.4.3 RQ2: Precision

Figure 3.4 presents the precision (i.e., the correctness of analysis) of the two treatment groups as an aggregate (left-hand side) and across each individual threat category (right). Overall, the mean precision is **0.60** for both treatment groups. Therefore, null hypothesis H_0^P cannot be rejected. We have also analyzed the differences in precision within threat categories. There is a statistically significant difference in the precision of Tampering threats (Element: 0.58, Interaction: 0.81, p-value = 00.027). A difference can also be observed for the Denial of Service category (not statistically significant).

3.4.4 RQ3: Recall

Shostack states that the "STRIDE-per-interaction leads to the same number of threats as STRIDE-per-element" [9]. Yet in our study, the number of reported threats was higher for the Element treatment, especially the number of FP (see Figure 3.2). Figure 3.4 presents the recall (i.e., completeness of analysis) of the two treatment groups as aggregate (left-hand side) and across each individual threat category (right). The average recall for the Element treatment is **0.62**, whereas the average recall of the Interaction is **0.49**. The difference is statistically significant (p-value = 0.028). Therefore, null hypothesis H_0^R can be rejected. We have also analyzed the differences within each threat category and found a statistically significant difference in the recall of the Denial of Service category (Element: 0.55, Interaction: 0.34, p-value = 00.014). One possible explanation relates to the fact that the Element group tends to create smaller DFDs, as discussed in Section 3.5. Alternatively, the difference could be linked to the fact that the threat examples in the documentation are more extensive in case of the Element treatment and the documentation of the Interaction treatment is more complex to navigate (as mentioned in Section 3.4.2). These are interesting hypotheses for future studies.

3.4.5 Exit questionnaire

In summary, the two treatments displayed a statistically significant difference only with respect to recall, with the Element group reporting more complete results (13% better). It is also important to appreciate how the two variants are perceived by the participants. This could have an impact on the successful

[1] Scandariato et al. [10] reported an average productivity of 1.8 TP/h.

adoption of the technique and, hence, become a deciding factor beyond the performance indicators investigated in the three research hypotheses.

To this aim, we asked the participants to fill in a questionnaire at the end of the experiment. Due to space limitations, the questions and the answers are not shown here. They are available on the companion web-site [184] under the "Questionnaires" tab. To investigate differences across the treatment groups, we have performed a Cochran-Mantel-Haenszel test (similar to the Chi-square test) with a level of significance equal to 0.05.

In general, participants from both treatment groups agreed about having a clear understanding of the task, though they were sufficiently prepared and were familiar with the experimental object. Overall the task was not too difficult, while the first sub-task of creating the DFD was perceived easier than the second sub-task of analyzing the DFD (for both treatments). According to the documentation, "STRIDE-per-element is a simplified approach designed to be easily understood by the beginner" [9]. This implies that STRIDE-per-interaction is perceived by Microsoft as the technique to be used in production. However, according to our results, the techniques are the same (i.e. per element is not simpler than per interaction) in terms of productivity and precision. Concerning the learnability, the teams from both treatment groups agreed that the techniques they used were easy to understand and learn (no significant differences). Although the teams were given sufficient time to carry out the task, both treatment groups perceived the techniques as lengthy and tedious.

The participants from both treatment groups mostly believed that 50-75% of their identified threats were correct. This is a fairly accurate estimation according to the observed precision in this study (0.6). Interestingly, participants were slightly less confident about the completeness of their analysis. The aggregate recall over all teams (regardless of the treatment group) is 0.5 and only less than half of the participants (47.7%) have this perception. Finally, participants generally liked the technique they used but were not especially fond of it either. No significant differences were observed across treatments.

3.5 Discussion

DFD. The participants followed precise guidelines for DFD creation [184]. As such, the created DFDs have limited variability as well as consistent quality, e.g., we did not find many mistakes in the DFDs. In this study, we have made a deliberate choice to minimize the influence of the DFD creation (common to both treatments) and focus on the alternative techniques of analyzing the DFD (the key difference between the treatments). Figure 3.5 depicts the number of DFD elements in the models created by the teams. On average, the teams created DFDs with about 26 data flows, 6 processes, 4 data stores, 3 external entities, and 3 trust boundaries. A few differences can be noted. On average, we observed a smaller number of DFD elements in the Element group (37.4) compared to the Interaction group (41.9). This difference is consistent across the different element types, yet not statistically significant.

Mistakes. As reported in Figure 3.5, the teams sometimes reported several threats more than once. Duplicated threats are considered to slow down the

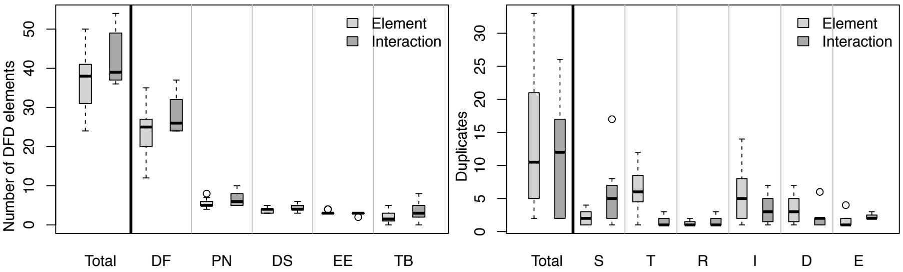


Figure 3.5: The average number of DFD elements (DF=data flows, PN=process nodes, DS=data stores, EE=external entity, TB=trust boundaries) (left) and the average number of duplicated threats (right).

analysis process, especially during threat prioritization. Note that the results about productivity (see Section 3.4.2) are not affected by duplicates, as they were discarded.

Most commonly, threats are duplicated due to (i) threat 'fabrication' or (ii) a misuse of the reduction technique. We consider threat fabrication as mistakenly identifying threats in order to achieve complete coverage of the STRIDE category mapping table. Proposed by STRIDE, threat reduction is a technique that aims towards minimizing the number of DFD elements that have to be analyzed. In particular, the reduction enables coupling the elements of the same type in order to analyze them at once. In other words, the threats identified for one DFD element may apply to other elements of the same type.

About 30% of all reported threats were duplicated. Of those, the majority belonged to the ELEMENT group (65%). The mean number of duplicated threats identified by the ELEMENT teams is bigger (6) than the INTERACTION teams (5) (not statistically significant). However, there is a significant difference in the amount of Tampering duplicates across treatments (ELEMENT: 6.27, INTERACTION: 1.67, p-value = 0.042). Incidentally, we also observed more Spoofing duplicates in the INTERACTION treatment. A possible explanation for fewer duplicates in the INTERACTION group is that the notion of interaction patterns might lend itself useful to a correct use of the reduction technique.

Interestingly, we observed that most teams correctly identified more outsider threats than insider threats. Very often, the reported insider threats were just unrealistic and assessed as false positives.

Analysis focus. Overall (including duplicated threats and *FP*), both treatment groups have focused their analysis on 'border' elements of the system, as well as the data flows that pierce through trust boundaries. The reports of the ELEMENT treatment group contain more threats to processes, external entities, and data stores compared to threats to data flows. In contrast, the reports of the INTERACTION treatment group contain more threats to data flows compared to threats to other types of DFD elements. In general, all teams were more likely to identify correct threats to the data flows crossing a trust boundary. This confirms the usefulness of using trust boundaries to focus the attention of the analyst. However, there is a lack of precise guidelines for how and where trust boundaries should be placed, as our teams showed uncertainties in this regard. The teams were more likely to falsely identify threats (FP) to processes and data stores. The participants found the most commonly known threats (e.g., phishing, SQL injection, stealing for credentials or account). Coincidentally, these are more commonly identified on data flows. Correctly identifying a Tampering threat to a data store within system boundaries would mean finding a way to by-pass the system access control or even overcome obstacles like file-locking or other system defenses. This kind of threat requires correct assumptions and more security background. Unfortunately, most teams did not document many (if any) assumptions.

3.6 Threats to validity

The time spent for performing the task was measured by the participants themselves. To mitigate this threat, we have continuously reminded the participants to pause and continue measuring time during breaks. The amount of work that was reported was consistent with the reported time, which indicates that this is a minor concern. The use of student participants instead of professionals is a potential issue threatening the generalization of results. This kind of population sampling is sometimes referred to as *convenience sampling* [185]. It is considered controversial due to certain drawbacks [186]. However, studies have shown [187–189] that the differences between the performance of professionals and graduate students are often limited. The experiments were conducted by using teams of 3-5 students, which threatens the generalization of results to a single analyst. Nonetheless, the state-of-the-art [9,105,190] advises sound-boarding the analysis with a team of experts in the industrial setting as well. This is what happens in the medium-to-large companies we collaborate with. Finally, the results of this study may be influenced by the experimental object and in turn, may not be applicable to a system of different complexity or from a different domain.

Possible mistakes might have been made during the assessment of the reports and during the creation of the ground truth. In order to avoid over-loading the participants and make them tired, the supervised experiment was performed in a span of 4 hours. Additional time was given for documenting the identified threats outside the supervised lab. The teams were not monitored after the experiment has ended, however, we have made sure that the final report included only the threats in the lab notes (e.g., by taking snapshots in the lab).

3.7 Related work

McGraw conducted a study including 95 well-known companies [183]. The study analyzes the security practices that are in place in the companies. The BSIMM model does not mention STRIDE per se, rather it highlights the importance of threat analysis. Microsoft has not published evidence of the effectiveness of the STRIDE variants analyzed in this paper. Guidelines, best practices, and shortcomings are discussed, yet there is no evidence about how the two approaches differ in terms of performance [9].

Scandariato et al. [10] have analyzed a previous version of STRIDE-per-element and evaluated the productivity, precision, and recall of the technique in an academic setting. The purpose of their descriptive study was to provide an evidence-based evaluation of the effectiveness of STRIDE. Our study, on the other hand, provides a comparative evaluation (by means of a controlled experiment) of the two latest approaches for STRIDE. Also, our study has a larger number of participants and uses a larger object. We remark that our study has some discrepancies with respect to the observed productivity (4.35 in our study vs. 1.8 threats per hour), precision (0.6 vs. 0.81), and recall (0.62 vs. 0.36). However, a direct comparison is not entirely possible, as the two studies use different versions of STRIDE-per-element (our being the most up-to-date).

A privacy oriented analysis methodology (LINDDUN [11]) has been evaluated with 3 descriptive studies [191]. LINDDUN is inspired by STRIDE and is complementary to it. Both techniques start from a representation of a system, which is described as a DFD. Similarly, the authors assess the productivity, precision (correctness) and recall (completeness) of the technique, as well as its usability.

Labunets et al [192] have performed an empirical comparison of two risk-oriented threat analysis techniques by means of a controlled experiment with students. The aim of the study was to compare the effectiveness and perception of a visual technique with a textual technique. The main findings were that the visual method is more effective for identifying threats than the textual one, while the textual method is slightly more effective for eliciting security requirements.

The work of Karpati, Sindre, Opdahl, and others provide experimental comparisons of several techniques. Opdahl et al. [193] measure the effectiveness, coverage and the perception of the techniques. Karpati et al. [194] present an experimental evaluation of MUC Map diagrams focusing on identification of not only vulnerabilities but also mitigations. Finally, Karpati et al. [195] have experimentally compared MUCs with mal-activity diagrams in terms of efficiency.

Diallo et al. [196] conducted a descriptive comparison of MUCs, attack trees, and Common Criteria [197]. The authors have applied these approaches to the same problem and discuss their observations about the individual technique's strengths and weaknesses. An interesting evaluation of the reusability of threat models (MUC stubs and MUC Maps diagrams, both coupled with attack trees) is presented by Meland et al. [198]. The authors conducted an experiment including seven professional software developers. The study suggests that overall, the productivity is improved by reusing threat models for both techniques.

3.8 Conclusion

This study has presented an empirical comparison of two variants of a popular threat analysis technique. The comparison has been performed in-vitro by means of a controlled experiment with master students. As presented in Section 3.4, this work provides reproducible analysis and observations about the effectiveness of applying both techniques, in terms of productivity, precision and recall. In summary, with the type of population used in this study (non-experts), our study observed better results with the STRIDE-per-element variant. For instance, STRIDE-per-element yielded 1 additional threat per hour in terms of productivity, with no consequences on the average correctness of the results (i.e., same precision). The proponents of STRIDE have claimed that “STRIDE-per-interaction leads to the same number of threats as STRIDE-per-element" [9]. However, in this study, we have observed a statistically significant higher level of completeness in the results returned by the teams using STRIDE-per-element. This is possibly influenced by the tendency of the STRIDE-per-interaction group to create larger DFD models, which might not be necessarily needed. Another explanation is related to the more complex documentation in the case of the Interaction treatment. As security budgets are quite tight in companies, knowing that one variant might be more productive is a useful piece of information.

This work calls for future studies about the effectiveness of the threat analysis variants, especially with more expert analysts. In particular, we are planning a case study in two companies where the local, per-element analysis is compared to a global, end-to-end analysis. Furthermore, it would be beneficial to study the effect on the importance (in terms of risk) of the discovered threats, as well as the quality of the of security requirements that are derived from them.

Chapter 4

Paper C

This chapter is based on

Towards security threats that matter,

written by

K. Tuma, R. Scandariato, M. Widman, C. Sandberg,

published in

Proceedings of the International Workshop on the Security of Industrial Control Systems and Cyber-Physical Systems (CyberICPS 2017), 2017.

Abstract

Architectural threat analysis is a pillar of security by design and is routinely performed in companies. STRIDE is a well-known technique that is predominantly used to this aim. This technique aims towards maximizing completeness of discovered threats and leads to discovering a large number of threats. Many of them are eventually ranked with the lowest importance during the prioritization process, which takes place *after* the threat elicitation. While low-priority threats are often ignored later on, the analyst has spent significant time in eliciting them, which is highly inefficient. Experience in large companies shows that there is a shortage of security experts, which have limited time when analyzing architectural designs. Therefore, there is a need for a more efficient use of the allocated resources. This paper attempts to mitigate the problem by introducing a novel approach consisting of a risk-first, end-to-end asset analysis. Our approach enriches the architectural model used during the threat analysis, with a particular focus on representing security assumptions and constraints about the solution space. This richer set of information is leveraged during the architectural threat analysis in order to apply the necessary abstractions, which result in a lower number of significant threats. We illustrate our approach by applying it on an architecture originating from the automotive industry.

4.1 Introduction

In an ever more complex Cyber-Physical System (CPS) domain, security and trust management are becoming burdensome for many organizations. New software products and frameworks are intended to support functionalities that handle privacy and security of sensitive data. Furthermore, the longevity of a CPS product is typically high (e.g., in the automotive, 25 years), which makes building a secure solution a substantial challenge. Security by design requires addressing security-related issues throughout the entire software development life-cycle. This paper focuses on the early stage of conceptualization of a software system, i.e., the architectural design. Planning for security in early design phases helps designers to steer the product development in a direction where threat mitigations are possible.

In particular, threat analysis is a method that strives towards validating the software architecture and discovering potential design weaknesses. This validation technique is an essential pillar of software security, together with other code-level verification techniques like static analysis and security testing [199]. For instance, Microsoft's STRIDE is a well-known and used technique to perform architectural threat analysis [9]. STRIDE and similar techniques (like LINDDUN [191]) follow the so-called software-centric approach. Such techniques center the analysis around a model of the system that resembles a graph and represents the software components (both computation and storage nodes) and the information exchanged between them (edges). From a syntactical perspective, Data Flow Diagrams (or DFD) are often used to represent such models. According to STRIDE the analysis proceeds by exploring the diagram and discovering several potential threats at each location. On one hand, this way of performing a security assessment has the benefit of being systematic. On the other hand, the analysis is prone to being repetitive and very time consuming. Empirical evidence shows that with only a handful of software components, the analysis can result in the discovery of 50 to 60 security threats, which means a scale factor of 10 [10]. This is known as the 'threat explosion' problem.

The experience of our industrial partners is that, trading systematicity for a timely discovery of most important threats is advantageous. As resources are scarce and time is limited, systematicity is considered an obstacle if it leads to 'wasting' time with security threats that are deemed as not important later on. We also learned that in the early design phase, stakeholders reason about security with a close eye on system assets. Rather than focusing the analysis on the software assets (e.g., software components and data stores), analysts observe information assets and how they move through the system. They do that by analyzing end-to-end usage scenarios which involve a certain information asset and trace the software components that are involved with exchanging, using and storing that particular asset. At each encountered software component, they reason about the potential threats to the asset.

In this paper we synthesize the lessons learned while analyzing the threats in an architecture of a connected vehicle and suggest a novel way of approaching threat analysis. We propose an architectural view for security that is based on DFDs, extended with end-to-end flows representing the information assets in the system. In our enriched model, assets are also annotated with their importance and with security objectives associated with them (e.g., confidentiality). The extended model also includes additional information that is routinely used during the threat analysis process, namely, security assumptions. The extended notation is used to guide the threat analysis and reduce the amount of 'uninteresting' threats that are found. For instance, if an end-to-end flow refers to an information asset that needs to stay confidential, it would be better to focus on disclosure threats (which directly impact confidentiality) rather than on denial-of-service threats (which impact availability instead).

In order to illustrate our approach, the first author applied it on the architecture provided in the context of the HoliSec project [200] on security of connected vehicles. The results of the analysis have been submitted to a domain expert with extensive security background (the third author). Our discussions concluded that our approach indeed led to the identification of valid threats, likely to have the most impact to the organization in reality. Clearly, the obtained results serve as a proof of concept and are a stepping stone for future work.

The remainder of this paper is structured as follows. Section 4.2 describes the running example. Section 4.3 presents the extended notation and the needs of the analyst, Section 4.4 presents the guidelines for abstracting the architectural model (DFD) applied to the running example and Section 4.5 identifies the related work. Finally, Section 4.6 includes a discussion and future work, followed by concluding remarks, presented in Section 4.7.

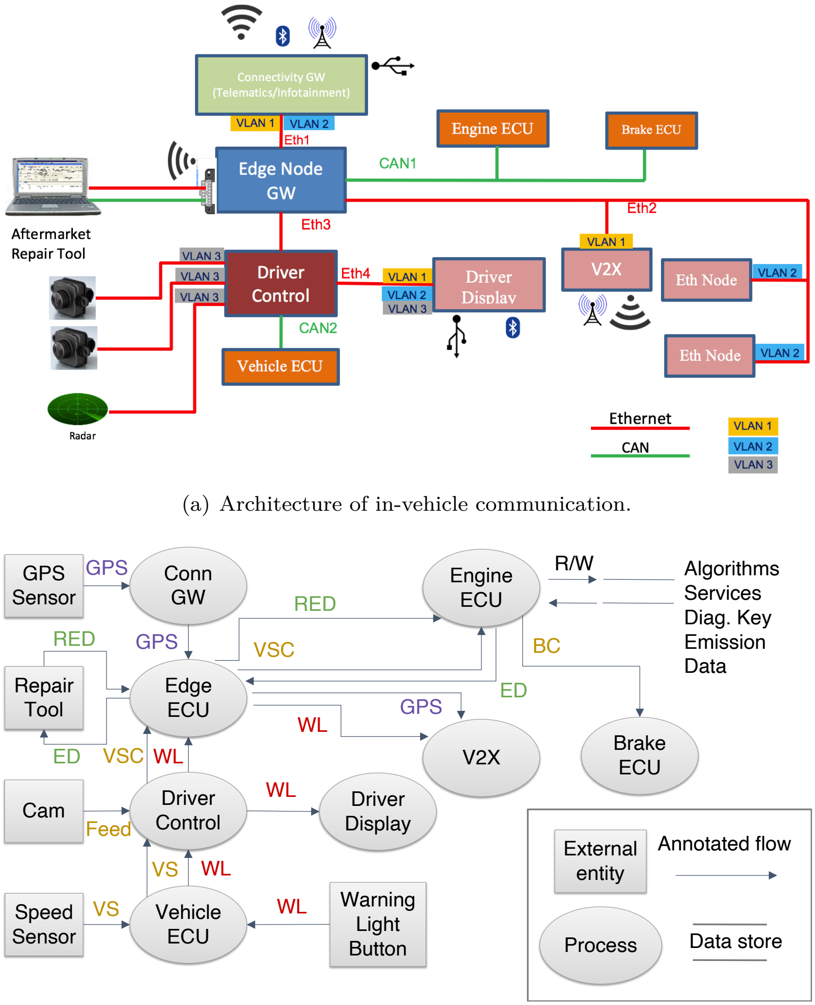
4.2 Running example

In this section we describe the architecture of a vehicle as shown in Figure 4.1. This example is used throughout the paper to illustrate the benefits of our proposed approach. Due to non-disclosure concerns, the example is realistic but does not correspond to an architecture of an existing product on the market.

Modern vehicles are highly complex systems, comprised of hundreds of different components called Electronic Control Units (ECUs), which are responsible for one or more particular features of the vehicle. Individual ECUs are connected to a Controller Area Network (CAN) bus, which is currently the most used in-vehicle communication protocol and also a very common target of attack [201].

As depicted in Figure 4.1(a), the architecture is composed of several ECUs, sensors and actuators exchanging data between each other following a specific communication protocol. The communication between individual components is further specified with a communication matrix (not shown here) of signals, source and receiver components, networks used and type of communication (e.g., broadcast or unicast). For instance, the warning light signals are broadcast on networks CAN 2, Eth 2, Eth 3 and Eth 4. The architecture in Figure 4.1(a) supports a number of functional scenarios, which are described below. Figure 4.1(b) presents a DFD, derived from the architectural information and the assets described in the scenarios. Notice that functional elements represent *processes*, while *data stores* represent the places where information is stored for later retrieval. Everything that is outside of the system (e.g., 3rd party systems) is modeled by means of *external entities*. The arrows represent the exchange of information.

Scenario 1: Set-up diagnostics connection and read emission data. A logging functionality collects information about the vehicle over time, such as



(a) Architecture of in-vehicle communication.

(b) DFD of the architecture.

Figure 4.1: The running example

the emissions data and the GPS position. In order for the data to be collected, the Repair Tool sends the emission data request signal (RED) via the Edge ECU to Engine ECU over the OBD network. The Engine ECU then sends the emission data response signal (ED), including the requested information, back to the external interactor.

Scenario 2: Extended vehicle warning Vehicle to X (V2X) communication allows the exchange of information between the vehicle and the road infrastructure or other vehicles. If the warning light button is active, the vehicle forwards the warning light status (WL) from the Vehicle ECU to the Driver Control. The latter sends it over the Eth 4 network to the Driver Display in order to alert the driver. The warning light status is then sent over the Eth 1 network to the Edge ECU, which, in turn, collects the current GPS position

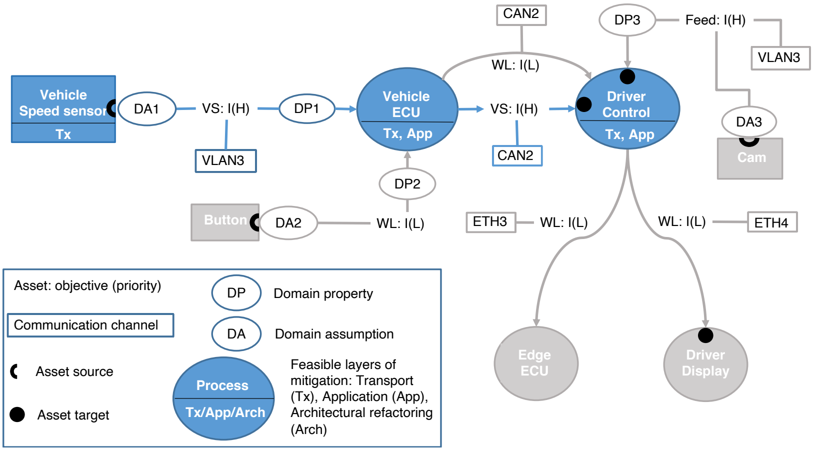


Figure 4.2: The extended DFD notation for an end-to-end, asset-centric flow.

from the Connectivity Gateway. The information (both WL status and GPS position) is sent to the Edge ECU via the Driver Control ECU on networks CAN 2 and Eth 3. Finally, both the GPS location and the warning light status are broadcast via the V2X ECU.

Scenario 3: Using city traffic collision prevention to brake for pedestrians in the trajectory of the vehicle This scenario describes the situation where city traffic collision prevention is used to slow down or stop the vehicle if pedestrians appear in the vehicles' trajectory. The camera sensor sends live video feed to the Driver Control ECU, which analyses it for upcoming obstacles. The Driver Control ECU also receives information about the vehicle speed (VS) from the Vehicle ECU. If a possible collision is detected, the Driver Control ECU generates a vehicle speed change request (VSC) and sends it to the Engine ECU, which orderly sends a brake command (BC) to the Brake ECU.

4.3 An extended DFD notation

Security experts performing threat analysis are aware of the threat explosion problem and try to counter it by making abstractions. For instance, analysts often group seemingly homogeneous elements of the DFD with respect to the type of threats they are subject too. In STRIDE, this technique is called reduction [190]. To make any sort of abstractions possible, candidate elements need to be closely inspected in order to decide whether an abstraction will overall have a negative or a positive outcome. This decision should be an intelligent choice, supported by evidence in the architecture. In order to trace assets through the system, we propose to use asset-centric partial DFDs. Figure 4.2 shows a DFD for the *vehicle speed (VS)* asset. Notice that the DFD is a slice of the overall model presented in Figure 4.1(b).

Security objectives and priorities. Each data flow is marked with the asset that is being transported (also part of the standard DFD notation). We extend the notation by introducing clear markers of where an asset is generated (*asset source*) and where it is consumed (*asset target*). If assets are to be protected, they have to be analyzed from the source all the way to the target architectural element. Additionally, the asset is labeled with one or more *security objectives*: confidentiality (C), integrity (I), availability (A). Security objectives are used to focus the identification of important threats during threat analysis. Often, the analysis spans across several months with brief sessions every week or two. After a few sessions, it is easy to forget about the analysis constraints (e.g., 'focus on confidentiality only') if they are not clearly visible in the diagram. The asset is also ear-marked with a *priority* label that signifies the importance of the objective. We use the values of low (L), medium (M) and high (H). These do not express the importance of the asset as such, but rather the impact of a compromised asset objective. This information helps calibrate the depth of the analysis to come.

Security assumptions and properties (at flows). During threat analysis, assumptions are made in order to assess whether threats are feasible in the underlying architecture. For instance, the integrity of VS can be compromised if it is transported in clear and if the attacker has access to the transport medium. Together with the domain expert, the analyst may choose to make an assumption about an existing security mechanism in place that protects VS against tampering threats. When making assumptions about the system, the analysts need to be very careful not to make optimistic assumptions, which can lead to overlooked threats. A false assumption about the security of a GPS sensor can, for example, result in overlooking spoofing threats. On the other hand, not making any assumptions can make the analysis highly inefficient and result in the elicitation of irrelevant threats. Today, assumptions are sometimes still documented separately in an informal way. Considering that they are easily forgettable, they must be made visible in the model, right where they are needed.

In this paper, we distinguish between domain assumptions and domain properties, as described by Van Lamsweerde [202]. Domain properties are used to describe non disputable facts about the domain, whereas domain assumptions are statements about the domain that may or may not hold. For instance, "It is infeasible to encrypt the CAN bus" is a domain property. This is something that we can not change about the domain. On the other hand, "The CAN bus is not accessible to the attacker" is a typical domain assumption.

In Figure 4.2 there is one assumption and one property on top of the flow going from vehicle speed sensor to Vehicle ECU. The domain assumption DA1 is the following: "The vehicle speed sensor is a Commercial-Off-The-Shelf product and is working securely." The domain property DP1 placed on the same flow is the following: "There is a feasible mitigation solution on the transport layer for the flow between the vehicle speed sensor and the Vehicle ECU." The assumption DA1 and property DP1 are later on used to argue about the security of assets on the flow, which is discussed when abstracting the diagram. The limited layers of mitigation solutions are annotated within processes and external entities with capital letters.

Finally, there is one property that is so important (especially in embedded systems) that it gets its own annotation. We refer to the representation of the *communication channel* for each data flow. The communication channel explicitly shows which network the data flow and the corresponding asset belong to. A regular DFD notation does not include the topological behavior gathered in the communication matrix. Keeping that information visible is important, because most domain properties and assumptions that have to be made are about network and protocol capabilities.

Forward assumptions (at processes). Processes are ear-marked with this annotation, which is important in the perspective of simplifying the analysis process, as described in Section 4.4. In essence, we suggest that it is useful to explore the space of possible solutions which are realistic to implement in terms of mitigation mechanisms. For example, some ECUs include a Hardware Security Model (HSM), which provides transport layer encryption. This exploration needs to be done before the threat analysis starts. As the same threats start to appear more often (e.g., the assessment of the integrity of the VS is generating a lot of tampering threats along the asset flow), it is more efficient to turn a forward assumption into a domain property (i.e., mandate the adoption of a certain security mechanism, like turning on the encryption) and stop bothering about that asset all together. This kind of of backtracking in the analysis process is supported by the extended notation.

Our work differentiates between threat mitigation solutions on different layers of abstraction: transport (Tx), application (App) and architectural (Arch) layer. For instance, on the transport layer, symmetric cryptography may be used to establish a secure communication protocol between one of the sensors connected to the vehicle and the receiving ECU. On the other hand, an application layer mitigation solution would include an application layer firewall that inspects packets traveling to and from the Repair Tool in a remote diagnostic scenario. On the architectural layer, security mitigation techniques include architectural re-factoring, where possible design decisions are required to modify the system architecture. Note that middle-ware solutions, such as message queues, are also grouped as architectural layer mitigation techniques.

In addition, while the focus of abstraction is on a single end-to-end flow, elements indirectly involved in the end-to-end flow are still represented in the diagram. The reason for including additional elements is because abstractions directly effect neighbor elements. Consider what happens if two processes are folded. One part of the end-to-end flow gets successfully abstracted. However, there might have been other flows between that were unaccounted for. Neglecting the neighboring elements can increase the risk of missing important threats during the analysis. We use a different color for such elements (gray) to stress this point.

In summary, there are three important actors that participate in the process of obtaining the extended DFD. Figure 4.3 shows how a domain and a security expert work together with a business expert to gather the necessary information. The eDFD is built after having analyzed the problem and solution space. First, the domain expert defines an architectural model including security objectives, purpose and priority of main assets in the system. To that end, the business expert contributes with determining the priorities of assets, while the security expert helps define the objectives. The architectural model is comprised of a structural, behavioral and a topological view. This architecture is used as input to create a DFD. After the DFD defined, all three actors contribute to asset analysis, where they reason around the *problem space*. Asset analysis is performed in an iterative manner. Each asset is first identified, then the source and target components of the asset are located. Although the domain

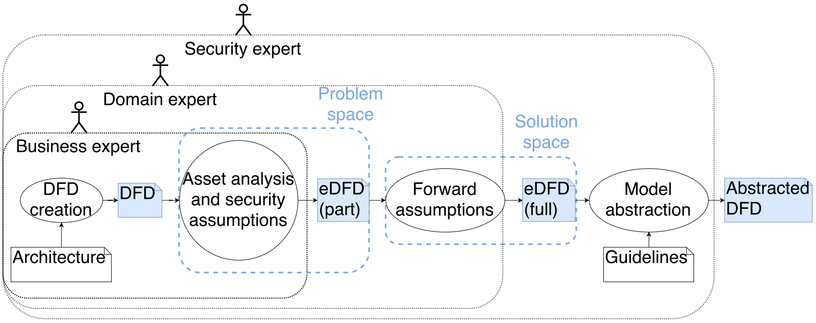


Figure 4.3: Roles and responsibilities for extending the DFD using the proposed approach.

and security experts are mostly active in this phase, the business expert contributes to the discussion from a financial perspective. The asset is then traced in the DFD, where all components affiliated with the asset are marked accordingly. Asset analysis activities are repeated for all assets identified within user scenarios (behavioral view). At this point, the partially extended DFD includes security objectives, priorities and security assumptions at flows, put forward in Section 4.3 In order to complete the eDFD, forward assumptions about processes need to be made. When making forward assumptions about security mitigations, the actors discuss the so called *solution space*. To wit, domain and security experts limit possible mitigations in accordance with domain specific constraints and possibilities. Lastly, using the complete eDFD, the security expert makes use of the guidelines to abstract the DFD. Section 4.4 discusses the model abstraction activity and how it counters the threat explosion. Threat analysis begins after the DFD is abstracted.

4.4 Handling the threat explosion

In this section we discuss how our approach tackles the previously mentioned problem of the ‘threat explosion’, i.e., when too many (often irrelevant) threats are found when STRIDE is applied to a medium-to-large DFD. With the support of the running example, we illustrate how the abstraction is performed before the threat analysis and how the effort reduction is supported during the analysis.

4.4.1 Abstraction before threat analysis

The first approach towards solving the problem is to reduce the number of DFD elements (before the analysis starts) by means of heuristics. In such a manner the model is simplified and consequently, the analysis produces less unimportant threats. We have defined two initial guidelines for flow bundling and process folding. To this aim, the extended notation introduced previously plays an important role. The guidelines were obtained through iterative sound-boarding with industrial partners while working on the architecture from Figure 4.1. The reader should note that the guidelines are not meant to be a complete set of criteria, but rather the result of our initial observations. Having said that, we defined two different sets of guidelines for components with either critical or non-critical assets. For critical components, a more strict set of criteria is used, while for non-critical components the criteria are somewhat loosened. In this way, the abstraction is done in accordance with the “heat” of the system (obtained from the asset analysis) in a each region.

Bundling data flows. We consider two or more data flows (arrows in a DFD) between two processes, a process and an external entity or a process and a data store. When bundling data flows, the highest security objective of assets dictates the level of criticality. For non-critical assets, the corresponding data flows must be associated to the same communication channel (e.g., CAN 1) for the bundling to be possible. For instance, in Figure 4.1(b) the flows between the Repair Tool and Edge ECU include assets that are not critical and they are broadcast over the same network. Therefore, they can be bundled. After bundling, the resulting data flow is annotated with a new name and the union of security objectives from both bundled data flows. If the flows contain the same security objective, the highest priority is included in the union.

If any of the data flows considered for bundling contains a high security objective, the area is considered critical and additional criteria need to be met. We must look at the end-to-end flows (as in Figure 4.2) for the involved assets. These end-to-end flows must have either the same source, target or both. Otherwise, if they have different source and target, the unaligned parts of the end-to-end flows should not be critical. For instance, in Figure 4.2 the data flow VS between Vehicle ECU and Driver Control ECU is marked with a high-priority integrity objective, while WL has a low-priority integrity objective. Therefore, this area in the architecture is considered critical. Both VS and WL data flows are broadcast on the same communication channel. The diagram shows that the VS end-to-end flows goes from the Speed Sensor to the Driver Control. The WL signal end-to-end flows goes from the Warning Light Button (pressed by the driver) to the Driver Display. Therefore, the assets VS and WL do not have the same source or target. Rather, they only align between the Vehicle ECU and Driver Control. However, the criticality of the non aligned parts is low. In conclusion, the data flows can be bundled according to the more restrictive criteria.

Process folding. Candidate elements for process folding are two adjacent processes. All flows between the candidate processes are considered when determining the criticality of the region.

If the flows do not transport high-priority assets, the region is considered as non-critical. Non-critical processes may be folded if (a) they are not near to a trust boundary and (b) there is a mitigation in place (security assumption) that ensures that at least one of the objectives for the surrounding flows is covered. If a trust boundary is next to the considered region, the processes are likely part of the attack surface and it is considered too risky to bundle them, as relevant threats might be overlooked. In the example of Figure 4.2, this guideline applies to the region containing the Driver Control and Driver Display: there are no high priority assets broadcast on ETH 4 and the processes are not near to a trust boundary. Hence, the two processes can be folded.

If any flow between the candidate processes contains a high security objective, the criteria are more restrictive. In addition to the above-mentioned conditions, the processes (c) need to be “mounted” on the same communication

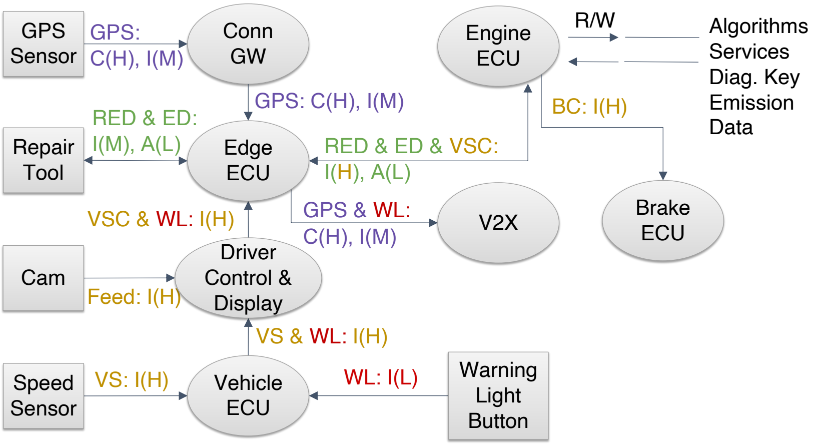


Figure 4.4: The abstracted DFD after applying the guidelines.

channels and (d) there must be mitigations in place ensuring that *all* security objectives over the flows are covered. Unfortunately, no such fold was possible in the running example.

Figure 4.4 presents the results of the abstraction obtained by applying the guidelines described in this section. Overall, the difference between the original DFD (in Figure 4.1(b)) and the abstracted DFD (in Figure 4.4) lies in the number of flows and processes. The abstracted DFD has 1 process less (Driver Display) and 7 flows less compared to the initial DFD. The simplifications to the model result in a reduced number of identified unimportant threats, as further discussed in Section 4.4.3.

4.4.2 Effort reduction during threat analysis

Abstracting the architectural model before threat analysis can only take the analyst so far. It is not only the number of DFD elements that makes threat analysis time consuming, but also the type and amount of threat patterns that must be considered for each element. However, the additional information about the assets and the forward assumptions may also be used *during* the analysis to guide the analyst towards the important threats, while omitting the not important ones.

For instance, in Figure 4.4, the RED and ED flows have been bundled as a result of the abstraction step. The assets on this bundled flow are ear-marked with the integrity objective (medium priority) and the availability objective (low priority). Therefore, the analyst can focus on the tampering and denial of service threats, and ignore the information disclosure threats.[1] As the priorities are not high, the analyst also knows that it is not necessary to dig too deep in the analysis process of this flow. In this respect, the analyst can bring these risk considerations into the process of threat identification and leverage them to reduce the effort spent analyzing this “cool” spot of the system. As a

[1] According to STRIDE, a data flow is subject to three types of threats: tampering (T), information disclosure (I), and denial of service (D).

result of the analysis, we found two important threats on that particular flow: physically damaging the OBD port and tampering with the ED signal before it is displayed on the Repair Tool.

Another means of reducing the effort spent eliciting the threats is to use the security assumptions during the threat analysis. For instance, let us suppose we are analyzing the vehicle speed (VS) asset-centric flow described in Figure 4.2. The asset is ear-marked with a high-priority integrity objective and, hence, the analyst should carefully consider the potential tampering threats. However, Because of the domain assumption DA1 ("The speed sensor is working securely") and the domain property DP1 ("Transport-layer security is used between the ECUs"), the tampering threats can be discarded altogether as they are non interesting.

4.4.3 Effect of abstraction

The resulting abstracted DFD has been analyzed (by the first author) using the off-the-shelf threat catalogs of STRIDE. This led to the identification of 15 threats, which are not presented here due to space constraints. To appreciate the efficiency of our approach, we remark that a previous study [10] has shown that according to the traditional STRIDE approach, the number of identified threats from a DFD this size should have been around 100. The 15 identified threats have been reviewed by a security expert (the third author) who routinely performs the threat analysis of automotive systems. The validity and relevance of the identified threats have been confirmed by the expert. Further the expert confirmed that no relevant threats had gone unnoticed.

Our approach resulted in finding less threats because of two reasons. Processes and flows are the elements that are the main cause of threat explosion in a DFD. First, together they make up a large number of architectural elements. Second, they are prone to many types of attacks and, hence, have to be analyzed for several threat categories. Out of a total of six threat categories, STRIDE mandates the consideration of three categories for flows and of all six categories for processes. It is apparent that reducing the number of processes and flows in the DFD can help govern the threat explosion problem. Second, a further reduction of effort is due to a more focused analysis, as explained in Section 4.4.2.

4.5 Related work

Significant work has been done in the area of threat analysis and risk assessment (TARA) methods in the automotive domain. Macher et al. [14] recently performed a review of TARA methods in the automotive context. In their main findings, the authors identify most applicable TARA methods for early phase analysis, which closely relate to our approach. The EVITA [203] method is an adaptation of the ISO 26262 HAZOP analysis for security engineering. The method considers potential threats for particular features from the functional perspective by developing attack trees and eventually discovering worse case scenarios. Even though the method employs leveled qualitative risk assessment, no effort is mentioned regarding attack tree minimization. HEAVENS security model [204] analyzes threats based on the STRIDE threat modeling approach and ranks them by assessing the risk with determining the threat, the impact and finally the security level. In contrast to our approach, HEAVENS model is oriented towards ranking the threats only after identifying them. SAHARA [164] method combines the automotive HARA safety analysis with the STRIDE approach to discover impacts of security threats in the safety analysis. The focus of SAHARA lies in understanding the relationship between security threats and safety implications, whereas our work focuses on security aspects only. A security analysis of several applications within a Connected Vehicle Reference Architecture (CVRIA) has been performed by ITERIS and is available online [205]. The published documents include a CIA asset analysis of the V2X communications, while our work focuses on the in-vehicle communications.

Beyond the domain of automotive software, other asset- or software-centric threat modeling approaches are relevant to our work. STRIDE [9] is a popular threat modeling approach, which is based on DFDs. The methodology is comprised of 7 steps: define users and realistic use scenarios, gather assumptions, construct a DFD diagram of the system, map STRIDE to DFD element types, refine threats, document the threats, assign priority via risk analysis (to counter threat explosion problem) and select mitigations associated to threats. In STRIDE, threat prioritization according to risk value is done after the threat elicitation. Additionally, although STRIDE suggests to start by gathering the security assumptions, no explicit guidance is provided on how to represent and use them in the threat analysis process. Similar to STRIDE, TRIKE [148] is a software-centric methodology. TRIKE includes the identification of assets and actors and offers tool support for attack tree and graph generation. CORAS [25] is an asset-centric methodology for risk analysis consists of a language, a tool and a method. The methodology employs CORAS threat diagrams that describe the threats, vulnerabilities, scenarios and incidents of relevance for the risk in question. Similarly to the aforementioned CORAS methodology, PASTA [29] is an asset-centric, risk-based threat modeling methodology. In PASTA threats are analyzed with the help of use cases and Data Flow Diagrams (DFDs). Further steps are taken for detailed analysis, namely the use of attack trees and abuse cases.

Looking towards recent initiatives to automate threat modeling, our approach relates to the work on extracting threats from DFDs by J. Berger et al. [45]. Similarly to our work, the authors introduce additional semantics, including the topological behavior. Furthermore, they also develop a set of guidelines, which are used to build a threat model of the architecture. However, these rules are used to discover only cataloged threats and do not aim to handle threat explosion. Perhaps more importantly, our approach differs by analyzing end-to-end assets which, in turn, drives the model abstraction and threat reduction.

Interesting work has been done by Rauter et al. [206] in developing a metric that quantifies software components by their ability to access assets. By doing so, the authors are also able to identify critical areas in the software architecture and consider those for a detailed threat analysis. However, they do not discuss the specifics of how threat analysis techniques benefit from their architectural risk assessment method. Our work also relates to the automated software architecture risk analysis studied by Almorsy et al. [44]. The introduced approach is accompanied by a tool and explores security risk analysis by means of formalized signatures of security scenarios and metrics. Similarly, the authors develop a risk-centric architectural analysis approach. However, their work focuses on operationalizing attacks and security metrics to assess the risk level, instead of aiming towards systematically identifying important threats. In addition, the formalized signatures do not seem to consider end-to-end flows of assets.

We also identify several related approaches that could be grouped as attack-centric threat analysis techniques. In these approaches, an explicit model of the attacker is introduced and the analysis is performed from the perspective of an attacker. Examples of such techniques are anti-goals [139], misuse cases [35], misuse case maps [145], abuse cases [137], abuse frames [38], and attack trees [39, 207].

4.6 Discussion and limitations

The process of creating the enriched model described in Section 4.3 results in a deeper understanding of the system before threat analysis activities take place. Not only does this contribute towards a common security awareness, but it also enables the identification of realistic threats in the system. As previously mentioned, one of the drawbacks of STRIDE is that the analysis of DFD elements is performed in isolation. In contrast, attack strategies target an asset and often affect a combination of elements. Our approach implicitly considers all the model elements that are related to an asset, which contributes towards detecting attack strategies earlier on in the software development life-cycle. Most importantly, our approach is a step further towards optimizing the effort spent on analyzing threats. This aspect is of primary importance in the industry (especially for complex systems, like CPS) and is often overlooked in the related work. Even though the guidelines are only initial observations, they are synthesized in a domain-independent way, where the main role is played by the semantics of end-to-end flows and not the domain-specific content. Therefore, there is potential for generalizing the guidelines to domains outside the scope of cyber-physical systems. In particular, domains where the software architecture is comprised of different networks and communication protocols may benefit from our approach (e.g., Microservice architecture). However, more investigations have to be made in order to confirm this claim.

Our approach relies on the correctness of the domain assumptions and the truthful representation of the domain properties. This means that the presence of a domain expert is mandatory. Another draw back is that the approach assumes that there are in fact non-critical areas in the architecture. If all the candidate model elements for abstraction have high-priority security objectives, the abstraction may result fewer simplifications, if any at all. Another problem might arise once the amount of end-to-end flows increases. The guidelines do not consider what happens when new scenarios are added. New scenarios might bring along different assets that travel through the same communication channels. As a result, successful abstractions have to be reconsidered. Some of these problems could potentially be alleviated by a tool. In terms of increasing the productivity, the application of guidelines for abstraction can be (semi)automated. Such support may take advantage of our approach, while enabling experts to freely move between abstractions during the analysis. Working towards the automation of threat analysis also caters to security related activities later in the product life-cycle. Notably, after implementation, the planned architecture comes seldom into question again. However, the implemented architecture often differs from the planned architecture too much. Checking for compliance is therefore a complex and important task. As part of future work, we plan to explore ways to synchronize the extended DFDs and the implemented architecture. Furthermore, we acknowledge that we have validated the approach only by illustrating its potential on one simplified architecture. In future work, we will systematically evaluate the benefits of our approach in a series of comparative studies involving both students and industrial experts.

4.7 Conclusion

In this work, we presented a novel approach for a risk-first security analysis of design artifacts. Without departing too much from well-known techniques like STRIDE, our approach is focused on an end-to-end asset analysis and accommodates forward reasoning on the constraints imposed by the solution space. Our main contributions are (i) a notation to represent end-to-end asset flows and to enrich the analysis models with important security assumptions, and (ii) a set of guidelines for traversing the model and making suitable abstractions. The contributions aim at mitigating the problem of threat explosion, commonly present in STRIDE and other techniques. Additionally, the extended notation supports a more appropriate representation of communication channels, which is valued in the domain of Cyber-Physical Systems. We illustrated our approach by applying it on a simplified system provided by industrial partners in the automotive domain. Preliminary results show that the analysis method indeed yields a reduced number of low priority threats. In future work, we plan to extend the validation of our approach and explore the opportunities for threat analysis automation.

Chapter 5

Paper D

This chapter is based on
Finding Security Threats That Matter: Two Industrial Case Studies,

written by
K. Tuma, C. Sandberg, U. Thorsson, M. Widman, T. Herpel, and R. Scandariato,

submitted to
Journal of Systems and Software (JSS), 2020.

Abstract

In the past decade, speed has become an essential trait of software development (e.g., agile, continuous integration, DevOps, etc.) and any up-front inefficiency is considered unaffordable time waster. Such a fast pace causes challenges for architectural threat analysis. Leading techniques for threat analysis, like STRIDE, have the advantage of being thorough and systematic. However, they are not equipped to discern between important and less critical threats, *while* the threats are being discovered. Consequently, many threats are discarded at a later time, when their risk value is assessed. An alternative technique, called eSTRIDE, promises to remove these inefficiencies by focusing the analysis on the critical parts of the architecture. Yet, no empirical evidence exists about the actual effect of trading off a bit of systematicity, which is a benefit of STRIDE, for a more focused attention on high-priority threats, which is the goal of eSTRIDE. This paper contributes with an empirical study comparing and contrasting these two approaches in the context of two industrial case studies. We have found that the two approaches are similar in terms of overall productivity. However, participants using eSTRIDE found twice as many high-priority threats. In addition, we found that, in the industrial setting, security expertise may be traded for a faster-paced and less precise threat analysis.

5.1 Introduction

Security-by-design techniques aim to avoid security pitfalls in software systems early on, starting from the design phase, when the major development effort is yet to come [5, 190]. The intent is to cut the maintenance cost induced by fixing security design flaws when it is too late. In this context, architectural threat analysis is a common activity performed by experts (often manually) to analyze the high-level design of a system for potential security issues related to its assets of interest [208]. These threat analysis techniques are routinely used in application domains where upfront design is still dominant, for instance in safety-critical systems like automotive. For instance, Microsoft's STRIDE is a well-known threat analysis technique that is also used in the automotive domain [9, 209]. This technique has the tendency to lead to the discovery of a high volume of potential threats [10, 210]. After the discovery phase, threats are ranked according to their risk value, which is a combination of impact and likelihood. As a result, many threats are later-on discarded due to their low risk value, even though significant time went into their discovery. This is a clear element of inefficiency, which is common to these family of so-called *risk-last* approaches, where risk is considered only *after* the threats have been found.

In the past decade, speed has become an essential trait of software development (e.g., agile, continuous integration, DevOps, etc.) and any up-front inefficiency is considered unaffordable time waster. Consequently, such a fast pace causes challenges for threat analysis [33]. Further, security expertise is scarce in organizations and the time available from experts needs to be used in an optimal way. Driven by these observations, in prior work we have defined eSTRIDE, a *risk-first* threat analysis approach [182]. The core idea is to enrich the analyzed model with risk-related information, so that the analysis activity is more focused on the critical parts of the architecture. This would lead to the early discovery of high-priority security threats, hence by-passing threat prioritization all together. This seems promising for organizations where development speed is key to surpassing the competition. Yet, no empirical evidence exists about the actual effect of trading off a bit of systematicity (a benefit of STRIDE) for a more focused attention on high-priority threats (the goal of eSTRIDE). Could important threats go unnoticed? Are high-priority threats being discovered faster? These and similar questions beg for an answer.

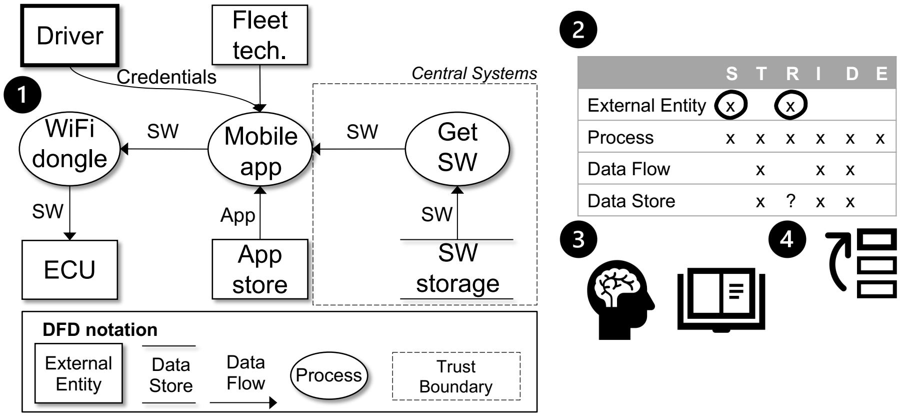
The purpose of this study is to gather empirical evidence about the similarities and differences between a risk-last (STRIDE) and risk-first (eSTRIDE) threat analysis technique (see Section 5.2) in an industrial setting. To this aim, we conduct two case studies (see Section 5.3) with industrial participants (15 in total) from two automotive organizations located in different countries. Within each organization, we observe and compare two teams analyzing the same system, where each team uses one of the two mentioned techniques.

The contributions of this work are three-fold. First, we try to understand whether a risk-first technique leads to the discovery of more high-priority threats and sooner, hence optimizing the time and effort spent in performing a threat analysis. Second, we carry out a qualitative and quantitative comparison of how the two techniques are performed, e.g., by looking at what activities are more prevalent. This allows to identify key insights into the way teams work when they use the two techniques and, accordingly, gauge the potential for

Table 5.1: Activities of STRIDE and eSTRIDE

Step	Activity	STRIDE	eSTRIDE
Building diagram	Scope discussion	✓	✓
	Drawing the DFD	✓	✓
	Model abstraction/refinement	✓	✓
	Asset analysis		✓
	Extending the DFD		✓
Analyzing diagram	Diagram exploration	Element by element	Scenario-based
	Threat types considered	Mapping table	Pruned mapping table
	Attack scenario development	✓	✓
	Threat feasibility discussion	✓	✓
	Threat reduction	✓	✓
	Threat prioritization	✓	

Figure 5.1: The main steps of performing STRIDE



	S	T	R	I	D	E
External Entity	x		x			
Process	x	x	x	x	x	x
Data Flow		x		x	x	
Data Store		x	?	x	x	

optimization. Third, we investigate the effect of security expertise on the use of both techniques. As security expertise is a scarce commodity, it is interesting to study whether less skilled teams could produce acceptable results with any of the two techniques.

The results of this study (Section 5.4) show no differences in productivity and timeliness of discovering high-priority security threats. But, we find differences in analysis execution. Specifically, participants using the risk-first technique found *twice as many* high-priority threats, developed detailed attack scenarios, and discussed threat feasibility in greater detail. In comparison, participants using the risk-last technique found more medium and low-priority threats. In addition, we find that security expertise may be traded for a faster-paced and less precise threat analysis. The results are further discussed in Section 5.5 and their validity is reproached in Section 5.7. We contextualize our results with respect to the related literature in Section 5.6, and present our conclusions in Section 5.8.

5.2 The compared techniques

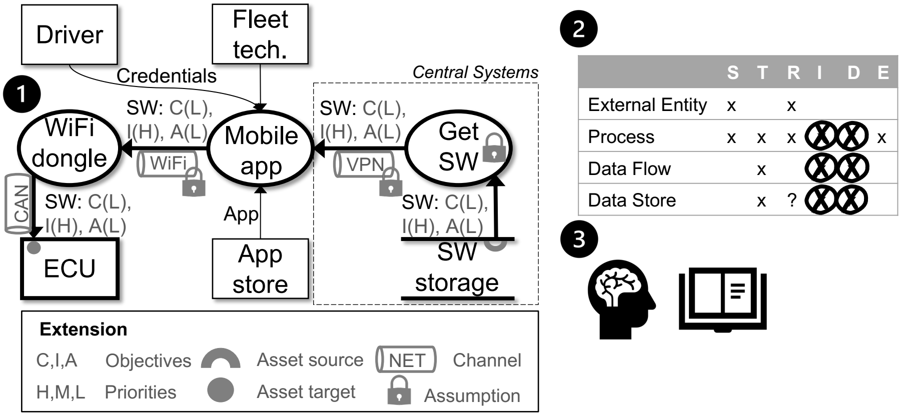
Stride – STRIDE is a family of techniques developed by Microsoft to help identify threats (e.g., potential attack scenarios) that software systems are exposed to, especially because of design-level flaws. The name itself is an acronym that stands for the threat categories of Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege. For the definition of threat categories, we refer the reader to the documentation of STRIDE [9]. In particular, this study considers the 'STRIDE per element' variant (hereafter Stride).

Table 5.1 summarizes the activities performed during the analysis with Stride. It is a model-based technique which starts with the creation of a graphical representation of the software system under analysis, namely as a Data Flow Diagram (DFD). Figure 5.1 shows a simplified DFD and how it is used during Stride. We depict an industrial case from the automotive context (also under analysis in this work) for a firmware update of an Electronic control unit (ECU) in a truck. On a high-level, the driver (or technician) connects their mobile device to a WiFi dongle in their vehicle, logs into the Mobile app (installed on their device), gets the software from a remote software repository and installs it on the ECU of their vehicle. Creating the DFD typically involves discussing the scope of the analysis (i.e., defining the breadth of analysis), drawing the diagram (e.g., on a board), abstracting or refining the model (hence defining the granularity of the analysis), and so on. A DFD represents how information moves across a software system and consists of processes (active entities, like *'Get SW'* in Figure 5.1), data flows (exchanged info), external entities (e.g., *'Driver'*), data stores (e.g., *'SW storage'*), and, optionally, trust boundaries (e.g., *'Central Systems'*).

The second phase in Stride is the systematic exploration of the DFD to identify the threats. It involves several activities (see Table 5.1) which can be roughly grouped into four steps (steps 1-4 in Figure 5.1).

First, the analysts explore the diagram one element at the time (step 1). For each element type, STRIDE suggests to look into specific threat categories, which is given by the so called threat-to-element mapping table (step 2 in Figure 5.1). For instance, for external entities (e.g., *'Driver'*) the analysts are suggested to only look for spoofing and repudiation threats. Next, the analysts engage in a brainstorm to develop concrete attack scenarios, during which they can use exemplary threats (of each category) for inspiration (step 3). Each discovered threat must be discussed with respect to its feasibility in order to determine its relevance. The relevant treats and discovered attack scenarios are immediately documented. Sometimes, the same threats can be present at multiple locations in the diagram (e.g., an information disclosure threat on all data flows that are not encrypted). In such situations, analysts can apply the *threat reduction* technique and continue with the diagram exploration. This technique essentially allows copying the identified attack scenarios to all the said locations in the report. After the diagram has been explored the threats are prioritized according to their risk value (step 4). As the risk assessment happens *at the end*, Stride is a 'risk-last' threat analysis technique. An unfortunate consequence of this approach is that the effort spent identifying low-risk threats could be a potential time-waster and a source of lower productivity.

Figure 5.2: The main steps of performing eStride



	S	T	R	I	D	E
External Entity	x		x			
Process	x	x	x	⊗	⊗	x
Data Flow		x		⊗	⊗	
Data Store		x	?	⊗	⊗	

eStride – The second technique we study is the Extended STRIDE (hereafter eStride, which is an example of a 'risk-first' technique [182]. As shown in the right-hand side of Table 5.1, the analysis technique is similar to Stride, with a few substantial differences. In eStride, an asset analysis is performed *at the beginning*, during the model building phase. This includes the identification of assets, their security objectives (e.g., confidentiality should be preserved) and their importance (e.g., high confidentiality). This security relevant information is inserted in an extended diagram, or eDFD. Figure 5.2 shows an eDFD (of the same system as shown before) and how it is used. For brevity, we have omitted the accountability objective (effected by the threat of repudiation) from the figure. The eDFD also carries information about the source and sink of each asset (i.e., an information scenario), the type of communication channels in the system, and domain assumptions that are relevant to the security analysis (e.g., existing security solutions). To save time, this additional information is only added to elements involved in the transfer (or storage) of high-priority assets (as observed in Figure 5.2).

The second phase of eStride is a guided exploration of the eDFD (steps 1-3 in Figure 5.2). Instead of visiting each element in the graph (as is done in Stride), the analysts consider end-to-end information scenarios (i.e., paths from source to sink, such as the path of the *'SW'* from the *'SW Storage'* all the way to the *'ECU'*) of assets with at least one high-priority objective (step 1).

Further, in step 2 the mapping table used in Stride is pruned before threats are identified. In particular, the categories of threats that do not relate to the high-priority objectives are discarded. For instance, in Figure 5.2 the only security objective with a high value is integrity, thus information disclosure (which directly effects confidentiality) and denial of service (effecting availability) threats can be skipped. Spoofing and elevation of privilege threats can not be skipped so easily, as they can potentially threaten all the security objectives. However, the analysts may make domain assumptions about existing security solutions. For example, they may assume a mutually authenticated and end-to-end encrypted channel (*'VPN'* in Figure 5.2) between the components of the *'Central Systems'* and the *'Mobile app'*. Therefore, spoofing the process *'Get SW'* to distribute malicious updates could be skipped in the analysis. In this case, spoofing the *'ECU'* or the *'WiFi dongle'* must still be considered.

Similar to Stride, attack scenarios are built and discussed for feasibility with respect to the domain, and threat reduction can be applied (step 3). However, using eStride the analysts are equipped with more information to discuss the feasibility (e.g., existing security solutions). The expected benefit is a reduced effort due to bypassing threat prioritization (step 4 in Stride) and investigating fewer threats.

5.3 Design of the Study

We conduct two case studies where we compare Stride to the eStride. In what follows, we present the research questions, industrial cases used in this study, and the participants. We also describe the task performed by the participants, the on-site workshops, and the data collection methods.

5.3.1 Research questions

This study answers research questions regarding the differences in the analysis outcomes (RQ1, RQ2, RQ4) and procedure (RQ3) for the studied techniques.
RQ1. *What are the differences between a risk-last and a risk-first analysis technique in terms of productivity?*

Risk-last threat analysis prioritizes threats at the end of the analysis procedure. In contrast, risk-first analysis aims to by-pass threat prioritization by analyzing the high risks first, at the cost of a more extensive modeling phase. The purpose of the first research question is to understand whether the extra up-front effort has an impact on the productivity, measured as the amount of correctly identified threats per time unit.
RQ2. *What are the differences between a risk-last and a risk-first analysis technique with respect to the timeliness and amount of discovered high-priority threats?*

In realistic circumstances, threat analysis sessions are pressed for time. Achieving complete coverage with a manual analysis is challenging in this context. Therefore, threats are often overlooked [10, 210]. It seems reasonable to 'knowingly' overlook less-important threats as compared to high-priority threats. The purpose of the second research question is to investigate whether risk-first analysis produces important threats faster (and in a larger quantity) when compared to the risk-last analysis technique.
RQ3. *What are the differences between a risk-last and a risk-first analysis technique with respect to both the timeliness and the amount of activities as well as activity patterns?*

Apart from the activities in Table 5.1, important events and support activities take place during a threat analysis session. For instance, updating the diagram, or making an assumption. Support activities include pointing at the board, taking a break, documenting, referring to case documentation, etc. Due to the repetitive nature of manual threat analysis, these activities tend to re-occur. We are interested to investigate which activities appear more often or sooner, and how that differs for the two techniques. In addition, we observe combinations of activities, or activity patterns to understand which technique better facilitates constructive thinking. Therefore, the purpose of the third research question is to investigate the differences in the 'way of working' for the studied techniques.

RQ4. *What is the effect of the security expertise of the participants on the productivity and correctness of a risk-first and risk-last analysis technique?*

Previous studies paint a picture of the current security activities, skills [211] and threat analysis practices [15, 33] in agile organizations. In [15] three (out of four) interviewed companies revealed that developers are already involved in threat analysis. Further, in two organizations [15] they are even responsible for identifying threats, but still need input (i.e., from security managers, or consultants) when it comes to asset and risk analysis. Considering the fairly acceptable performance measured for STRIDE in the academic setting [10, 210], we are interested to study how security expertise relates to the performance of the two techniques.

5.3.2 Industrial partners

Org A The study was first executed within a multinational automotive organization with over 100 000 employees worldwide. The core activity of Org A is the production, distribution and sale of trucks, buses, and construction equipment. This multinational has established security practices in the development life-cycle and performs STRIDE-like threat analysis on a daily basis. From a security standpoint, participants from Org A can be regarded as experts.

Org B The study was replicated within a younger (and smaller in terms of number of employees) organization based in a different country. This organization is specializing in the development and testing of autonomous driving solutions. In comparison, development teams in Org B are much smaller and more cross functional. In addition, threat analysis is not performed routinely within Org B. From a security standpoint, participants from Org B can be regarded as novices.

5.3.3 Industrial cases

We ask each company to identify a software system they want to analyze from a security standpoint. Each of our industrial collaborators put together a document describing their system, including textual specifications and technical diagrams. We provided feedback on the documentation in order to guarantee that it was clear and contained enough information. In what follows we briefly describe the cases but omit details due to confidentiality concerns. Both cases are from the automotive domain and deal with (safety critical) embedded software.

Org A– ECU update. The analyzed industrial case is about the firmware update of an Electronic Control Unit (ECU) in a truck. The update can be performed by an authorized party (e.g., the driver who wants to change the speed limiter when crossing countries) via a mobile app and without visiting the workshop. In the analyzed scenario, the driver (or technician) connects their mobile device to the WiFi dongle of the vehicle to update the ECUs. Next, they can use the mobile application to (i) configure (if properly authorized) certain ECU parameters, or (ii) download the ECU software updates from a remote software repository (owned by the OEM) and install them on the ECUs of their vehicle. This case is documented as a box-and-arrow reference architecture and a handful of pages containing text.

Org B– HIL testing. The analyzed system is a simulator with hardware in the loop (HIL). The platform allows the execution of automated tests of autonomous driving components. In the analyzed scenario, an authenticated test operator can schedule performance tests on a piece of embedded software. The component is rigged to the system, which provides simulated sensor data and collects performance measurements. The platform executes the appropriate test cases and provides the obtained measurements to the test operator (while also storing them in a private cloud).

5.3.4 Participants

In each organization, we divided the participants in two teams, with each performing the threat analysis of the same case with a different technique. Within each organization, the two teams had similar size and comparable expertise. A thorough discussion of the seniority level and security expertise of the participants (across organizations) can be found in Section 5.4.4.

Org A The participants are industrial experts with some experience in threat analysis. We assembled two teams with 3 (Stride) and 4 (eStride) members. The eStride team had an additional member, a threat analysis trainee. Each team member had an assigned role (process enforcer, security expert, and domain expert) according to their expertise. The roles were assigned to reflect how threat analysis sessions were performed within the organization. The role of the process enforcer is to ensure that the team was performing the analysis in accordance with the prescribed procedure, and that the discussions (typically about attack feasibility) do not loop or stray to unrelated topics. Security experts take the lead in suggesting attack scenarios, and the role of the domain experts is to describe technical details required to contextualize the attack to the system under analysis. The trainee in eStride was assigned the role of a security expert, given the background of the participant.

Org B The participants are industrial experts with deep knowledge of the case but with no prior threat analysis experience and little security background. We assembled two teams with 3 members each. Differently from the other company, there was no strict role separations among members of the groups. This is in line with the way of working at the company. All members played a mix of both domain and security expert. Additionally, each team had one student member (doing an industrial MSc thesis at the company) that joined as observer. The students did not contribute to the work of the teams.

Supervision of participants. The experimenters were present in all analysis sessions for two reasons. First, they monitored the participants to ensure that the analysis was indeed performed according the instructed procedure. Second, they were taking notes and making observations, which were used to support the data analysis afterwards. We remark that the experimenters strictly refrained from influencing the analysis in any respect and did not contribute to the discussion. In Org A, the experimenters joined purely as observers. As the teams in Org B were unexperienced with STRIDE, the experimenters also answered procedural questions.

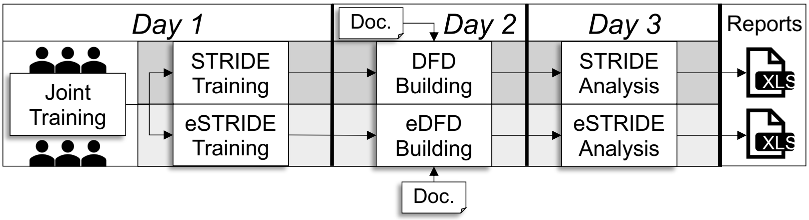


Figure 5.3: The execution of the study in both organizations

5.3.5 Task

In both organizations, the two teams were presented with the same task: perform a threat analysis on the industrial case using the prescribed technique. Both teams were asked to (1) build a diagram based on the architectural documentation, and (2) analyze the diagram according to the assigned technique.

Building. The type of diagram to be built differed across teams: a DFD for the STRIDE teams and an eDFD for the eSTRIDE teams. As described in Section 5.2, we remind that eDFD are richer models that require, among other things, more in-depth thinking about the assets. During this phase, the participants resorted to their domain knowledge and the available system documentation.

Analyzing. The second phase required a discovery of as many threats as possible given the available time. The STRIDE teams performed a systematic, element by element exploration of the diagram. In contrast, the eSTRIDE teams performed a guided analysis of each end-to-end scenario containing high-value assets (see Section 5.2). In addition, the STRIDE teams were asked to assign priorities (i.e., risk values) to the identified threats at the end. Along with reported threats, the threat priorities were assessed by experimenters. In contrast, the eSTRIDE teams were asked to assign priorities to assets before analyzing the diagram. The experimenters assessed the asset priorities, but also marked the threat priorities to enable a comparative quantitative analysis. We remind the reader that threat priorities are approximated by considering the likelihood and impact (i.e., severity of compromising an asset objective). The priority of the compromised asset objective is a deciding (but not the sole) factor in determining the threat priority.

Reporting. As soon a threat was discovered, it was documented in a report, which had to be submitted electronically at the end. The reports contained a list of the identified threats, their locations in the diagram, attack scenarios exemplifying the threats, and estimated priority of each threat (in case of STRIDE). The participants were given a template for the report, in the form of a spreadsheet. The purpose of the template was to simplify and standardize our analysis of the results.

5.3.6 Execution of the study

Figure 5.3 depicts the execution of the study in both organizations. The study was split into sessions (3 hours per day in ORG A and about 5 hours per day in ORG B) taking place on site on three separate days during the same week.

The authors supervised all sessions. All the material (i.e., documentation of the industrial case, training material, description of task, report template) was shared with the participants a week in advance. They were strongly encouraged to read all the material (about 20 pages) before the start of the study.

Day 1: Training. The teams were separated and specifically trained to accomplish their task. The training consisted of brushing up basic security concepts (Joint Training in Figure 5.3), understand how to build DFDs (or eDFDs), and learning how to perform STRIDE (or eSTRIDE).[1] The training sessions contained several hands on exercises for the participants. Due to their limited security expertise, in Org B we planed a longer training session (Org B: 5h vs Org A: 3h), in which we elaborated more on the security concepts, including security threats and countermeasures.

Day 2: Building diagram. On the second day, the teams were given printed copies of all the material and started worked on the task (see Section 5.3.5). Once the participants finished building the diagram, they were allowed to continue with the diagram analysis. In proportion to the complexity of the system under analysis, we allotted only 3 hours per day to simulate realistic time constraints in Org A. We intentionally relaxed this constraint in Org B (5 hours) to avoid overloading the less security experienced group of participants and putting them under stress.

Day 3: Analyzing the diagram. On the third day, the teams were given the same printed material, and the diagram they had created the previous day. They continued where they left off until they finished with the entire task. All the teams finished in the alloted time, and some even finished early (namely, the Stride team in Org A and the eStride team in Org B).

5.3.7 Qualitative measures

We have collected voice recordings of analysis sessions in Org A and took detailed hand notes of those sessions in Org B. This material has been analyzed to answer RQ3.

Org A The tape recordings have been manually transcribed by the first author using dedicated software.[2] The manual transcription process helped the experimenters gain a deeper understanding of the recorded material. After having a thorough understanding of the recordings, the first author *coded* the transcriptions. Coding is a technique for systematically marking chunks of transcriptions. The analysis of code occurrences reveals trends and supports a qualitative analysis. Table 5.2 depicts the hierarchy of codes we used. We coded activities and events related to *diagram building*, *analysis of diagram*, *support activities*, and *detours*.

Activities are durable actions of participants, such as drawing on the board and architecture abstraction and refinement, which occurred during the diagram building phase. Regarding diagram analysis, we coded the activities of attack scenario development, threat feasibility discussion, threat consequence, and the like. We also coded detour activities (e.g., terminology discussion) and support activities (e.g., pointing on the board) to better understand activity patterns.

[1] In Org A, we had a separate, short training session with the process enforcers to remind them to monitor the progress and speed up the discussion, if necessary.

[2] `https://www.qsrinternational.com/nvivo/home`

Table 5.2: Codes used to mark threat analysis activities and events. Codes for events are marked by the † symbol

Activity groups	Coded activities and events
Building diagram	Drawing on the board
	Architecture abstraction/refinement
	Asset analysis
	Extending the diagram
	Focusing on critical architecture
	Scope discussion
	Making an assumption†
Analyzing diagram	**Attack scenario development**
	Domain discussion
	Threat feasibility
	Threat consequence
	Threat prioritization
	Threat reduction
	Using assumption†
	Updating diagram†
	High-priority threat found†
	Low or Medium-priority threat found†
Support activities	Pointing at board†
	Referring to task description†
	Referring to assumptions†
	Referring to case document†
	Referring to training material†
	Break†
	Unsure
	Documenting
Detour	Chatting
	Difference in opinion
	Terminology

On the other hand, events (marked with † in Table 5.2) are instantaneous participant actions. For instance, while discussing about the scope of the analysis the participants may have made an assumption about trusting an external entity, which they documented immediately. Certain events were only possible to code after having assessed the handed-ins. In particular, the event of correctly discovering a high-priority threat could be coded in the transcriptions after having assessed the threats and their priorities. Therefore, we revisited the transcriptions to manually insert such codes.

Org B Recording of the analysis sessions was not allowed by Org B. Therefore, the experimenters took detailed notes about the activities occurring during the sessions, including timestamps of such activities and events. These notes have been labeled, using a subset of codes in Table 5.2 (see codes in bold). We could not use all the codes because it would be impossible to for the experimenters to reliably keep track of that many activities in their note. Nevertheless, we believe that the subset of the codes is representative and did not have an impact on the quality of the analysis.

Interviews. To answer RQ4, we have organized short interviews (20 minutes for each team) at the end of the third day. The semi-structured interviews contained a prepared list of questions (6), which helped us better understand their background knowledge and experience. Namely, we inquired about their perceived progress, the challenges they experienced, and asked them to explain how they approached the task at hand (e.g., What was your strategy for visiting the diagram and why?). The collected information (in the form of notes) was used as a complement to the questionnaire (see below) to better understand their background knowledge and experience.

5.3.8 Quantitative measures

Questionnaire. To answer RQ4, we designed a brief entry questionnaire. We used the entry questionnaire to collect the background of participants, in terms of the years of professional experience, their familiarity with security (on a scale with 4 levels), and their prior experience with threat analysis (on a scale with 4 levels).

Reports. To answer RQ1 and RQ2, we assessed the reports handed in by our participants. The hand-ins included pictures of the created diagrams (DFD or eDFD), a list of security assumptions made during the analysis, and a list of identified threats (documented according to the provided template). The reported threats have been assessed and marked as correct (true positives) or incorrect (false positives) by the first author. For the cases where the assessor was not feeling fully confident (5 to 10 threats in each organization), we had a discussion with the industrial experts (in both organizations) in order to come to an agreement. This also acted as a form of quality control for the assessment made.

A true positive (TP) is a correctly identified threat. This means that: (a) the participants found the threat in the correct diagram location, (b) the participants found a realistic attack scenario for the security threat or the participants found a security vulnerability, and (c) the threat is correct with respect to the assumptions the participants made.

A false positive (FP) is an incorrectly identified security threat. This means that the participants found the security threat in the wrong location or the threat is not correct with respect to the assumptions the participants have made. Additionally, some reported threats had insufficient information for us to assess them. In these cases, we followed a strict approach and marked them as false positive as well.

With the above measures, we compute the aggregate indicators of precision ($P = \frac{TP}{TP+FP}$) as the ratio between correctly identified threats and all reported threats.

We measured the time it took for participants to complete the task. In particular, we measured the time it took for participants to complete the 3 phases of the task. For Stride these were (1) building the diagram, (2) analyzing the diagram, and (3) *prioritizing the threats*. Similar, for eStride the phases were (1) building the diagram, (2) *extending the diagram*, and (3) analyzing the diagram. We do not include coffee breaks in the measured time. Accordingly, we compute productivity ($Prod = \frac{TP}{TotalTime}$) as the amount of correctly identified threats per hour.

In contrast to eStride, the Stride teams also handed in the threat priorities (as high, medium, low). This is due to the different technique procedures, as explained in Section 5.2. However, we did not rely on the reported threats priorities of Stride, and instead assessed the threat priorities for both teams ourselves. The assessment of threat priorities was also discussed together with our industrial partners (in both organizations). Similar to the threat assessment discussions, a set of 5 to 10 threat priorities were in focus. There were only 2 disagreements (in Org B) regarding threat priority (2 high

Table 5.3: Correct (TP) and incorrect (FP) threats reported by the teams

	Org A			Org B		
	Stride	eStride	Common	Stride	eStride	Common
TP	12	13	6	40	32	14
FP	15	0	-	12	14	-
Precision (%)	0.4	1		0.8	0.7	

threats were marked with a medium priority).

5.3.9 Additional quantitative measures in Org A

To answer RQ3, we also analyzed the transcribed recordings. As mentioned before, coding the transcriptions enabled us to track the exact location of a particular activity or event in the transcript (e.g., position index in the text where the activity starts). Therefore, we analyzed the locations of the codes in the transcriptions and the spatial distance between them. The spatial distance between codes is a proxy measure of time distance between activities. We used the distance in the text instead of the actual time distance for convenience reasons, as the transcription software did not provide timestamps for the transcribed text. However, the spatial distance has some advantages. For instance, it is insensitive to a pause in the conversation. The *normalized average distance between codes* was measured as the average number of characters separating the (starting indexes of the) occurrences of each two codes, normalized to the total length (in characters) of the transcription.

5.4 Results

For each research question, this section reports the results of the objective analysis of the collected data. We further discuss and answer each research question in Section 5.5.

Before we dwell on the RQs, Table 5.3 reports the observed levels of true positives and false positives in each team (Stride vs eStride) of the two organizations (Org A vs Org B). The table also shows the threats that are in common across each pair of teams (this is further discussed in Section 5.5. Within organizations, the number of correctly reported threats (TPs) is similar (Org A: 12 vs 13 and Org B: 40 vs 32). In Org B the amount of mistakes is similar between treatments (Stride: 12 vs eStride: 14). Therefore, the precision of the teams in Org B is similar (Stride: 0.8 vs. eStride: 0.7). However, in Org A the Stride team reported 15 incorrect (FPs) threats while the eStride team reported none. Most of the incorrect threats of this team (13 out of 15) were marked as such due to not having sufficient information for their assessment (see Section 5.3.8). This is reflected in the measured precision (Stride: 0.4 vs eStride: 1).

The results concerning the precision are inconclusive due to the strict assessment in Org A (i.e., threats not having sufficient information are marked as incorrect), but, otherwise, we have not observed major differences between

Table 5.4: RQ1. Productivity

	Org A		Org B	
	Stride	eStride	Stride	eStride
Building DFD (h)	1h10	0h15	1h55	1h05
Extending DFD (h)	-	2h20	-	1h15
Analysis (h)	2h20	2h35	4h15	3h25
Prioritization (h)	0h20	-	0h50	-
Total time (h)	3h50	5h10	7h00	5h45
Productivity (TP/h)	3	2.6	5.7	5.6

Table 5.5: RQ2. The correct threats (TP) are broken down into high (H), medium (M) and low (L) importance.

		Org A			Org B	
TP	Stride	eStride	Common	Stride	eStride	Common
H	4 (33%)	8 (62%)	4	6 (15%)	15 (47%)	4
M	2 (17%)	1 (1%)	-	22 (55%)	10 (31%)	4
L	6 (50%)	4 (37%)	2	12 (30%)	7 (22%)	6
Total	12	13	6	40	32	14

the two techniques.

5.4.1 RQ1: Productivity of teams

As shown in Table 5.4 (bottom line), the productivity levels are pretty similar across the two techniques. As discussed later in Section 5.4.4, there are some differences in productivity between the two organizations.

We have looked into the time it took for each team to accomplish the different parts of the given task. Across organizations, it took the eStride team less time to build the the initial DFD diagram (Org A: 0h15 vs 1h10 and Org B: 1h05 vs 1h55). However, the eStride teams had to put in significant extra time to extend the diagrams with the necessary security information (see Section 5.2). In Org A the time spent on analysing the diagrams and identifying the threats was similar across teams, while in Org B the eStride team spent about 1h less on this task. Finally, the Stride teams had to put in additional effort to prioritize the identified threats at the end (this activity is not necessary in eStride).

5.4.2 RQ2. Discovering high-priority threats

As shown in Table 5.5, in both organizations, the eStride teams have found more high priority threats (Org A: 62% vs 33% and Org B 47% vs 15%). Only a part of the discovered threats were common, therefore we have observed that eStride is inclined to produce more high-priority threats compared to Stride.

Figure 5.4 depicts when the teams discovered high-priority threats in Org A.[3] We have manually inspected the recordings to recover the time-stamps of the discovered high-priority threats due to the importance of this code. The Stride team started analyzing the diagram one hour into the first day and

[3] This data in not available in Org B because we were not allowed to tape the sessions.

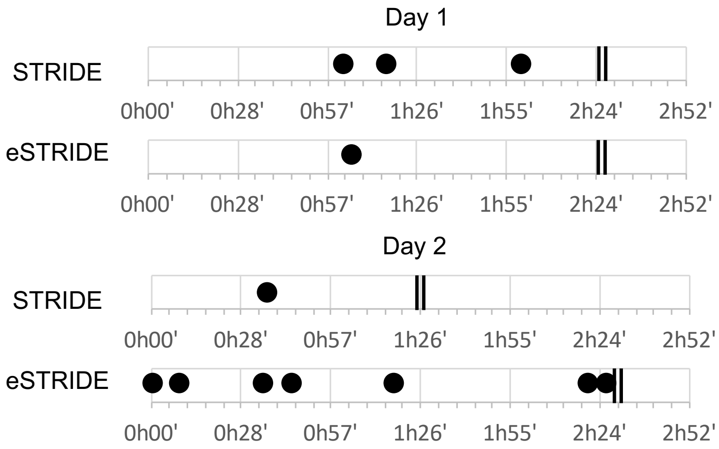


Figure 5.4: RQ2. High-priority threats (dots) discovered by the Stride and eStride team in Org A. The end of each session is marked with a double vertical bar.

discovered most high-priority threats during the first day. The last high-priority threat was discovered by the Stride team about 40 minutes into the second day. The eStride team started analyzing the diagram on the second day, hence they discovered most high-priority threats during the second day. Yet, they discovered one high-priority threat already during the first day while discussing security objectives of assets and their values. Compared to the Stride team, the eStride team did not find high-priority threats faster.

Although we do not have precise measurements, we have informally observed a similar trend regarding the discovery time of high-priority threats in Org B. In addition, we remark that many high-priority threats were found around the trust boundaries (in both teams), therefore the strategy of visiting the diagram may have an impact on the timely discovery of high threats.

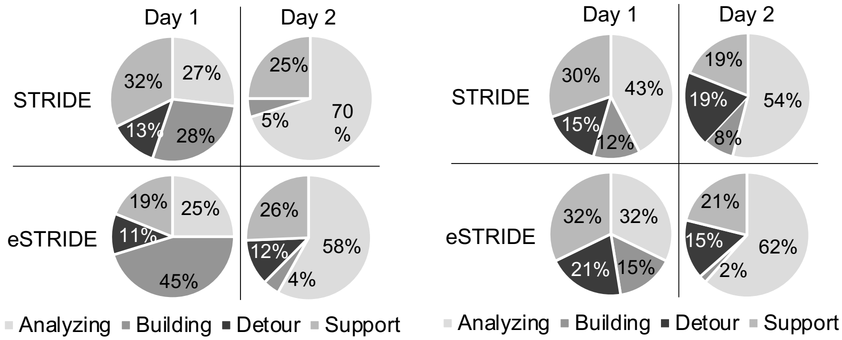
5.4.3 RQ3. Focus on activities and activity patterns

First, we report on the activity focus in both organizations. Then we report on time-lines of activities, and activity patterns for teams in Org A.

5.4.3.1 Focus on activities

The focus of activities was observed by analyzing the coded transcriptions (for Org A) and the structured notes (for Org B). Figure 5.5 shows the prevalence of each of the four activity groups: building the diagram, analyzing the diagram to identify threats, performing support activities (like taking breaks, referring to task description, training material, etc.), and detouring from the task (essentially, loosing focus and wasting time). We remind the reader that we used a subset of all codes from Table 5.2 in Org B.

Org A. During the first day, the Stride managed to complete the diagram and started the identification of threats. In general, the team did not focus on one particular activity. Building the diagram covered 28% of the transcription.



(a) Org A

(b) Org B

Figure 5.5: RQ3. Distribution of activities in both organizations

The diagram analysis covered 27% of the time (mainly discussing the domain and developing attack scenarios). The team detoured often from the prescribed procedure (13%). Finally, the Stride team was involved in more support activities during the first day (32%).

The eStride team invested more on the diagram building during day 1 (45%). They did not identify threats during the first day, however, they performed other analysis activities, namely a thorough asset analysis, for a similar percentage than the other team (25%). Similarly to the other team, the eStride team detoured often during the first day (11%). In contrast to Stride, the support activities amounted to only 19% during the first day.

During the second day, both teams made minor changes to the diagram (5% for Stride and 4% for eStride). The Stride team focused on diagram analysis more than the eStride team did (70% vs 58%), and without detouring from the task. Support activities are comparable.

Org B. Overall, a similar trend of activity focus can be observed for the teams in the second organization. Namely, the Stride team started analyzing the diagram earlier. However, the eStride team did not spend significantly more time on building and extending the diagram. Both teams focused on diagram analysis during the second day (Stride: 54% and eStride: 62%). In contrast to Org A, teams in Org B detoured from the prescribed procedure more often, which is explained by a lesser familiarity of the participants with threat analysis.

5.4.3.2 Summary

Contrary to our expectations, the eStride team in Org B (less knowledgeable) did not spend more time to create and extend the diagram. This is surprising, considering the additional step of asset analysis, which took more attention of the sibling team in Org A. Similar to Stride, the eStride teams still analyzed the diagram on the first day, but focused on the analysis on the second day. We did not observe differences in detour and support activities across techniques.

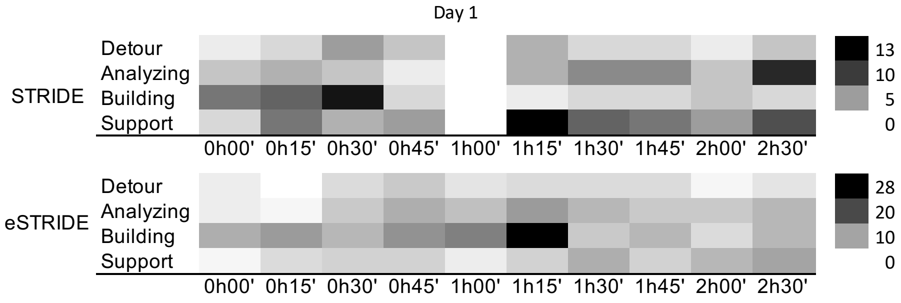


Figure 5.6: RQ3. Day 1: Intensity of activity groups over time for the STRIDE (top) and eSTRIDE team (bottom) in ORG A

5.4.3.3 Timeline of activities in Org A

Figures 5.6 and 5.7 depict the intensity of each activity group over time. The intensity (darker color means more intense) is computed by counting the number of code occurrences for each activity group per ten-minute time frame. Note that, the STRIDE transcription is almost half the size of the eSTRIDE transcription (90,612 vs 151,907 characters). This explains the different proportion of code occurrences in the timelines. In what follows, we discuss the similarities and differences in activities during the first day and the second day.

Similarities (Day 1). In the first 15 minutes both teams focused on building the diagram. In particular, both teams focused on abstracting and refining the architecture, discussing the domain, discussing the scope, and drawing on the board. Other support activities in this time-window include referring to the case documentation. In the span of the entire session, both teams sometimes detoured from the instructed analysis procedure. The detours during the first day are fairly evenly distributed across teams. Both teams made the assumptions during the first day, and made one last assumption about one hour into the second day.

Differences (Day 1). About an hour into the first day, both teams focused on support activities (particularly, referring to case documentation). The STRIDE team finished building the diagram after about an hour. They read parts of the case documentation aloud to validate the diagram before they started to analyze it. On the other hand, the eSTRIDE team started extending the diagram with domain assumptions after about an hour. They verified each assumption by reading the case documentation aloud. The eSTRIDE team started looking for threats only on the second day. In contrast to STRIDE, the eSTRIDE team made, overall, less assumptions and documented them early on. The STRIDE team agreed upon some assumptions but did not document them.

Similarities (Day 2). As instructed, both teams performed activities related to diagram analysis which are accompanied by support activities (mainly, documenting threats). Roughly speaking, the participants alternated between analyzing the diagram and documenting threats. In Figure 5.7, this pattern is more apparent for the eSTRIDE team, as the STRIDE team was very quick in documenting threats. Both teams had a strong focus on diagram analysis in two time-frames (STRIDE 01:10:00-01:15:00 and 01:20:00-01:25:00, eSTRIDE 01:20:00-01:25:00 and 02:00:00-02:05:00). In all four cases, the teams managed

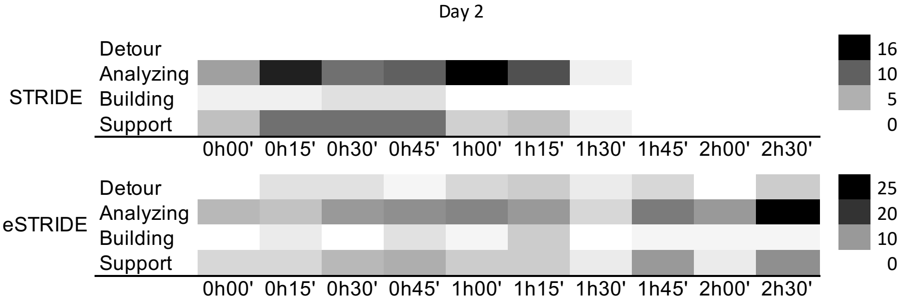


Figure 5.7: RQ3. Day 2: Intensity of activity groups over time for the Stride (top) and eStride team (bottom) in Org A

Table 5.6: RQ3. The average spatial distance (in number of characters normalized by transcription length) between activity pairs in both teams. The top part contains the pairs that have the most similar distances between the two techniques (small values in the third column). The bottom part contains the pairs with the least similar distances (big values)

Activity pairs	Stride	eStride	Δ dist
Threat reduction & Ref. to assumptions	close	close	0.10
Terminology & Domain discussion	close	close	1.70
High-priority threat found & Attack scenario or vulnerability	close	close	1.84
Asset analysis & Updating diagram	far	close	29.0
Ref. to training material & Unsure	close	far	38.38
Scope discussion & Updating diagram	far	close	38.24

to thoroughly analyze one threat in a span of five minutes. This entailed (1) developing attack scenario, (2) using an assumption, (3) discussing threat consequence, (4) determining feasibility, and (5) finding a correct threat. We have observed that focusing on the above-mentioned pattern is beneficial for correctly discovering threats.

Differences (Day 2). Compared to the first day, both teams detoured less from the instructed analysis procedure. In particular, the Stride team did not detour at all. In fact, the Stride team finished about one hour earlier. Compared to eStride, during the second day the Stride team focused less on feasibility analysis and on attack scenario development. The Stride team often updated their diagram during the second day. Concretely, the team merged data flows and removed one external entity and three data stores. Simplifying the diagram helped the team to finish early.

Summary. During the first day the eStride team spent more time building the diagram and during the second day, the Stride team did not detour from the analysis procedure. We further discuss this in Section 5.5.

5.4.3.4 Distance between activity pairs in Org A

We calculated the average distances between all activity pairs for both teams. Clearly, we cannot present all the results. Rather, In Table 5.6, we focus on the activity pairs that have the most similar distances (top) and those with

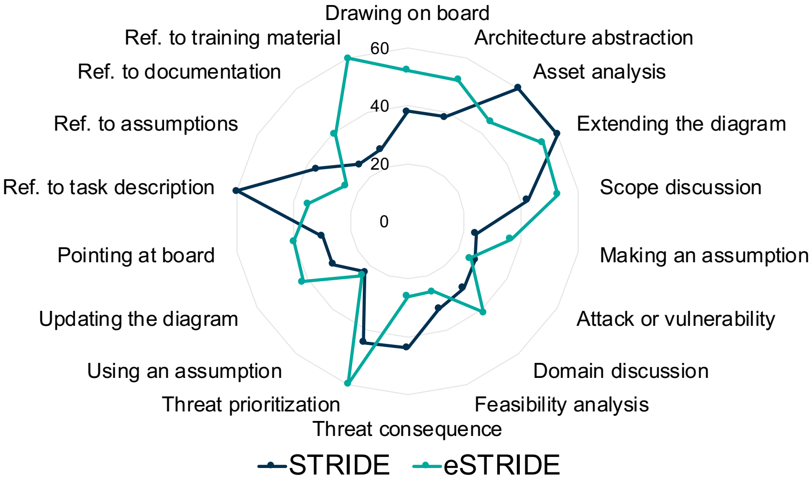


Figure 5.8: RQ3. The average spatial distance (in number of characters normalized by transcription length) between finding a high-priority threat and other activities for both teams

the most different distances (bottom) between the two techniques.

In addition, we analyzed the distances between single activities in relation to all other activities for both teams. In particular, Figure 5.8 shows the average distance between finding a high-priority threat and each other activity.

Similarities. Both teams referred to their assumptions during threat reduction to make sure the reductions do not lead to overlooked threats ($\Delta dist = 0.10$). When the teams referred to assumptions, they read the assumption out loud. In addition, both teams engaged in a domain discussion while clarifying the terminology. Finally, both teams found high-priority threats while developing attack scenarios or identifying vulnerabilities.

Figure 5.8 shows, that the average distance between using assumptions and finding high-priority threats is small in the transcriptions of both teams. The teams used assumptions to justify their reasoning for a threat or vulnerability existence. Therefore, the average distance between referring to assumptions and a finding high-priority threat is small in the eSTRIDE transcriptions.

Differences. In contrast to STRIDE, the eSTRIDE team performed an asset analysis and iteratively updated the diagram with the extra security information ($\Delta dist = 29.0$). In addition, the eSTRIDE team discussed the scope of the analysis while updating the diagram. For instance, they discussed which parts of the system can be left out of the analysis (assumed as trusted). This was not discussed at length in the STRIDE team. During the first day, STRIDE team referred to the training material when unsure.

Compared to STRIDE, the average distance between finding important threats and discussing threat feasibility (and consequence) is smaller in the eSTRIDE transcription (see Figure 5.8). Further, the eSTRIDE team found the first high-priority threat when analyzing the assets and extending the diagram in the first day. Compared to eSTRIDE, the average distance between finding important threats and referring to training and case documentation is smaller in the STRIDE transcription. In fact, the STRIDE team relied more on the

Table 5.7: Entry questionnaire about seniority and security knowledge

Frequencies of the answers				
Q1. How many years of working experience do you have?				
Org A:	1 year (2)	2 - 5 years (2)	5 - 10 years (0)	> 10 years (3)
Org B:	1 year (1)	2 - 5 years (2)	5 - 10 years (2)	>10 years (3)
Q2. How would you rate your familiarity with information security?				
Org A	No background (0)	Security novice (1)	**Security trained (3)**	**Security expert (3)**
Org B	No background (1)	**Security novice (7)**	Security trained (0)	Security expert (0)
Q3. How many threat analysis sessions have you been previously part of?				
Org A	**None (5)**	1 - 5 (1)	5 - 10 (1)	10+ (0)
Org B	**None (7)**	1 - 5 (1)	5 - 10 (0)	10+ (0)

support material, whereas the eStride team relied more on the domain expert. This may be due to factors of team dynamics, rather then the differences in the techniques. Finally, the Stride team made several assumptions during diagram analysis, therefore the average distance between making assumptions are finding important threats is smaller, compared to the eStride transcription.

Summary. For both teams, assumptions played an important role in finding high-priority threats and in reducing threats. In addition, developing attack scenarios and discussing threat feasibility supported finding high-priority threats (more so in the eStride team). However, we are aware that differences in activity patterns might depend on factors related to team dynamics rather then the differences in the techniques.

5.4.4 RQ4. Security expertise

As shown in Table 5.7, we handed out an entry questionnaire to understand the background knowledge and experience in security of the participants. Clearly, we trust the self-assessment of the participants.

From the answers, it is clear that the seniority (Q1) is equally distributed between the participants of the two organizations. Also, the participants have, predominately, no prior practical experience with threat analysis (Q3). Across organizations, however, the participants differed with respect to their knowledge about information security (Q2). In Org A most participants were previously at least trained in security (3) or were security experts (3). In contrast, the participants employed by Org B considered themselves security novices (7 out of 8), at best. We have also inquired about their current role in the organization. In Org A two participants were security consultants, two were hired to conduct threat analysis, and three participants were senior software architects with a security focused role. In Org B two participants were team leaders and the rest were software developers with experience in a variety of programming languages and platforms.

In summary, the above observations confirm the fact that participants from Org A have a high security expertise with respect to Org B. In the following, we summarize the effect of such disparity on the outcomes and execution of the two techniques.

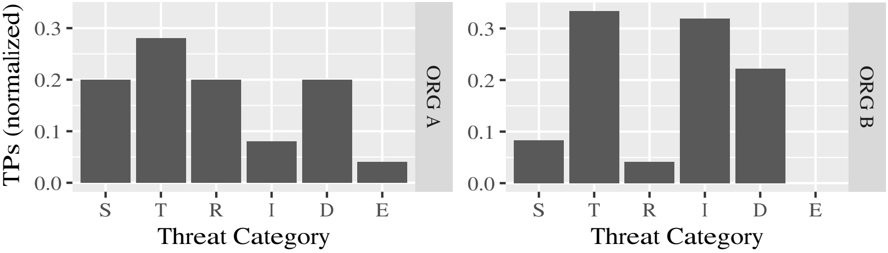


Figure 5.9: The number of correctly identified threats per STRIDE category normalized by the total number of correct threats found in organization

5.4.4.1 Outcomes

Compared to Org A, both teams in Org B made mistakes. Namely, the false positive rate ($FP/(TP + FP)$) is 23% and 30% in Org B, while only one team (Stride) made mistakes in Org A (56%). We remind the reader that most false positives of the Stride team in Org A were assessed as such due to missing information in the documented threat scenarios. The achieved precision (i.e., correctness) of the less experienced teams in Org B is still high (0.8 and 0.7) compared to the best performance achieved by more experienced analysts in Org A.

To gather more insight into the effect of security experience on the quality of the threat analysis outcomes, we observed the distribution of the correctly identified threats over the threat categories (i.e., STRIDE) and made a comparison between the two organizations. Figure 5.9 depicts the number of correctly reported threats per category (irrespective of the technique) normalized by the total number of correct threats in each organization. In Org B (less knowledgeable), the teams found more tampering, information disclosure and denial of service threats, compared to spoofing, and repudiation. Elevation of privilege threats were not reported, which is reasonable considering that these are very technical and often used as a stepping stone for other threats [9]. Incidentally, similar distributions have been recorded in studies observing STRIDE performance in an academic setting [10, 210].

In comparison, the threats found by Org A are more evenly distributed across threat categories. Discussing threat feasibility led teams in Org A to discard many tampering, information, disclosure, and denial of service threats within the trust boundaries of the system. Possibly, the less experienced teams are not able to make such judgments. Another possible explanation is that the teams in Org A modeled smaller diagrams, in particular with respect to the number of modeled processes. Namely, in Org A 16% of modeled elements were processes, while in Org B 26% of elements were processes.

Further, the teams in Org B took about the same amount of time to build and extend the diagrams as teams in Org A (see Table 5.4), but spent more time analyzing threats, which was observed as the most time consuming and challenging task of threat analysis [212]. Despite longer sessions, the less experienced teams in Org B have a much higher productivity (about 6 TP/hour vs about 3). Higher productivity does not, however, imply identification of more high-priority threats. In fact, Table 5.5 shows that more experienced analysts identify a bigger percentage of high-priority threats, no matter the technique used.

5.4.4.2 Execution

Overall, the focus of activities in Org B is comparable to Org A (as observed in Figure 5.5). In both, more support and diagram building activities occurred during the first day, while during the second day the teams focused on diagram analysis. But in Org B, the difference in focus between the teams is smaller. This is explained by their the cross-functional organization of teams, which had a positive effect on team dynamics. Further, we have observed that the less experienced teams in Org B did not discuss feasibility of threats in detail. This observation is in line with the measured high productivity in Org B, as teams rarely got stuck in reasoning about the probability of threat occurrence and its impact. However, the teams in Org B detoured more often during both days. In particular, we have made note of teams inquiring the experimenters on-site for support. This occurred 3 times in the Stride team, and 7 times in the eStride team, where the interactions were either of procedural nature or assurance of progress (e.g., ‘Is the created diagram sufficient, can we move on?’). Despite such insecurities and shallow feasibility discussions, the teams were able to quickly learn and correctly execute the analysis regardless of the assigned technique.

5.5 Discussion

In this section we discuss the results and answer the research questions.

5.5.1 RQ1: Productivity

Beyond counting the amount of TPs, we are also interested in getting insight into whether the two techniques provide different results (conversely, overlaps), in terms of security issues identified. Therefore, we have looked into the correct threats reported by both teams and identified those that are similar with respect to (i) diagram location, (ii) threat category, (iii) vulnerability and threat description. In Org A, six security threats (4 high, 2 low) were correctly discovered by both teams. In this organization, the Stride team discovered 6 threats that were not discovered by the eStride team (2 medium and 4 low-priority). In such cases, the eStride team either skipped some diagram locations by using reductions (2 low, 1 medium) or agreed that the attack is not feasible (2 low, 1 medium). The eStride team discovered 5 threats that went unnoticed by the Stride team. In contrast, these threats were of high (4) and medium priority (1). In these cases, the Stride team could not find any vulnerability or attack. A possible explanation is that the Stride team may not have discussed threat feasibility enough to find feasible attack scenarios or that they were simply overlooked.

In Org B, 14 security threats (4 high, 4 medium, and 6 low) were common for the two teams. The Stride team discovered some security threats that were not discussed in the other team, but many were afterwards marked with a medium (18) or low priority (6). Similar to Org A, the eStride team discovered several (11) high priority threats, which were not discussed in the other team.

Concerning, productivity (**RQ1**), we did not observe a difference between the two techniques. Interestingly, however, the two techniques seem to guide the teams to the discovery of different threats. Furthermore, the eStride teams found more high-priority threats which were overlooked by the Stride teams. In contrast, the Stride teams discovered more threats of low-priority.

5.5.2 RQ2: Discovering high-priority threats

We looked into how the teams approached the exploration of the diagrams and the potential relation to finding high-priority threats. Despite the dictated exploration strategy by the techniques, the teams were still free to choose concrete elements to follow (e.g., which particular data flow, or which asset flow to consider next). For instance, the Stride teams started exploring the diagram starting from one external entity, but then continued differently. Stride of Org A chose to first analyze all the processes, and then proceed to other elements. The Stride in Org B instead explored the diagram following the steps of the scenario (from the documentation) regardless of element types. In both industrial cases, some high-priority threats were located on the trust borders of the system (i.e., external entities, data stores, and processes communicating with the aforementioned). Thus, an *out-side-in* exploration strategy may be useful to find high-priority threats sooner. In addition, an attack on the system boundary element is usually the first step of a chained attack. Hence, accounting for the first steps systematically (at the beginning) may help in discovering chained attack scenarios.

As observed, the eStride procedure helps in discovering more high-priority threats, but to discover them sooner, the technique must emphasise the importance of analysing the asset sources and sinks first. In addition, an out-side-in procedure may support the discovery of chained attacks.

Concerning high-priority threats (**RQ2**), we found that the eStride teams found *twice as many* high-priority threats compared to the Stride teams. Further, all high-priority threats that were discovered by the Stride teams were also included in the reports of the eStride teams. In the context of the conducted case studies, the eStride teams were more complete with respect to finding high-priority threats. Yet, no evidence suggests that eStride can identify high-priority threats sooner.

5.5.3 RQ3: Focus on activities and activity patterns

The time spent in 'detour activities' is significant (between 10-20% in Figure 5.5). Regardless of the technique and organization, the teams often discussed the terminology of the threat categories: in particular, the spoofing category in relation to tampering and repudiation. Perhaps, this could be expected for novice analysts, but it happened consistently in all teams. Generally, such detours (or disagreements) were minimized by the process enforcer steering the discussion. In Org A the Stride team often referred to the material to reach consensus, instead. Possibly, this motivated the team to stay closer to the instructed procedure on the second day (with no detours). The way participants handled detours may depend on the team dynamics.

In Org A, detours often happened when discussing threat feasibility (especially so in the eStride team). This is confirmed by the small average distance of codes for these activities in the transcription. Therefore, feasibility analysis may have slowed down the overall threat analysis. Discussing threat feasibility often leads to estimating the probability of threat occurrence, which is difficult and can lead to 'analysis paralysis', where too much focus is put on a single threat.

In Org B, the feasibility of threats was not discussed in great detail. Therefore this pattern (of slowing down the analysis) did not emerge as prevalent. Further, we have observed that detours happened due to discussing the terminology and domain, rather then feasibility analysis.

Regarding the focus on activities (**RQ3**), we found that in one organization (Org A) the eStride team spent more time on diagram building. Yet, in the other organization (Org B) the eStride team built the diagram faster and still found more high-priority threats. Across organizations, threat feasibility discussion lead to 'analysis paralysis' which slowed down the overall analysis. For what concerns the activity patterns, we found that teams in Org A were careful when making threat reductions, backing those decisions by referring to assumptions. In addition, assumptions were used by teams to justify the existence of threats (in particular high-priority). Our analysis indicates that differences in activity patterns might depend on factors related to team dynamics.

5.5.4 RQ4. Security expertise

Our results show that less experienced teams (in Org B) made more mistakes in their analysis. We looked into the FPs of these teams to better understand their nature.

Most incorrect threats in Org B were duplicates (Stride: 6 out of 12 and eStride: 12 out of 14). Duplicates are threats with the same diagram location, category and attacker scenario. For instance two spoofing threats of an external entity with a slightly different threat description. For instance, in the following example, the second scenario is a special case of the first one:

"1: The attacker can send data from anywhere in the world."

"2: The attacker can pretend to be an operator."

Other mistakes included incorrectly reported locations (we have no explanation here) or threat categories (this is certainly due to a lack of familiarity with the definitions of STRIDE). Finally, some threats were incorrect with respect to the domain assumptions. For example, assuming an existing security measure for logging user actions on a data base, and reporting a threat to accountability of a process reading from that data base.

In summary, and in light of the scarcity of security professionals, the trade off of having more FPs in the analysis results (when using less knowledgeable analysts) could be acceptable in many organizations, particularly in smaller ones. For instance, a security expert could be hired to *validate* the analysis results at the end, which is a much more light-weight and less costly activity than performing the entire threat analysis. In our experience, some mistakes can be also quickly corrected via tool support (e.g., threat locations). However, some mistakes may require more training on security vulnerabilities and classes of attacks (e.g., to avoid infeasible attack scenarios).

Regarding the security expertise (**RQ4**) we found that, in the industrial setting, security expertise may be traded for a faster-paced and less precise threat analysis. The teams with no previous security expertise were able to learn both techniques and perform them effectively. This indicates that there is a benefit in employing less security experienced practitioners to perform an initial threat analysis, the outcomes of which could be submitted for a review by security experts. In fact, a recent study [15] reports that agile organizations currently employ similar strategies for conducting threat analysis.

5.6 Related Work

In this section we position our contributions in the context of related work. First, we discuss related treat analysis methodologies and techniques organized with respect to their risk inclusion. Second, we provide an overview of the related empirical studies.

5.6.1 Threat Analysis with Risk

Risk-first. The main characteristic of risk-first threat analysis is that the outcomes of risk analysis (i.e., to some extend quantified risk of compromised assets) are used as input to the threat identification and analysis. However, little existing literature actually leverages such information during threat identification.

CORAS [25] is a model-driven threat analysis methodology. The approach provides systematic guidelines and tools (e.g., asset, threat, risk, and treatment diagrams) to analyze risk during the design phase. After the creation of asset diagrams (step 3), the analysts conduct a high-level risk analysis, where the most important assets (and their threats) are identified. Similar to eSTRIDE, the purpose of this step is to focus the analysis discussion early-on, without going in deeper detail. Next, the threats are identified by means of structured brainstorming. But, the threats are not identified only with respect to the important assets, and afterwards further risk estimation (part of step 6) and risk evaluation (step 7) is required. In comparison, eSTRIDE suggest a detailed account of risks (with respect to security objectives of assets) and existing solutions (treatments) beforehand, and leverages this information to perform reductions (i.e., the pruned table in Figure 5.2).

Operationally Threat Asset, and Vulnerability Evaluation (OCTAVE) [26–28] is an asset-centric threat analysis methodology. OCTAVE [26] is organized into three phases. In the first phase, assets and threats are analyzed, current practices and vulnerabilities are scrutinized, and security requirements are derived. In the third phase, threats to the most critical assets are used to prioritize the security strategy. But, the risk analysis is conducted after all the threats have been identified. Interestingly, OCTAVE-S [27] is a light-weight variant targeted to smaller organizations, and has a risk-first flavor. OCTAVE-S starts with an asset identification and evaluation of security practices. Similar to eSTRIDE, OCTAVE-S suggests to only identify and analyse threats to important assets. In contrast, the identified threats are still evaluated for impact and probability (i.e., are prioritized), while eSTRIDE aims to skip this step entirely. Further, OCTAVE-S does not provide clear guidelines to determine asset importance.

Risk-last. The common characteristic of risk-last approaches is that the outcomes of threat analysis are used as input for risk identification and analysis. In this respect, the works that follow differ from eSTRIDE.

LINDDUN [11] is a privacy threat analysis methodology. LINDDUN is analogous to STRIDE in that it is model-based (using DFDs), and executes a similar procedure (e.g., using a privacy threat-to-element mapping table) to identify privacy threats. In addition, threat tree patterns are provided by the methodology to help threat identification. After all the privacy threats have been analyzed and documented with misuse cases, the methodology suggests to estimate risk levels for each threat (step 8). Notably, the methodology also provides a mapping of privacy objectives to (more than 40) privacy-enhancing techniques (PETs) which could be used for planning mitigations.

Recently, Affia et al. [213] proposed a risk management approach for e-commerce systems. The authors propose to use STRIDE in combination with Information System Security Risk Management (ISSRM) method. Similar to eSTRIDE, the proposed technique starts with asset identification. In what follows, the threats in [213] are identified by performing STRIDE (on the identified assets) and documented in accordance with the ISSRM method. Essentially, the risk values of threats are determined as soon as they are discovered instead of all at once, as in step 4 in Figure 5.2. Further, instead of using risk information, the authors scope the analysis by eliciting only one threat per each STRIDE category. In contrast to eSTRIDE, there is no notion of systematic threat reduction.

Mollaeefar et al. [214] propose a trade-off analysis technique to solve the problem of analysing risk with multiple stakeholders in the context of privacy concerns. The authors define the problem as a set of (weighted) threats, and a set of security controls associated to said threats. To evaluate the final risk, the technique also takes stakeholders preferences into account. As the threats are input to their risk evaluation, this technique is a risk-last analysis proposal.

PASTA [29] is a methodology targeting business owners for estimating risk by means of attack simulation and threat analysis. The methodology contains seven steps, some of which are similar to the analysis with STRIDE (e.g., creating data flow diagrams, diagram decomposition, determining trust boundaries, and risk and impact analysis - as a final step). In comparison to STRIDE, PASTA suggests a broad list of activities for identifying threats (and vulnerabilities), and modeling attack scenarios.

Semi-automated & risk-centric. Several techniques focus the analysis around system assets and include risk as part of their technique, but are semi-automated, thus the list of prioritized threats is not necessarily the main outcome of the analysis. Though the following works are certainly risk-centric, it is hard to determine the exact stage where risk information is used.

Almorsy et al. [44] propose an automated technique using static security metrics (implemented as OCL constraints) to conduct a trade-off analysis with respect to system security. One of the inputs to evaluate these metrics are so-called Security Specification Models, which (among other) contain security countermeasures, objectives and their priorities. But, the proposed technique [44] may also leverage system descriptions models, and abstract source code representations in the analysis. Though all this information may be used in the final trade-off analysis, not all representations are necessary to evaluate the security metrics. For instance, only two (out of 7) metrics include a condition about the criticality of components (or functions).

Halkidis et al. [135] have developed an approach for a semi-automated risk analysis of threats by analyzing annotated UML diagrams. The authors built a mathematical model of the systems and its defenses, and analyzed it by means of fuzzy fault trees. Similar to eSTRIDE, Halkidis et al. [135] extend the design model with existing security countermeasures (e.g., Secure Pipe). But, the identified vulnerabilities and approximations of risk values are the input for the automated evaluation of risk.

Chen et al. [121] proposed a risk-driven approach for a trade-off analysis of Commercial Off The Shelf (COTS) products. In particular, the authors developed an automated way to extract the vulnerabilities of COTS from a vulnerability database (i.e., CVE), estimate threat risks, and conduct a trade-off analysis by analyzing attack paths.

Finally, in the field of security requirements engineering (SRE) several works [35,130–132,139,215] are centered around system assets (modeled as goals) and may consider their risks. However, threat analysis is usually performed before the requirements are elicited. For an account of related SRE works we refer the interested reader to [208].

5.6.2 Empirical Investigations

Two recent studies [15, 33] conduct case studies to investigate the challenges of performing a STRIDE analysis. In [15] the authors conduct semi-structured interviews in four agile organizations to investigate the perceived challenges by practitioners conducting the analyses. Interestingly, despite the fact that threat analysis is time-consuming, the practitioners of all four agile organizations see value in performing threat analysis at regular time intervals. Similar to this work, the case studies involve industrial practitioners and use the coding technique to discover patterns in the collected data. But, the focus of the mentioned works [15, 33] is to record challenges in agile organizations. In contrast, our work is an empirical comparison of two techniques with respect to performance and execution.

Recently, Stevens et al. [216] conducted a case study investigating the efficacy of threat analysis in an enterprise setting. The authors develop qualitative measures to determine the efficacy of the Center of Gravity (CoG) technique. The CoG originated in the 19th century as a military strategy and is by nature a risk-first technique. The authors design a six-step protocol (including surveys and classroom sessions) and involve 25 practitioners in the study. Similarly to this study, they report a very high accuracy of the results handed-in by industrial practitioners. In addition, they provide empirical evidence for a perceived usefulness of threat analysis even after 30 and 120 days, which is very promising. Our study is novel in that it investigates the timeliness of high-priority threats, and the activity focus of a risk-first and a risk-last technique.

McGraw conducted a study including 95 companies [183]. The study reports on the security practices that are in place in these companies. The BSIMM model does not mention STRIDE per se, rather it highlights the importance of threat analysis. Microsoft has not published evidence of the effectiveness of the STRIDE-per-element technique [9]. Similarly, eSTRIDE (coupled with eDFD) [182] is a recently proposed technique, evaluated solely on the basis of an illustration.

Tuma et al. [210] conducted a controlled experiment comparing the two STRIDE variants, STRIDE-per-element and STRIDE-per-interaction. Similarly to this work, their study quantitatively measures the precision, and productivity of both variants. Their study concludes that there is no statistically significant differences in precision, recall, and productivity of the two STRIDE variants. Yet, the authors speculate that enlarging the analysis scope from one (or two) elements to an end-to-end scenario might have an effect on performance. Their findings are based on quantitative measures, while we adopted a mixed methodology, including a qualitative analysis of recorded sessions.

Scandariato et al. [10] have analyzed STRIDE-per-element and evaluated the productivity, precision, and recall of the technique in an academic setting. The purpose of their descriptive study was to provide an evidence-based evaluation of the effectiveness of STRIDE. Our study, on the other hand, provides a comparative evaluation (by means of a controlled experiment) of STRIDE-per-element and the recently proposed eSTRIDE.

Labunets et al [192] have performed an empirical comparison of two risk-oriented threat analysis techniques by means of a controlled experiment with students. The aim of the study was to compare the effectiveness and perception of a visual technique with a textual technique. The main finding of this study shows that the visual method is more effective for identifying threats than the textual one, while the textual method is slightly more effective for eliciting security requirements.

Existing literature reports on different measures, such as perception of techniques compared to misuse cases (MUC). The work of Karpati, Sindre, Opdahl, and others provide experimental comparisons of several techniques. Opdahl et al. [193] measure the effectiveness, coverage and the perception of the techniques. Karpati et al. [194] present an experimental evaluation of MUC Map diagrams focusing on identification of not only vulnerabilities but also mitigations. Finally, Karpati et al. [195] have experimentally compared MUCs with mal-activity diagrams in terms of efficiency.

5.7 Threats to Validity

With respect to the *external threats* to validity, we consider the threat to generalizability of the results. The study was conducted in two different automotive organizations, yet it is not clear to what extent can our findings be carried over to organizations from other domains. In addition, the number of participants was small (13 in total).

With respect to the *internal threats* to validity, we mention the confounding factors that may have influenced the results. The most important confounding factor is team dynamics. The performance of a team might depend on how well the participants work together. It is virtually impossible to control for this factor in an industrial context, as participants are selected based on convenience and availability.

Another potential confounding factor is the different background knowledge across teams. We control for this factor by dedicating a whole workshop (3 in Org A, and 5 hours in Org B) to training the participants. In addition, we have sent out a short exit survey (with about 10 questions) where we asked the participants whether they felt sufficiently prepared to carry out the task. In both organizations, participants felt like they had a clear understanding of the task, were sufficiently prepared for it, and had a very good understanding of the industrial case under analysis.

We also mention the risk of confirmation bias as some of the researchers are authors of one of the techniques. We mitigated this threat by discussing our assessments (as a form of quality check) with participants (in Org B) and reference experts (in Org A).

Finally, we could not replicate the study in Org B with exactly the same methodology, as the organization did not allowed us to tape-record the sessions. Some measures had to be adapted, which might have led to more imprecise results.

5.8 Conclusion

This study investigates the benefits and shortcomings of performing a risk-first (eStride [182]) compared to risk-last (Stride [9]) threat analysis in an industrial setting. We conducted two case studies with industrial participants employed by two organizations (based in different countries). In this setting, we gathered empirical evidence about the performance and execution of the two techniques. The contributions of this work are three-fold: (i) a quantitative comparison of performance, (ii) a quantitative and qualitative comparison of execution, and (iii) a comparative discussion of the benefits and shortcomings of the two techniques. This study found no differences in the productivity and timeliness of discovering high-priority security threats. Yet, we showed that the risk-first approach produces twice as many high-priority threats (in both organizations). On the other hand, the risk-last technique found more medium and low-priority threats. Further, we find that security expertise may be traded for a faster-paced and less precise threat analysis To find high-priority threats sooner (in addition to their complete account), the eStride procedure can be easily tailored to an out-side-in diagram exploration strategy. An interesting future direction could be observing the performance of the two techniques by conducting a longitudinal study to understand whether eStride's benefits (prioritizing the discovery of high-priority threats) out-weight the limitations (required effort to build eDFDs and sacrificed coverage of low-prioritized threats).

Chapter 6

Paper E

This chapter is based on
Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis,

written by
K. Tuma, M. Balliu, and R. Scandariato,

published in
Proceedings of the International Conference on Software Architecture (ICSA), 2019.

Abstract

This paper presents a practical and formal approach to analyze security-centric information flow policies at the level of the design model. Specifically, we focus on data confidentiality and data integrity objectives. In its guiding principles, the approach is meant to be amenable for designers (e.g., software architects) that have very limited or no background in formal models, logics, and the like. To this aim, we provide an intuitive graphical notation, which is based on the familiar Data Flow Diagrams, and which requires as little effort as possible in terms of extra security-centric information the designer has to provide. The result of the analysis algorithm is the early discovery of design flaws in the form of violations of the intended security properties. The approach is implemented as a publicly available plugin for Eclipse and evaluated with four real-world case studies from publicly available literature.

6.1 Introduction

Security and privacy threats to software systems are a significant concern in many organizations, in particular due to recent legislations regarding data privacy (e.g., GDPR) and upcoming standards about security engineering (e.g., ISO 21434). In essence, there is an increasing push towards adopting methods and techniques that provide security and privacy assurance from the very beginning of a software development project. In particular, this paper focuses on the (architectural) modeling phase, when the structure of a software system is defined. In this context, threat analysis techniques like STRIDE [9] and LINDDUN [217] have gained significant popularity in the industry. Such techniques use Data Flow Diagrams (DFDs) as the notation of choice to represent the key computing elements in a system and to describe how information flows among them. The intrinsic value of such approaches is not under dispute. However, these approaches have two limitations. First, the DFD notation is geared towards business analysts. As such, the notation is very informal as there is no formal semantics attached to the model elements in a DFD. Second, threat analysis techniques like STRIDE hinge on the expertise of the analyst and do not provide any guarantee about the correctness and completeness of the analysis results [10].

Contribution. This paper has the ambition to lift the level of preciseness and automation used in the security-centric validation of design models. In particular, we are inspired by code-level information flow analysis techniques [77, 85], which are here raised to the level of abstraction of DFD-like design models. To this aim, the first contribution of this paper is a lightweight extension of the modeling capabilities provided by DFDs. In particular, the designer (e.g., software architect) has to provide the intended security policy for the information assets that flow in the system. For instance, the designer could specify that the geo-location of the user is private (high confidentiality), and the social network feed is public (low confidentiality). This is achieved by adding a security label to the data flows in the design diagram. In this paper, we focus on data confidentiality and data integrity properties. Additionally, the designer has to specify an abstract input-output security contract for the computational nodes. For simplicity, this is done by choosing from a small set of predefined options. For instance, the designer can specify that the asset on an input flow is copied to an output flow. The second contribution of this paper is a tool-supported, formally-based flow analysis technique that leverages the above-mentioned information to propagate the security labels across the design model (similar to taint analysis [218]). The result of the analysis algorithm is the early discovery of design flaws in the form of violations of the intended security policies. The analysis technique enables the enforcement of security contracts by means of static analysis and for security properties such as non-interference and declassification [77,219]. The approach is implemented using the Viatra framework and packaged as a publicly available plugin for Eclipse [220]. Further, the approach is evaluated with four real-world case studies from publicly available literature.

Novelty. As discussed in the related work, some approaches have automated the threat analysis of DFDs by means of pattern-matching techniques, however, they do not provide soundness guarantees [221]. On the other hand, previous attempts to provide a formal semantics for DFDs have often resulted in a complicated language, hence losing the intuitive flexibility of DFDs [84]. This paper aims at achieving the benefits of a formal analysis technique by retaining the simplicity and intuitiveness of DFDs.

Relevance. Our approach is related to existing secure design practices and can be used to support secure software development. Further, it could easily synergize with existing code-level analysis techniques. In particular, our technique rests on the assumption that the contracts specified by the designer are correct, namely for what concerns the input-output relationships at each processing node. These contracts could be translated to code-level properties and verified at the implementation level. The rationale for a two-tier analysis approach (i.e., model and code) lies on the observation that code-level information flow analysis techniques do not scale well to large systems. Hence, it is preferable to verify localized contracts on smaller, amenable code units and delegate the verification of global, end-to-end policies to the model level.

In conclusion, we remark that our approach can also be used in the context of privacy, e.g., to analyze unintended flows of personal information in a system. Interestingly, as the modeling notation is intentionally kept simple, the approach can be used as a communication medium among several stakeholders: the privacy officers (who identify the privacy-sensitive information), the software architects (who specify the input-output behavior of the processing nodes), and the developers (who have to enforce such behavior in the code).

The rest of the paper is organized as follows. Section 6.2 presents an overview of the approach. Section 6.3 describes a formal model for the security analysis of DFDs, including a description of a security specification language and the underlying semantics of Security Data Flow Diagrams (SecDFD) labels. Section 6.5 describes the real-world case studies and evaluation results. Section 6.6 discusses the relation between the global design-level analysis and the local code-level analysis, and considers limitations of our work. Finally, Section 6.7 discusses the related work, while Section 6.8 presents future work and concluding remarks.

6.2 Overview of the Approach

The analysis approach relies on modeling a Security Data Flow Diagram (SecDFD) using a Domain Specific Language (cf. Section 6.4). Designers are required to invest some effort in modeling the appropriate components to use our plugin. Example SecDFDs used for the evaluation are available in the repository [220]. This section describes the SecDFD and gives an overview of the analysis approach by means of an example.

SecDFD. In a nutshell, a SecDFD is composed of interconnected nodes enriched with security concepts. Generally, nodes represent a piece of code as a series of commands. Each command might take some input from an incoming flow and transform it into an output. Inspired by the model in [222], we define the security semantics for an initial set of commands (dubbed 'node types'). The latter differ with respect to the security contracts, as depicted in Table 6.1. We define four such security contracts.

- Encrypt or hash contract. The contract for encrypting (possibly several) assets always results in propagating a low (public) label on the output flow(s).

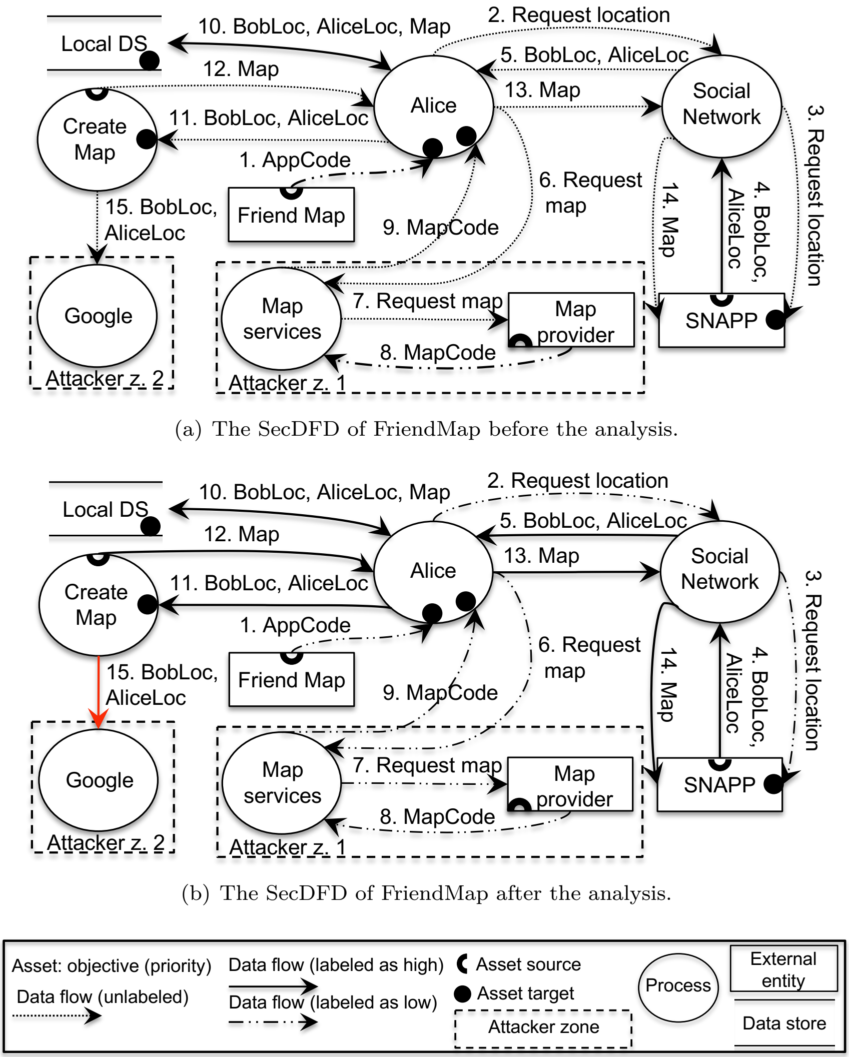
Table 6.1: Type of nodes and their semantics.

Node type	Security contract
Compare, Use, Join, Split	Join
Forward, Copy	Copy
Encrypt	Encrypt
Decrypt	Decrypt

- Decrypt contract. If the input asset is low, decrypting it results in propagating a low label on the output flow. However, if the input asset is high, decrypting it results in propagating a high label on the output flow.
- Join contract. The propagation function for joining two or more assets propagates the label equivalent to most restrictive input asset. This contract is applied to nodes comparing, using, joining, and splitting assets.
- Copy contract. This propagation function will copy the labels of the input assets to the corresponding output flows. This contract is applied to nodes copying and forwarding assets.

Inspired by eDFDs [182], SecDFDs are enriched with assets, their traces and security objectives, and security policies. We focus on tangible information assets and track them from the source node to target nodes. The asset source is a node in the diagram where an asset is first created. The asset target(s) are either nodes in the diagram where an asset rests (where it is stored permanently) or nodes in the diagram where a functionality makes use of an asset (where it has an impact on the application logic). We distinguish between a *global* security policy and a *local* security policy. The designer defines the global security policy by specifying security objectives of initial system assets and attacker observations (i.e., attacker zones). The global security policy stipulates that no information from a confidential input asset flows to a public output asset. The local security policy is defined at the level of nodes, and it can be parametric on the security labels of input and output flows. Attackers are commonly modeled as individual malicious nodes interacting with the system on a particular level of granularity. In this work, we explore the possibility to model the attacker as a set of nodes. We refer to the “attacker zone” as a non-empty set of node elements whose vulnerabilities can be exploited by attackers. For each attacker zone the designer can specify the capabilities of attackers launching attacks in that zone. For confidentiality, the attacker can either observe (read) an asset or not. Designers are able to also run the analysis with a more conservative attacker model, where the attacker can observe assets at all nodes. Finally, the analysis identifies design flaws where the global policy fails in the model. A formal specification language for the SecDFDs is described in Section 6.3.

Local temporal dependencies. The standard DFD does not require to define the sequence of events. However, if a node has several propagation functions the order in which these functions are executed is important. For instance, $forward(c); join(a, b) \mapsto c$ is not equivalent to $join(a, b) \mapsto c; forward(c)$. In the first case, c is forwarded before it is created, thus a label propagation of such a sequence would give a faulty result. In the second case, however, a and b are first joined into a new asset c. Only after the join function is executed, the asset c is forwarded. Our model allows to specify such temporal dependencies.



(a) The SecDFD of FriendMap before the analysis.

(b) The SecDFD of FriendMap after the analysis.

(c) Legend.

Figure 6.1: A SecDFD of FriendMap before the analysis (a), after the analysis (b), and legend (c).

FriendMap example. Figure 6.1(a) depicts the SecDFD of a real-world application for a distributed computing platform developed by Liu et al. [223]. The platform's programming language is based on Java Information Flow (JIF) [224] and it controls the computation and data through security type annotations representing security policies. The application enables users (e.g., "Alice") to create a map of their friends (e.g., "Bob") and to post it on a social network. A client app on Alice's device first downloads the application code and executes it locally. The application fetches the locations of Alice and Bob from the social network and requests a map's code from a third-party map provider (e.g., Google Maps). The map is then created with the respective locations. Finally, Alice can also choose to post this map on the social network.

Analysis algorithm. Algorithm 1 shows the procedure for the analysis. In essence, the analysis propagates security labels according to local security

```
Data: SecDFD
Result: SecDFD'
derive security labels of initial flows;
while every graph node is not visited do
    for function : ordered list of propagation functions do
        propagate security label on the output flow;
    end
    visit next node;
end
if end-to-end view then
    query graph for end-to-end flow;
end
SecDFD' = fire global policy constraints on graph;
```

Algorithm 1: Analysis of a SecDFD.

policies. The input of the analysis is an instance of a SecDFD (similar to the one presented in Figure 6.1(a)). First, the security labels of initial flows are derived from the global security policy. After this step all flows are unlabeled, except for the initial flows (see flows 1, 4, 8, 10 and 15 in Figure 6.1(a)). If the most restrictive security objective on the flow is low, the label is derived as low (e.g. flow 1). If there is one confidentiality objective on a flow the label is derived as high (e.g. flow 4). The global policy dictates that no confidential assets are allowed to be revealed to the attacker. Thus, the security labels of input flows to malicious nodes are derived based on the attacker zone (flow 15). Second, all nodes of the SecDFD are visited. At each visit, propagation functions are executed in the proper order. Every propagation function propagates the security labels according the respective security contract. For instance, in Figure 6.1(a) the Alice node triggers several events for which the correct order is listed in Table 6.2. If the event to forward the location (event 5) triggers right after the AppCode is executed (event 1), then the labels will first propagate on flow 11 (Figure 6.1(a)). When propagating labels, the assets determine which input flows will affect the propagation. In this case, flows 5 and 10 are the input flows transporting the locations. The current labels of flows 5 and 10 are initialized as low, therefore the propagation would incorrectly label flow 11 as low. Instead, the correct propagation requires event 2 to occur before

Table 6.2: The sequence of events for node Alice.

	Event	Input asset	Output asset
1	Execute	AppCode	-
2	Forward	-	Request location
3	Forward	-	Request map
4	Execute	MapCode	-
5	Forward	BobLoc, AliceLoc	BobLoc, AliceLoc
6	Store	BobLoc, AliceLoc, Map	-
7	Forward	Map	Map

event 5. As a result, flow 11 is correctly labeled as high. After this step, all the flows in the graph are labeled and the analysis terminates. Optionally, designers can trace the assets with an end-to-end view. Figure 6.1(b) shows the state of the SecDFD after label propagation. The analysis outcome is the result of verifying the global policy over the annotated SecDFD'. Design flaws are identified where the verification fails (e.g., flow 15).

6.3 Security Analysis for DFDs

This section presents a formal model underpinning the security analysis at the level of Data Flow Diagrams, as described in Section 6.2. The analysis focuses on data confidentiality and data integrity objectives of a system at the time of system's architecture design. The main objective is to introduce a lightweight model that features the advantages of DFDs such as simplicity and usability, yet it contains enough information to reason about the security aspects of a system in a formal manner. Drawing on the theory of information-flow analysis [77], we introduce a security specification language for a system's design that describes the security objectives at the level of individual processes and functions of the system, and their interactions. The specification language supports a system designer in expressing flexible security policies locally for each process. Further, it enables an automatic procedure to analyze system-wide security objectives in an end-to-end fashion, thus unveiling potential security flaws at design time.

6.3.1 A security specification language

We now introduce the security specification language for DFDs. A process consists of a set of function signatures $f(i_1, \cdots, i_n : o)$ using a (possibly empty) set of input assets $i_1, \cdots, i_n$ to compute an (possibly empty) output asset o. We write $f(i_1, \cdots, i_n :)$ and $f(: o)$ whenever the set of input and output assets is empty, respectively. We define the interaction between processes through function composition, by linking the output of a function to the input of another function. As an example, consider the function signatures $getAliceLocation(i_A : o_A)$ and $getBobLocation(i_B : o_B)$ that retrieve the location i_A of Alice and location i_B of Bob, and forward them to the output channels o_A and o_B, respectively. Consider also a function signature $computeDistance(loc_1, loc_2 : dist)$ that computes the distance $dist$ between locations loc_1 and loc_2. We can model a DFD that uses the locations of Alice and Bob to compute their distance by composing function

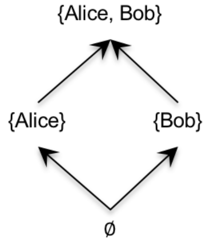


Figure 6.2: Security lattice for confidentiality.

signatures as follows: $getAliceLocation(i_A : o_A); getBobLocation(i_B : o_B);$ $computeDistance(o_A, o_B : dist)$. Note that the matching between function inputs and outputs allows to model temporal dependencies between different functions in the DFD. Specifically, the outputs of the first two functions are used as inputs to the third function. In general, these dependencies induce a partial order between function signatures, which we use to capture the dependencies in a DFD (as represented by arrows). For simplicity, we assume a linearization of evaluation order between function signatures of a DFD, as denoted by the sequential composition of function signatures.

To reason about the security objectives of a system, we enrich function signatures with *security contracts* that enable a system designer to express the security policies *locally*, for each function and process. We then leverage the sequential composition of security contracts to analyze system-wide security policies over DFDs.

Concretely, we enrich the inputs and outputs of a function signature with security labels for confidentiality and integrity. For a given (input or output) asset x, we write $(C(x), I(x))$ to denote the pair of confidentiality and integrity labels of asset x, respectively. For confidentiality, this means that the information stored in the asset x can only *flow to* assets that are at least as confidential as $C(x)$, and, for integrity, it means that the information stored in the asset x can only *affect* assets that are at most as trustworthy as $I(x)$. In this section we focus on confidentiality, noting that integrity is similar through dualization [225].

We assume a bounded lattice of security labels $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$. A label $\ell \in \mathcal{L}$ represents the confidentiality level of an asset. We write $\sqsubseteq$ to denote the ordering relation between security labels and, $\sqcup$ and $\sqcap$ to denote the *join* and *meet* lattice operators, respectively. We write $\top$ and $\bot$ to denote the top and the bottom element of the lattice. Figure 6.2 depicts a security lattice that models the confidentiality objectives for two principals, Alice and Bob. The lattice consists of 4 elements $\mathcal{L} = \{\{Alice, Bob\}, \{Alice\}, \{Bob\}, \varnothing\}$, where $\top = \{Alice, Bob\}$ and $\bot = \varnothing$. The ordering relation $\sqsubseteq$ (displayed by arrows) is set inclusion $\subseteq$, and the join and meet operators correspond to set union $\cup$ and set intersection $\cap$, respectively. For instance, an asset x labeled as $\{Alice, Bob\}$ is more confidential than an asset y labeled as $\{Alice\}$, as defined by the ordering relation $\{Alice\} \subseteq \{Alice, Bob\}$. Hence, the asset y can flow to the asset x, but not vice versa. In fact, the asset x requires the security clearance of both Alice and Bob in order to be observable, while the asset y only requires the security clearance of Alice. An attacker is an observer that can see information with a given security label from the lattice. For instance, if an attacker has security label $\{Alice\}$, the attacker has the security clearance to observe information that is at most as confidential as $\{Alice\}$. In particular, the attacker cannot observe assets labeled as $\{Bob\}$ or $\{Alice, Bob\}$. We remark that by fixing the security label of the attacker, the lattice can be reduced to a two-element lattice, where any asset below the attacker's label in the lattice is considered as public, otherwise it is considered as confidential. In what follows, we use the two-level security lattice $\mathcal{L} = \{\mathbf{L}, \mathbf{H}\}$ consisting of level $\mathbf{H}$ (high) for assets containing confidential information and level $\mathbf{L}$ (low) for assets containing public information. Further, we have that $\mathbf{L} \sqsubseteq \mathbf{H}$ and, for all $\ell_1, \ell_2 \in \{\mathbf{H}, \mathbf{L}\}$, $\ell_1 \sqcup \ell_2 = \mathbf{L}$ only if $\ell_1 = \ell_2 = \mathbf{L}$. Finally, we assume that the attacker has security label $\mathbf{L}$, hence the goal is to prevent the attacker from learning any information about assets labeled as $\mathbf{H}$.

We lift function signatures to *security contracts* $f(i_1^{\ell_1}, \cdots, i_n^{\ell_n} : o^{lbl(\ell_1, \cdots, \ell_n)})$, where $\ell_1, \cdots, \ell_n \in \mathcal{L}$, and $lbl : \mathcal{L} \times \cdots \times \mathcal{L} \mapsto \mathcal{L}$ is a *labeling* function, mapping input labels to an output label. We sometimes write lbl or its definition for $lbl(\ell_1, \cdots, \ell_n)$. Security contracts allow a system designer to assign security labels to the input and output assets of a function. Moreover, the labeling function allows to specify how the security label of input assets affects the security label of an output asset. Security contracts have the unique property of being parametric on the security labels of input assets, and enforcing relationships between input and output labels. Label parametricity is an important feature at the design phase where processes and functions are designed in isolation and the system's security policy may still be unknown. In fact, the same process or function can have different security labels, depending on the context in which it is used. Finally, we remark that concrete (non-parametric) security labels can still be expressed through constant labeling functions, as in $lbl(\ell_1, \cdots, \ell_n) = \mathbf{H}$.

Because security contracts are an extension of function signatures with security labels, they can be composed through function composition in a similar manner. To prevent information leaks from confidential assets to public assets, we can ensure that the output label of a security contract is at most as confidential as the input label of the corresponding security contract. We achieve this by using the ordering relation from the security lattice, as in $f(: x^{\ell_1}); g(x^{\ell_2} :)$, where $\ell_1 \sqsubseteq \ell 2$. The following example elucidates the security contracts.

Example 1 *Consider the design of a system that calculates the distance between a public location, e.g., the location of a restaurant labeled as **L**, and a confidential location, e.g., the location of user Alice labeled as **H**, and sends the distance to the user Charlie labeled as **L**. A system designer specifies the following security contracts:*

- *getLocation*$(i^{\ell_1} : o^{lbl(\ell_1)})$ *and* $lbl(\ell_1) = \ell_1$*, constraining the output label to be the same as the input label.*
- *computeDistance*$(loc_1^{\ell_1}, loc_2^{\ell_2} : dist^{lbl(\ell_1, \ell_2)})$ *and* $lbl(\ell_1, \ell_2) = \ell_1 \sqcup \ell_2$*, constraining the output label to be the same as the join of input labels.*
- *sendDistance*$(i^{\ell_1} : o^{lbl(\ell_1)})$ *and* $lbl(\ell_1) = \ell_1$*, constraining the output label to be the same as the input label.*

The system can be designed by compositing the security contracts as follows: $getLocation(i_A^{\ell_A} : loc_A^{\ell_A}); getLocation(i_R^{\ell_R} : loc_R^{\ell_R}); computeDistance(loc_A^{\ell_A}, loc_R^{\ell_R} : dist^{\ell_A \sqcup \ell_R});$

$sendDistance(dist^{\ell_C} : o^{\ell_C})$*, as displayed in Figure 6.3.*

A designer can instantiate the DFD with a security policy for the source

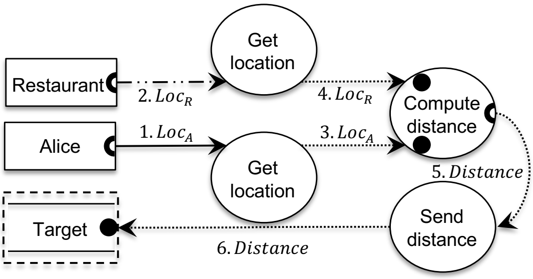


Figure 6.3: A security-centric Data Flow Diagram before the analysis.

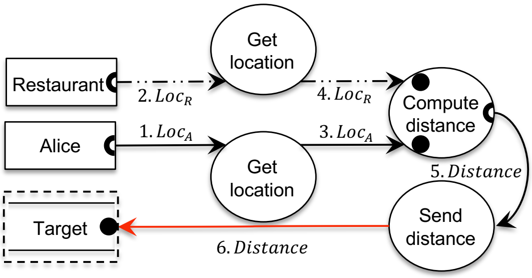


Figure 6.4: A security-centric Data Flow Diagram after the analysis.

and the destination assets. Concretely, the designer defines the security policy by instantiating the input assets as $\ell_A = \boldsymbol{H}, \ell_R = \boldsymbol{L}$ and the output asset as $\ell_C = \boldsymbol{L}$ (cf. Figure 6.3, top). Intuitively, the design above is not secure since, by observing the distance between the confidential location of Alice and the public location of the restaurant, Charlie (an attacker with security clearance $\boldsymbol{L}$) can learn Alice's location. The design flaw can be discovered by solving the constraints between the security labels globally in the DFD. The output label of the distance contract is $\ell_A \sqcup \ell_R = \boldsymbol{H}$, however the input label $\ell_C = \boldsymbol{L}$, which violates the security constraint $\ell_A \sqcup \ell_R \sqsubseteq \ell_C$, since $\boldsymbol{H} \not\sqsubseteq \boldsymbol{L}$ (cf. Figure 6.3, bottom).

6.3.2 Semantics of SecDFD labels

We now present a formal account of security contracts and their use in unveiling security flaws in DFDs. Figure 6.5 displays the syntax of our security specification language. We fix a two-level lattice $\mathcal{L} = \{\mathbf{L}, \mathbf{H}\}$. Security labels l consist of concrete labels ($\mathbf{L}$ and $\mathbf{H}$) and label variables ($\ell \in \mathcal{L}$). Security label *expressions* e consist of security labels l and lattice operations over security labels ($e_1 \oplus e_2$), where $\oplus \in \{\sqsubseteq, \sqcup\}$. We use label expressions to define the security labeling function lbl, and to enforce constraints between security contracts. A Security Data Flow Diagram (SecDFD) consists of (sequential compositions of) security contracts $f(i^{l_1}, \cdots, i_n^{l_n} : o^e)$, where e is the definition of the labeling function connecting input labels to output labels. We use two special security contracts, $src(: o^l)$ and $dst(i^l :)$, to represent explicitly the source and destination assets of a SecDFD, respectively. We use source and destination assets to define a *global* security policy over a SecDFD. Moreover,

$$\begin{aligned} l &::= \mathbf{L} \mid \mathbf{H} \mid \ell \\ e &::= l \mid e_1 \oplus e_2 \\ dfd &::= f(i_1^{l_1}, \cdots, i_n^{l_n} : o^e) \mid src(: o^l) \mid dst(i^l :) \\ &\quad\;\; \mid decl(i^{\mathbf{H}} : o^{\mathbf{L}}) \mid dfd_1; dfd_2 \end{aligned}$$

Figure 6.5: SecDFD Grammar.

$$\textsc{Contr}\quad \frac{\Gamma'' = \Gamma[i_k \mapsto \sigma(l_k)] \qquad \begin{array}{c}\Gamma(i_k) \sqsubseteq \sigma(l_k) \\ \Gamma' = \Gamma''[o \mapsto \sigma(e)]\end{array} \qquad k \in \{1, \cdots, n\}}{\sigma \;\vdash \Gamma\ \{f(i_1^{l_1}, \cdots, i_n^{l_n} : o^e)\}\ \Gamma'}$$

$$\textsc{Src}\quad \frac{\Gamma' = \Gamma[o \mapsto \sigma(l)]}{\sigma \;\vdash \Gamma\ \{src(: o^l)\}\ \Gamma'}$$

$$\textsc{Dst}\quad \frac{\Gamma(i) \sqsubseteq \sigma(l)}{\sigma \;\vdash \Gamma\ \{dst(i^l :)\}\ \Gamma} \qquad\qquad \textsc{Decl}\quad \frac{\Gamma' = \Gamma[o \mapsto \mathbf{L}]}{\sigma \;\vdash \Gamma\ \{decl(i^{\mathbf{H}} : o^{\mathbf{L}})\}\ \Gamma'}$$

$$\textsc{Seq}\quad \frac{\sigma \;\vdash \Gamma\ \{dfd_1\}\ \Gamma'' \qquad \sigma \;\vdash \Gamma''\ \{dfd_2\}\ \Gamma'}{\sigma \;\vdash \Gamma\ \{dfd_1; dfd_2\}\ \Gamma'}$$

Figure 6.6: Semantics of SecDFD labels.

we use the declassification contract $decl(i^{\mathbf{H}} : o^{\mathbf{L}})$ to downgrade the security of confidentiality information and consider it as public, e.g., after an encryption or hashing operation. We remark that our specification language captures the DFD node types from Section 6.2.

Figure 6.6 depicts the semantics of labels for SecDFD. For a SecDFD dfd, security *configurations* have the form $\sigma \;\vdash \Gamma\ \{dfd\}\ \Gamma'$, where Γ and Γ' are *security environments* of type $Var \mapsto e$ and σ is a *label environment* of type $LVar \mapsto \{\mathbf{L}, \mathbf{H}\}$. We write Var for the set of variables that are used in the security contracts, and $LVar$ for the set of parametric labels, i.e., $\ell \in LVar$. The security environment Γ keeps track of security labels during the execution of a SecDFD. The label environment σ instantiates parametric labels with concrete labels. We write $\Gamma(i)$ for the value of a variable i in Γ, and $\sigma(e)$ for the value of an expression e in σ. Moreover, $\Gamma[i \mapsto l]$ denotes a security environment Γ with variable i assigned the security label l. We also write _ (don't care), whenever a symbol is not important.

Intuitively, Γ describes the security labels of variables before the analysis of dfd, and Γ' describes the security label of variables after the analysis. Moreover,

σ describes an instantiation of parametric labels with concrete labels. The rules can be read as follows: If the premises of a rule are satisfied in a security environment Γ and a label environment σ, i.e., none of the security constraints fails, the security contract executes and yields the security environment Γ'.

Each rule models the process of label propagation, the generated security constraints, and the update of the security environment. The rule CONTR ensures that whenever a security contract is executed, the input labels of the matching contracts are upper bounded by the security labels of input of the current contract (cf. $\Gamma(i_k) \sqsubseteq \sigma(l_k)$). This constraint prevents insecure flows from an output of a confidential contract to an input of a public contract. Moreover, we update the security environment by first updating the label of the input variable (cf. $\Gamma'' = \Gamma[i_k \mapsto \sigma(l_k)]$), and then evaluating the label expression in the new security environment Γ'' (cf. $\Gamma' = \Gamma''[o \mapsto \sigma(e)]$). Finally, the analysis produces the security environment Γ'. The rule SRC updates the security environment with the label of an input asset. We use this rule to define the security policy for the input assets of a SecDFD. The rule DST checks whether or not the security label of the input to an output asset is upper bounded by the security label of that asset. We use the DST rule to prevent insecure flows from confidential inputs to public assets, e.g., attacker zones. The rule DECL downgrades the security label of confidential input by making the output public. Finally, the rule SEQ models the sequential composition of two SecDFDs by matching the corresponding security environments.

The security analysis targets insecure flows of information from sources labeled as confidential to destinations labeled as public. To achieve this, a system designer defines a global security policy by specifying security labels for sources and destinations. Following the rules in Figure 6.6, we implement a static analyzer that infers a label environment σ (if it exists) and verifies the correctness of a security policy over the SecDFD. The inference algorithm uses basic constraint solving over the security lattice [85]. Section 6.4 provides details about our implementation.

Definition 1 (Security policy for SecDFD) *Given a set of source assets* $\{src(: o_i^{l_i})\}$ *and a set of destination assets* $\{dst(i_j^{l_j} :)\}$*, a security policy is an assignment of concrete security labels to the source and destination assets, i.e.,* $\Gamma_0[o_i \mapsto l_i, i_j \mapsto l_j]$ *and* $l_i, l_j \in \{\boldsymbol{L}, \boldsymbol{H}\}$*.*

Definition 2 (SecDFD security) *A SecDFD* dfd *is secure wrt. security policy* Γ_0 *if there exists a label environment* σ *such that for any pair of matching security contracts* $c_1 = \_(\_ : x^{l_1})$ *and* $c_2 = \_(x^{l_2} : \_)$*,* $\sigma(l_1) \sqsubseteq \sigma(l_2)$*.*

Intuitively, the security condition requires that there is never a flow in the SecDFD from a confidential output of a contract to a public input of a matching contract. In particular, this ensures the absence of flows from confidential sources to public destinations, thus enforcing the security policy. The following theorem shows that the rules in Figure 6.6 enforce the security condition in Definition 2.

Theorem 1 *Given a SecDFD* dfd*, a label environment* σ*, and security policy* Γ_0*, then* dfd *is secure wrt.* Γ_0 *only if* $\sigma \ \vdash \Gamma_0 \ \{dfd\} \ \Gamma'$*, for some* Γ'*.*

The theorem can be proved by case analysis on the type of matching security contracts and the rules in Figure 6.6. We refer the interested reader to the full version of the paper [220].

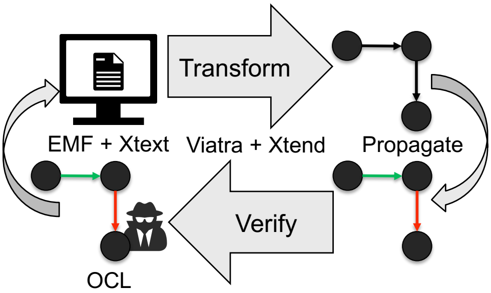


Figure 6.7: The plugin toolchain.

6.4 Implementation

Figure 6.7 shows the plugin toolchain. The approach is implemented as a publicly available Eclipse plugin [220] using the Eclipse Modeling Framework (EMF). We build two meta-models: for modeling SecDFDs, and for end-to-end security views. In addition, we built an external Domain-Specific Language (DSL) using the Xtext framework for modeling SecDFDs. The DSL is accompanied by a simple grammar and a textual syntax. We use the Viatra query engine and the Xtend language to transform SecDFDs to simple graphs. The new graph is visited with a recursive Depth First Search (DFS) algorithm. At each node, the labels of outgoing flows are propagated according to the node semantics. Afterwards, we statically validate the global policy over the resulting graph model. To this aim we write constraints in the Object Constraint Language (OCL). Optionally, the graph model is queried for end-to-end asset traces, namely, all graph elements handling a particular asset. Note that our implementation currently supports specifying local temporal dependencies (at node level) with flow enumeration.

6.5 Evaluation

We evaluate our approach by running the analysis on several open source projects. First, we test our approach on microbenchmarks from DroidBench,[1] an open test suite for evaluating the effectiveness of taint-analysis tools for Android apps. Second, we model four realistic applications, namely FriendMap, Hospital, JPmail, and WebRTC. The analysis of all four applications does not produce any false positives. We have analyzed alternative models with injected design flaws (or security solutions) for all four applications. Further details about the evaluation are available in the repository [220].

We have tested our plugin on several inter-component communication examples. These small examples have built-in design flaws. They were used to verify that the propagation functions work as expected and are able to identify built-in design flaws. The propagation functions were able to identify design flaws in all initial tests. Together with the core security analysis from Section 6.3, the microbenchmark increased our confidence on the correctness

[1] `https://github.com/secure-software-engineering/DroidBench`

of our analysis tool.

6.5.1 FriendMap

We have evaluated our approach on the example described in Section 6.2. Figure 6.1(b) depicts the results of the analysis.

The global policy for this scenario causes the inference of low labels for non-confidential assets. High labels are inferred on flows originating from sensitive locations of Bob and Alice. When fired, the security contracts cause the propagation of high labels to parts of the diagram (e.g., flows 11 and 12). The map code can be malicious, and attempt to leak the location from the browser. A static policy check is able to identify this flaw on flow 15. This is the only design flaw the analysis discovers under attacker zones 1 and 2 (see Figure 6.1(b)).

6.5.2 Hospital

We evaluated the plugin on an application for controlling access to sensitive patient data [223]. We refer the reader to the full paper [220] for figures depicting the analysis. The access control policy and the application code are first loaded to the Employee client (i.e. mobile application). The employees can sent a request to read the list of patients, including the patient HIV status. The request is forwarded to a node that handles read requests. Depending on the given permissions, the node retrieves the sensitive list of patients and forwards it to the employee node, where it is stored on a server. A similar node handles requests for modifying the patient list. We model the attacker as a node attempting to observe the list of patients. The global policy causes low labels on flows where non-confidential assets originate from. However, confidential assets (Patient List and Modified List) cause high labels on the corresponding (origin) flows. The label propagation causes high labels on all flows containing these assets. If the attacker is able to spoof the node handling the read requests (e.g., by injecting false requests), (s)he could gain access to the confidential list of patients.

6.5.3 JPmail

JPmail[2] is an email client implementing a subset of the MIME protocol. It leverages information-flow control to enforce a security policy. In essence, to send an email the user specifies the email body and header. These are reclassified so that email header is public and the email body is encrypted. As such, the email is sent to the SMTP server, which delivers it to the POP3 server. From there, the email recipient is able to retrieve it. The email header remains public, while the email body is decrypted using a (recipient) private key. In this scenario, the SMTP and POP3 servers are common targets of attack as they are exposed to open networks. In our analysis, the global policy causes low labels on the originating flows for non-confidential assets (e.g., recipient public key). Yet, the email body and header are initially confidential, hence the corresponding (origin) flows are labeled as high. At this point, the graph is visited and the

[2] http://siis.cse.psu.edu/jpmail/jpmaildetails.html

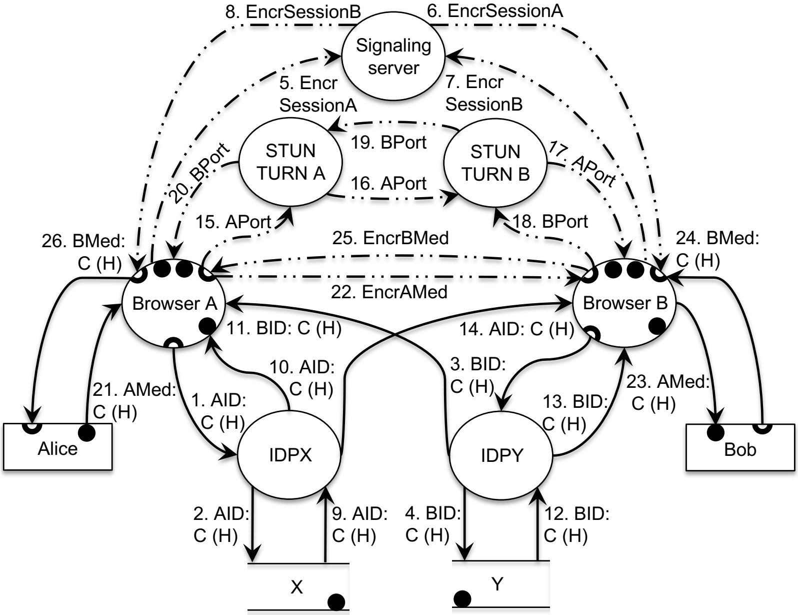


Figure 6.8: A SecDFD for WebRTC after the analysis.

propagation functions propagate labels accordingly. No information can be leaked to the attacker zone due to the modeled declassifications.

6.5.4 WebRTC

We extend our validation by applying our approach on a model of the WebRTC project [3] which facilitates real-time communication capabilities for browsers and mobile applications. In a nutshell, WebRTC provides the infrastructure needed for developing applications that require sharing user media (voice, video, files) between two or more parties. Figure 6.8 depicts the SecDFD for WebRTC after the analysis. First, a peer-to-peer network communication is established via a signaling server using the HTTPS protocol. Afterwards the client browsers obtain each other's Browser IDs. The browsers verify the identity of other involved clients. Next, the calling browser establishes a secure connection via STUN/TURN servers. A transport layer security solution is used to secure the data transfers between the browsers. At each node, we assume the attacker is able to observe the assets. The global policy causes flows 9, 12, 21, and 24 to be labeled as high. Upon visiting the graph, the security labels are propagated as follows. First, the AliceBrowserID is generated and stored on G-mail servers (flows 1-2). IDPX forwards the AliceBrowserID (confidential) causing the propagation of a high label. An analogous propagation happens for BobBroswerID. Nodes forwarding the encrypted session data (flows 5-8) via HTTPS cause the propagation of low labels. A similar propagation happens for flows 15-20. Encrypting AMedia in Browser A causes a low label on flow 22 (similar for flow 25). Decrypting EncrAMedia in Browser B causes a high label

[3] https://webrtc.org/

on flow 23 (similar for flow 26). Our analysis identifies potential design flaws at all nodes with high input/output flows. A second analysis of the WebRTC was performed using a different tool as a sanity-check of our results. We compared our results to the results obtained by an Eclipse plugin introduced by Sion et al. [42]. The authors identify several threats as less critical. The locations of those are in-line with our analysis results.

6.6 Discussion and limitations

A mismatch between intended architecture and implemented architecture is a source of frustration in many organizations. It manifests itself in a loss of resources for large code refactorings, loss of functionality, architectural decay and technical debt. Further, assuring architectural compliance is a hard problem [226]. In the following, we discuss challenges and opportunities of our approach.

Secure model to code. SecDFDs provide a simple and intuitive model for analyzing security policies at the design level. They can help a system designer to uncover security flaws or prove that the design is secure. On the other hand, the security guarantees provided by this approach hold under the assumption that the code implementing the security contracts does not violate the security labels. Such implementation can be verified in a second phase by using existing code-level security analysis. For our case studies, we leverage existing implementations in security-typed languages such as JIF [224].

A major advantage that comes with the two-phase security analysis is *compositionality.* By decoupling the security analysis into a global design-level analysis and a local implementation-level analysis, we only need to check the security of implementations locally, and obtain end-to-end security assurance for the entire system. Compositionality is an important and desirable property for scaling a security analysis to real-world systems. Further, it enables us to relate the results of the verification analysis to semantic security conditions such as non-interference [219], thus providing provable security guarantees. Intuitively, the security condition requires that confidential information is released to attacker zones only in a controlled manner. The design-level security analysis enforces such invariant for a given security policy. Further, the code-level security analysis ensures the implementation of a security contract entails the contract's security labels. Thus it follows that the system satisfies the security condition.

Secure code to model. Significant effort has been put into reverse engineering the architecture from the implementation [227]. Further, existing work aims to extract and analyze the security concepts in the implemented architecture [90]. Yet, existing tools for extracting the architectural design have scalability issues and often do not provide formal guarantees. We see potential benefits in leveraging our approach to lift the implementation to the design level and analyze security on that level of abstraction. For instance, a call graph can be extracted from code. By means of specifying the node types of the call graph (or possibly annotating code methods with node types) and specifying a global security policy, the SecDFD could be extracted from the code. Assuming a correct extraction of the node types, the security analysis could be performed on the *implemented* architectural design.

Limitations. The challenge of identifying the sources and sinks of information, including attacker zones, is also applicable to our work. At the current stage, our approach is best suitable for top-down architectural security design, where arguably the designer is aware of sources and destinations of information and their security requirements. Alternatively, we can leverage existing work on automatic discovery of source and sinks, e.g., the SuSi tool for the Android framework [79]. To our best knowledge, SecDFDs keep the desired simplicity of the DFD notation. However, the usability aspect was not subject to our validation. In the future we plan to conduct user studies to validate the difficulty of performing a SecDFD threat analysis with practitioners. Finally, we recognize potential subjectivity in reporting the evaluation results and remark that a thorough validation is planned for future work.

6.7 Related work

This section discusses related work in automated security analysis, DFD semantics, and information flow analysis.

Automating security analysis of diagrams. Almorsy et al. [44] propose an approach for automating the security analysis by capturing vulnerabilities and security metrics. It is beneficial to analyze system vulnerabilities and system defenses side by side. Similarly to our approach, their model-based approach requires input from the designer to discover vulnerabilities. However, Almorsy et al. [44] do not capture information disclosure threats and security policies. Further, the correctness of analysis results relies on the soundness of the signatures, which are very generic. More importantly, the described signatures for identifying the vulnerabilities are not supported by formal reasoning.

Berger et al. [45] propose an approach for semi-automated architectural risk analysis. This approach leverages a subset of open source vulnerability repositories and a rule checker for pattern matching the system model represented with EDFDs. The use of knowledge-bases has shown to be successful in finding security threats in the past. Yet, it is challenging to interpret vulnerability descriptions from open source repositories into graph query rules. Like us, Berger et al. [45] extend the DFD with data source, target, channels, and confidentiality objectives. In contrast, the SecDFD has more semantical flavor and includes the ability to provide security specifications and attacker zones.

Sion et al. [42] present an approach for a risk-centric threat analysis of DFDs, which enables threat elicitation and risk analysis. The ability to model security solutions in the form of architectural patterns is useful for a more comprehensive analysis. The approach relies on the security expert to provide the initial distribution estimates for the Monte-Carlo simulation which may result in low confidence and precision. Similarly, the authors enrich the DFD with security solutions. Further, their tool uses the Viatra query engine and a graph-based pattern language. In contrast, their approach is risk-centric whereas this work aims to provide security semantics to DFDs.

Jürjens et al. [57] have proposed UMLSec, an extension for UML to model security aspects in system design and prove security properties, such as secrecy. UMLSec's formal semantics scales well as it applies to a variety of model types, e.g., activity diagrams or statecharts. Like SecDFD, UMLSec defines a system as a composition of subsystems and enables modeling a security policy and attackers. In contrast, Jürjens et al. [57] focus the analysis on attacker behavior, rather than the semantics of security labels on flows.

Guerriero et al. [228] propose a privacy-by-design approach for specifying and enforcing privacy policies by code rewriting. Similarly to our work, the authors propose a model and a specification language for policies over sensitive data flows. In addition, the privacy policies are enforced algorithmically on an application data flow model. They introduce privacy-aware operators which enable policy enforcement. Further, their approach is focused on preventing disclosure of sensitive data under certain contextual conditions (i.e., when and how much data can be observed), rather than the way sensitive data is transformed by the system (i.e., security contracts of operations).

Breaux et al. [229] develop a methodology for mapping privacy requirements from natural language text to a formal language. Interestingly, the authors define traces of requirements around particular data (similar to our end-to-end view). However, Breaux et al. [229] focus on specifying policies and identifying conflicts between policies of different actors.

Security semantics of DFDs. Several works approach DFDs from a formal angle, by associating a formal semantics to the model. They aim at extending DFDs with lightweight specifications for expressing functional correctness properties. For instance, Leavens et al. [51] propose a DFD semantics that allows to specify the dynamic behavior of a concurrent system, and Larsen et al. [52] leverage formal specifications in the VDM language to formally reason about DFDs. We refer to the work by Jilani et al. [84] for an overview. In contrast, our work focuses on the *security* semantics of DFDs and it presents a simple label model that enables security analysis at the design level. The simplicity stems from the fact that our label model only focuses on security and ignores the functional correctness of the system.

Information flow analysis at code level. Abdellatif et al. [230] present an approach accompanied by a toolkit to automate information flow control in component-based systems. Their approach requires developers to specify the security properties with a configuration file, which in turn is used to validate the system for potential data leaks, before the security code is generated. Similarly to our work, the authors identify security leaks by automatically checking the security policy at the level of components. Yet, our work is unique with respect to label propagation and label specification for system components. In addition, our work defines attacker zones as part of a global security policy, while Abdellatif et al. [230] do not model the attacker explicitly.

Information flow control is a well-studied research area. A large array of security conditions and enforcement mechanisms have been proposed to address different computational models, languages and systems [77, 218, 219]. Our contribution is orthogonal to these works and it can leverage their results as discussed in Section 6.6.

6.8 Conclusion

In this paper we have presented a formal approach to analyze security objectives of information flows at the design level. The approach focuses the analysis of confidentiality and integrity objectives. We provide a formal definition of a security specification language for DFDs. In addition, we introduce the Security Data Flow Diagram (SecDFD) and provide semantics of SecDFD security labels. We prove security for the SecDFD with respect to a global security policy and a security label environment. We have implemented our approach using the Viatra framework and packaged it as a publicly available plugin for Eclipse. The approach is evaluated on four open source applications. The underlying compositionality of our approach provides opportunity to refine the analysis from a global design level to a local implementation level analysis. In the future we plan to implement mechanisms to refine the analysis on the level of implementation by combining existing information flow analysis techniques such as static, dynamic and hybrid analysis. Further, we plan to extend the node semantics, covering additional node types, such as authentication, authorization, and verification. Finally, we plan further validation efforts with respect to the usability aspects and the analysis of larger open source projects with (and without) known design flaws.

Chapter 7

Paper F

This chapter is based on

Inspection Guidelines to Identify Security Design Flaws,

written by

K. Tuma, D. Hosseini, K. Malamas, and R. Scandariato,

published in

Proceedings of the International Workshop on Designing and Measuring CyberSecurity in Software Architecture (DeMeSSA), 2019.

Abstract

Recent trends in the software development practices (Agile, DevOps, CI) have shortened the development life-cycle causing the need for efficient security-by-design approaches. In this context, software architectures are analyzed for potential vulnerabilities and design flaws. Yet, design flaws are often documented with natural language and require a manual analysis, which is inefficient. Besides low-level vulnerability databases (e.g., CWE, CAPEC) there is little systematized knowledge on security design flaws. The purpose of this work is to present and evaluate a catalog of security design flaws accompanied by inspection guidelines for their detection. To this aim, we conduct empirical studies with master and doctoral students. This paper presents a catalog of 19 inspection guidelines for detecting security design flaws and contributes with an empirical evaluation of the inspection guidelines. We also account for the shortcomings of the inspection guidelines and make suggestions for their improvement with respect to the generalization of guidelines, catalog re-organization, and format of documentation. We record similar precision, recall, and productivity in both empirical studies.

7.1 Introduction

Recent trends in software development, such as, Agile, DevOps, and Continuous Integration (CI), have shortened the software development life-cycle, impacting software security [231, 232]. For instance, CI tightened release cycles to days, or sometimes hours. This limits the activities that can take place for security analysis, causing the need for *efficient* security-by-design approaches. In the design phase of the development life-cycle, software architectures are often analyzed for potential design flaws and vulnerabilities. Knowledge reuse is an important factor that can help raise the efficiency. For instance, previous work ([44, 45, 233], to cite a few) has made use of publicly available records of low-level security vulnerabilities, such as CAPEC ,[1] CVE ,[2] CWE [3] to semi-automate the security analysis of systems. On the level of software architecture, Garcia et al. [234] introduce a catalog of architectural bad smells specified with UML diagrams. Similarly, Bouhours et al. [235] contribute with a catalog of 23 so called “spoiled patterns” or, architectural design antipatterns. Yet, the existing literature about architectural design flaws [234–238] lacks a systematized knowledge about security-relevant architectural design flaws. In addition, there is a lack of practical inspection guidelines for identifying security design flaws in software architectures.

This paper presents a catalog of 19 inspection guidelines for detecting security design flaws. As a key contribution, we have conducted some empirical experiments to assess precision, recall, and productivity when the guidelines are used. The experiments also provided the opportunity to track the problematic guidelines and suggest improvements to the catalog with respect to: (a) generalizing the guidelines, b) re-organizing the catalog, and c) format of documentation. This is another key contribution of this paper.

We observe three metrics for evaluating the inspection guidelines, namely, precision ($TP/(TP + FP)$), productivity ($TP/hour$), and recall ($TP/(TP + FN)$). We record a relatively high precision (92.6%) and productivity (11.5 TP/h). On the other hand, our results show that about half of the security design flaws go unnoticed (average recall is 50.4%). Similar measurements of precision and recall have been reported in related empirical studies investigating knowledge-based manual threat analysis techniques, i.e. STRIDE [10, 210]. During our experiments we have found that many participants have expressed doubts when detecting certain security design flaws. We systematically track which guidelines were problematic and provide an account for the re-occurring issues. Accordingly, we suggest simple improvements to overcome these issues. The detection guidelines can be currently used to manually analyze software design. However, we see potential in using these guidelines to automate the detection of security design flaws. Specifically, the closed questions are already operationalized to some extent, and can be easily transformed into graph queries.

The rest of the paper is organized as follows. Section 7.2 describes the guidelines for detecting security design flaws and shows how they are used. Section 7.3 describes the experimental design and execution. Section 7.4 presents the results and Section 7.5 suggests improvements for the guidelines.

[1] https://capec.mitre.org

[2] https://www.cvedetails.com

[3] https://cwe.mitre.org

Table 7.1: A list of the security design flaws evaluated in this paper.

Name	Description
Missing authentication	An absence of an auth. mechanism in the system.
Authentication bypass	The auth. mechanism does not cover all possible entry points to the system.
Relying on single factor auth.	The auth. mechanisms rely on the use of passwords.
Insuff. session management	Sessions are not managed securely throughout their life-cycle.
Downgrade authentication	Possibility to authenticate with a weaker (or obsolete) auth mechanism.
Insuff. crypto key management	Keys are not managed securely throughout their life-cycle.
Missing authorization	An absence of an authorization mechanism in the system.
Missing access control	An absence of access control in the system.
No Re-authentication	An absence of re-authentication during critical operations.
Unmonitored execution	Uncontrolled resource consumption due to interactions with external entities.
No context when authorizing	An absence of conditional checks for access control.
Not revoking authorization	An absence of a process for revoking user access.
Insecure data storage	Storage of sensitive data is in clear or weak access control mechanisms are in place.
Insuff. credentials management	Credentials are not managed securely throughout their life-cycle.
Insecure data exposure	Sensitive data is transported in clear text.
Use of custom/weak encryption	Generating small keys, using obsolete encryption schemes.
Not validating input/data	Absence of validation checks when receiving data from external entities.
Insuff. auditing	Access to critical resources or operations is not logged.
Uncontrolled resource consumption	Uncontrolled resource consumption of internal components.

Section 7.7 lists the threats to validity, while Section 7.6 discusses the related work. We conclude the paper in Section 7.8.

7.2 Evaluated Security Design Flaws

This section gives a short description of the security design flaws catalog used in the empirical experiments. A more detailed description if the security design flaws listed in Table 7.1 is provided in [239]. We remark that the catalog is optimised and improved as a result of the empirical evaluation, as discussed in Section 7.5.

The catalog is a list of 19 design flaws related to issues with authentication, access control, authorization, availability of resources, integrity and confidentiality of data. The catalog entries consist of (a) the name of the design flaw, (b) a description (using natural language), and (c) a series of closed questions that serve as detection guidelines. Listing 7.1 shows the first catalog entry. The catalog was compiled by systematically filtering vulnerability database entries (CVE, CWE, OWASP,[4] and SANS[5]) and existing threat and vulnerability taxonomies. The final catalog entries were obtained by grouping a filtered subset of database entries and taxonomies. The authors grouped the entries whenever the vulnerabilities could be caused by the same design-level issues. For instance, they relate 3 CWE entries (287, 306, and 862) to the security design flaw “Missing authentication” (Listing 7.1). The CWE entry 287 is a description of an improper authentication, where the software is not able to prove the identify of the actors in the system. Entry 306 is a description of

[4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[5] https://www.sans.org/top25-software-errors/

Design Flaw 1: Missing authentication

Description This refers to the absence of an authentication mechanism in the system. Apart from external entities, like users or other systems the system may interact with, authentication may be necessary within the system between processes/components/datastores that are located in different trust boundaries.

Detection

(i) Consider the external entities (users/subsystems) that interact with the system and which assets of the system they can access.

(ii) Determine the processes that interact with high-value assets in the system.

(iii) For each interaction examine:

(1) If it is an entity: Does the entity go through an authentication point in order to access the asset?

(2) If it is a process: Is the identity of a process accessing datastores or processes in a different part of the system (trust boundaries – requires different privilege levels) verified?

Listing 7.1: Textual description of the missing authentication design flaw.

commonly missing (re)authentication mechanisms when critical functions are executed. Finally the entry 862 is describing the weakness of missing authorization mechanisms system actors communicating with the system. All the above entries are describing weaknesses related to external entities or critical processes communicating with the system without an authentication mechanism in place. Therefore, the design issue causing all three entries is missing authentication. Notice that some inspection guidelines are overlapping (e.g., "Missing authentication" and "No Re-authentication" from Table 7.1), therefore the catalog is improved as discussed in Section 7.5. A detailed procedure of the catalog compilation can be found in the original study [239].

7.3 Empirical Experiments

Research questions. We conducted two controlled experiments with participants to investigate the performance of using existing guidelines for detecting security design flaws in a software architecture. The first experiment was conducted by the second and third author with master students. The second experiment was conducted by the first author with doctoral students. We were interested to measure the efficiency and effectiveness of the guidelines in both experiments. We observe effectiveness of the guidelines with a measure of *precision* and *recall*, and efficiency with a measure of *productivity.* This study focuses on the following research questions.

RQ1. What is the precision, recall, and productivity of the proposed guidelines for security design flaw detection?

RQ2. What are the shortcomings and benefits of the proposed guidelines for security design flaw detection?

Experimental object. Figure 7.1 shows the context DFD of the experimental object. The Home Monitoring System (HomeSys) is a system for remotely monitoring private residents. Its purpose is to provide the infrastructure and functionalities for customers to automatically receive and manage notifications about critical events in their homes. In principle, the system consists of a

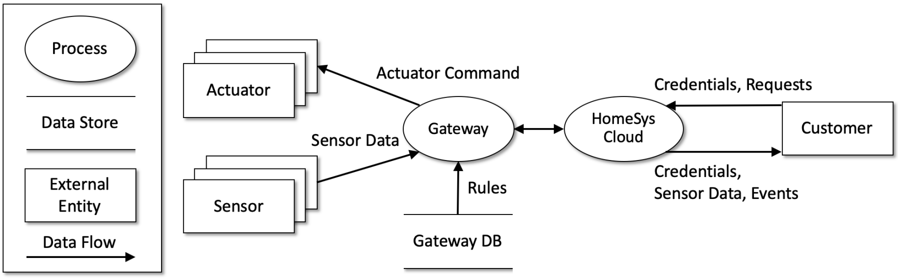


Figure 7.1: A high-level DFD of the experimental object.

gateway communicating with sensors, and a cloud system collecting data from gateways and displaying it on customer dashboards. Sensors are analog or digital hardware devices that produce measurements and send them to the gateway. Actuators are hardware devices that can receive commands from the gateway, like for instance, taking a picture, activating a buzzer or flicking a switch. The gateway is a hardware device which relays measurements to the cloud (via a 3G or WiFi network) and manages the actuators in the residency. The cloud is a software system which communicates with the gateways and provides services for the customers.
The system documentation (about 30 pages) includes (1) the description of the problem domain with scenarios, (2) the requirements of the system including non-functional requirements and (3) a hierarchically decomposed architecture specified with UML (e.g, deployment diagram). The complete description of the system has been used in previous studies [210] and is open to the public. The participants were tasked to read the preparatory material (including the system documentation) before the experiment took place. Email reminders were sent out before-hand. The documentation of the system does not include security requirements, as the purpose of this exercise is to derive those from the detected design flaws.

Participants. The participants of this study are three master-level students and three doctoral candidates in computer science and engineering disciplines. We excluded one report handed in by a master student as the task was not taken seriously. All the participants had experience with modeling, basic security concepts, and UML notations used in the system documentation. The participants in the second experiment were doctoral candidates in software engineering with a master degree in computer science and engineering. They have worked with modeling before, and were familiar with basic security concepts and the notations used for describing the experimental object. We have made sure that the participants were able to complete the task before starting the experiment. The participants performed the task individually.

Task. In essence, the participants were tasked to go through each entry in the catalog, use the guidelines to detect flaws in the HomeSys design, and document the identified flaws. The guidelines prescribe detecting a subset of elements in the architecture for inspection, therefore the participants had to also indicate the location of the identified flaw. For each architectural element under inspection, the guidelines provide a list of closed questions. A negative answer indicates the existence of a design flaw for that element. In case of insufficient information in the documentation, the participants were instructed to report a design flaw, anyway. If they thought a particular entry was not applicable in the system under examination due to some restrictions or regulations in the domain, they were instructed to mark that entry as not applicable. Finally, the participants handed in all the printed documents, including the filled-in form documenting the detected flaws.

Execution. Before the execution of the first experiment, four participants were selected due to having successfully completed a master level course on advanced software architecture. Therefore, they were familiar with basic concepts of software architecture design, and have studied analyzing architectures for security in the context of this course. On the day of the first experiment, the participants were gathered in a classroom. They were given printed copies of (i) a one-page task description (ii) the HomeSys documentation, (iii) the catalog of security design flaws as described in Section 7.2, and (iv) a form for documenting the identified flaws. Regular breaks were allowed during which the experimenters made sure the participants did not compare solutions.
Before the execution of the second experiment, preparatory reading was handed out via electronic mail (2 weeks in advance). The preparatory reading included the same documents used in the first experiment. Regular email reminders were sent until the last day before the execution. On the day of the experiment, the participants met the experimenter for an individual session on faculty premises. They was given printed copies of all the documents (i-iv). The first author explained their task again and briefly described the printed documents. Only procedural questions were answered during the experiment. Due to the complexity of the task, no strict time limit was enforced, short breaks were permitted, and accounted for.

Measures. We adopt the same ground truth and measures of precision, recall, and productivity in both experiments. The ground truth was re-assessed after the first experiment to ensure its correctness. It consists of 47 security design flaws. Conventionally, precision ($TP/(TP+FP)$) is measured as a ratio between the true positives (i.e., correctly identified flaws) and all identified flaws (including the false flaws). A true positive (TP) is a correctly identified security design flaw. This entails that the documented flaw exists also in the ground truth and is identified at the same location of the architecture. A false positive (FP) is an incorrectly identified and documented flaw that does not exist in the ground truth. Recall ($TP/(TP+FN)$) is measured as a ratio between the true positives and all correctly identified flaws (including the overlooked flaws). A false negative (FN) is a design flaw that exists in the ground truth but has not been documented. Productivity (TP/h) is measured as the amount of correctly identified flaws per hour. We measured the time it took for the participants to complete the task and subtract the time that was lost during the breaks. Participants sometimes noted that a flaw was not applicable. All participants marked the correct flaws as not applicable, therefore we do not assess flaws that were not applicable as TP, FP, or FN.

We include an additional measure for keeping track of the guidelines the participants struggled with. We flag the TP, FP, FN whenever the participant expressed that there was insufficient information (II) to determine the existence of a flaw. If a correctly identified flaw (TP) is flagged with II this means that the participant has identified the correct flaw in the correct location in the

Table 7.2: Results from both experiments (accumulated results are in bold).

		Design Flaws			Measure		
	Participant ID	TP	FP	FN	P [%]	R [%]	Prod [TP/h]
MSc	1	23	2	24	92	48.9	9.5
	2	26	3	21	89.7	55.3	9.4
	3	19	4	28	82.6	40.4	6
	Avg	22.7	3	24.3	88.1	48.2	8.3
PhD	4	24	1	23	96	51.1	8.2
	5	20	1	27	95.2	42.6	8.5
	6	30	0	17	100	63.8	17.7
	Avg	24.7	0.7	22.3	97.1	52.5	14.8
	Total Avg	**23.7**	**1.8**	**23.3**	**92.6**	**50.4**	**11.5**

architecture, has documented this in the hand-in, but has also expressed doubt due to missing information in the documentation. If an overlooked flaw (FN) is flagged with II this means that the participant failed to identify the correct flaw but also expressed doubt due to missing information.

7.4 Results

Table 7.2 summarizes the results of both experiments. First, we have calculated the average precision, recall, and productivity for both experiments separately. On average, participants 4-6 from Table 7.2 performed slightly better compared to participants 1-3 (avg 97.1% precision vs. 88.1%, avg 52.5% recall vs. 48.2%, 14.8 TP/h productivity vs. 8.3 TP/h). Yet, these differences are small and not significant. The performance differences between experiments can be explained by the different level of education. Henceforth we refer to the accumulated TP, FP, FN, measures of precision, recall, and productivity of the inspection guidelines.

Precision, recall, and productivity. On average, the participants identified about half (23.7/47) of the security design flaws correctly. Yet, on average only about 2 reported security design flaws were incorrect (FP). Therefore, the average *precision* is quite high (**92.6 %**). The low number of FPs may indicate that the inspection guidelines were not misleading the participants towards a false design flaw discovery. On the other hand, about half of the flaws were overlooked. On average, the *recall* is measured at **50.4 %**. This result is not surprising. Similar measurement of precision and recall have been reported in related empirical studies investigating manual knowledge-based threat analysis techniques, i.e., STRIDE [10,210]. In general, high precision and low recall may be a common trait for techniques that manually analyze software architectures. The average *productivity* of the approach is **11.5** correct threats per hour. This result is more surprising, as the related literature reports a much lower number of correctly identified threats per hour (1.8 TP/h in [10] and about 4 TP/h in [210]). This can be explained by the different goals of threat analysis techniques vs detection of design flaws. Threat analysis techniques help to systematically identify security threats on the level of software architecture. The security threats are considered correct only when a realistic attack scenario is found. Finding a realistic scenario requires thinking about possible attack paths

Table 7.3: Flaws flagged with insufficient information (II) (problematic flaws are in bold).

Flaw ID	TP (Participant ID)	FN (Participant ID)	Σ
2	4 (6)	0	4
3	1 (6)	0	1
4	**7 (1,2,3,6)**	**3 (3,5)**	**10**
7	0	4 (5)	4
9	8 (1,2)	0	8
12	**3 (1,2)**	**11 (1,2,5,6)**	**14**
13	2 (6)	2 (6)	4
14	2 (5,6)	0	2
15	**1 (6)**	**15 (1,2,5,6)**	**16**
17	2 (5)	2 (6)	4
18	**0**	**2 (3,4)**	**2**
19	2 (6)	0	2
Total	32	39	71

and how to break into the system. This may be cognitively more demanding than answering a set of closed questions about the architectural design.

Insufficient Information. In the second experiment, we measure which guidelines posed problems to our participants. Particularly, we flag correctly identified and overlooked flaws with *Insufficient Information (II)*. To this aim, we have re-assessed the data collected in both experiments. A FN was flagged when the participant reported the flaw, but never specified the location due to missing information. A FN was also flagged when the participant did not report the flaw, but made notes about missing information next to the check-list in the catalog. Incorrectly identified flaws (FP) were never flagged. The participants never expressed doubt about missing information when making a mistake.

Table 7.3 shows the flagged TP, FP, which flaw they relate to, and how many participants expressed the same doubts. We have recorded that **25.2%** (71/282) of all reported TP and FN from Table 7.2 ($\Sigma TP + \Sigma FN = 282$) were flagged with II. Visibly, security design flaws 4, 12, 15, and 18 seem to be problematic. These design flaws were often subject to missing information, for more than one participant. This may indicate that some guidelines assume the availability of detailed information about the system (information that is rarely available in the design phase). In addition, we have gathered participants feedback in the exit questionnaire. We asked the participants about what they did not like about the approach. Some participants referred to missing information. For instance:

> *"I felt that the approach and the architecture were too detached, and that I* ***needed much more information*** *that what was provided in the architecture to complete the analysis properly."*

This confirms that the participants indeed had problems with some of the guidelines. In particular we have identified three kind of problems, and suggest improvements in what follows.

7.5 Improving the inspection guidelines

In what follows we describe the problems we encountered with the proposed guidelines, and provide suggestions for improvement. *Guidelines generalization.*

Design Flaw 4: Insufficient Session Management

Description Not managing a session properly throughout its life-cycle can leave the system vulnerable to session hijacking attacks. Session management involves creation (the session should be established through a secure channel and session identifier should be encrypted), the time frame the session is active (an attacker might attempt to reuse the session ID to gain access) and its destruction/invalidation (proper session invalidation should take place when the user logs out or session timeout. Not terminating sessions can also lead to resource depletion).

Detection

(i) Determine which sessions are established in the system and between which endpoints.

(ii) For each session examine:

- (1) Is the session established through a secure channel?
- (2) Is the session ID encrypted when in transit?
- (3) Is the session ID hard to guess?
- (4) Is the use of session ID as a parameter in URLs prevented?
- (5) Is the session ID validated on server side?
- (6) Are secure cookies used?
- (7) Is the session ID tied to other user properties like IP, SSL session ID?
- (8) Can the same session be accessed simultaneously from two endpoints? Should it?
- (9) Is session timeout set? Is it the minimum possible value?
- (10) Is the session invalidated on logout?
- (11) Is the session ID renewed in the event of privilege change?
- (12) Is there a mechanism to monitor the creation/destruction and attempts to connect to a session?
- (13) Is the user required to re-authenticate after a period of inactivity?

Listing 7.2: Textual description of the insufficient session management design flaw.

We analyze the problematic guidelines related to design flaws 4 and 15. Both guidelines are **not general enough** to be useful for a design-level analysis. We have observed this trend in other guidelines as well (namely, 12, 18).

Listing 7.2 shows the guidelines related to security design flaw 4. Overall, participants still managed to correctly identify this flaw despite having doubts due to missing information (c.f., Table 7.3). They were able to do so as some question posed for detection are in fact very useful (e.g., *Is the user required to re-authenticate after a period of time?*) and can be answered at design time. One possible explanation for our participants expressing doubts is that there were too many technical questions posed for detection. For instance, *Is the session ID tied to other properties like IP, SSL,?* The properties of the session IDs are technology-dependent. The choice of technology is decided in later stages of the development life-cycle [149].

Listing 7.3 shows the guidelines related to security design flaw 15. Compared to the previous flaw, these guidelines are much shorter. All three questions posed for flaw detection are very technical. For instance, *Is the reuse of packets prevented (Replay attacks)?* The participants did not know what replay attacks are, or how to counter them. This may have caused the participants to simply ignore this flaw without even finding the possible locations, and making a note

Design Flaw 15: Insecure Data Exposure

Description Data is not transferred in a secure way. For example a web application uses the HTTP instead of HTTPS. This leaves the channel vulnerable to eavesdropping, Man In The Middle (MITM) attacks etc.

Detection

(i) Locate the valuable information in the model.

(ii) Track them through the architecture to determine where and how they are transferred.

(iii) At each step examine the following:

   (1) Is the reuse of packets prevented (Replay attacks)?

   (2) Is there any form of timestamping, message sequencing or checksum in the exchanged packages?

   (3) Is the traffic over an encrypted channel (SSL/TLS)?

Listing 7.3: Textual description of the insecure data exposure design flaw.

Design Flaw 4.1: Insufficient Session Management

Description Not managing a session properly throughout its life-cycle can leave the system vulnerable to session hijacking attacks. Session management involves the creation, the time frame the session is active, and its destruction. The attacker may attempt to disrupt or manipulate these processes for her gain.

Detection

(i) Determine which sessions are established in the system and between which endpoints.

(ii) For each session examine:

   (1) Is the session established through a secure channel?

   (2) Is the session ID hard to guess?

   (3) Is the session ID protected when in transit (e.g., encrypted)?

   (4) Is there a process for validating the session ID on server side?

   (5) Is the session destructed (or invalidated) on logout?

   (6) Is the session ID renewed in the event of privilege change?

   (7) Is the user required to re-authenticate after a period of inactivity?

Listing 7.4: Textual description of the improved insufficient session management design flaw.

about missing information (resulting in a flagged FN).

Listing 7.4 introduces a few simple improvements. First, the description of the design flaw is shortened and redundant questions removed. For example, session timeout directly relates to the last posed question *Is the user required to re-authenticate after a period of inactivity?* Second, the questions posed for detection that were too specific were removed. For instance, the usage of secure cooking is technology-dependent.

This kind of improvement could help to relate the guidelines to a design-level description, making them easier to apply. In the exit questionnaire we also ask the participants for their suggestions for improvement. Generally, the participants suggested similar improvements. One participant even suggested to have several versions of the same guideline but on different level of abstraction: *"Maybe divide the 19 points [Security Design Flaws] into classes based on*

Table 7.4: Suggestion for re-organization of the proposed guidelines.

Old		New	
ID	Name	ID	Name
1	Missing authentication	1	Missing authentication
2	Authentication bypass		
3	Relying on single factor auth	2	Weak authentication
5	Downgrade authentication		
9	No Re-authentication		
14	Insuff. credentials management		
6	Insuff. crypto key management	3	Missing/weak encryption
16	Use of custom/weak encryption		
7	Missing authorization	4	Missing authorization
8	Missing access control		
11	No context when authorizing	5	Weak authorization
12	Not revoking authorization		
13	Insecure data storage	6	Leaking important data
15	Insecure data exposure		
10	Unmonitored execution	7	Uncontrolled resources
19	Uncontrolled resource consumption		
17	Not validating input/data	-	
18	Insufficient auditing	8	Insufficient auditing
4	Insufficient session management	9	Insufficient session management

> *what* ***level of design*** *is provided.”*

Overlapping guidelines. Most participants agreed that the approach takes too much time. We found that many guidelines are **overlapping** and could be compressed and re-organized. In addition, as per comparing guidelines in Listing 7.2 and Listing 7.3, the guidelines are not well balanced in length and complexity. For instance, the guidelines in Listing 7.3 try to help detecting where data can be leaked in transit. Yet, the proposed catalog contains another set of guidelines to detect the design flaw for insecure data storage (Flaw 13). Detecting both flaws requires to first identify valuable information and track it through the architecture. Therefore, merging them and removing redundant questions would help speed up the manual detection. The exit questionnaire revealed that the participants generally agree with this notion. For instance, two participants criticized the categorization of guidelines:

> *“Maybe I would not categorize them [Security Design Flaws 1-19] so much. Having 5-10 [questions] under the same category* ***does not always represents the category****.”*

> *“Try to group the items of the list (e.g., there are many references to* ***encryption scattered through all the guidelines****).”*

Table 7.4 suggests a reorganization of the proposed guidelines. Implementing such a restructuring (incl. the removal of redundant guidelines) could result in half the original size of the catalog (9 vs 19). A closer look into the guidelines showed that many security design flaws were very similar. For instance, the guidelines for detecting missing authorization and missing access control were asking the user for the same kind of information twice. We also suggest to gather all encryption-related guidelines and group them into one comprehensive guideline (*Missing/weak encryption*). We suggest the removal of one security design flaw altogether, namely *Not validating input/data* as missing field validation is a bad programming practice, rather than a design flaw.

Tedious documentation. Finally, in the exit-questionnaire, participants have expressed a dislike for navigating through documents and documenting the flaws. For instance, when asked if the approach takes too much time, one participant responded:

> *"Yes, since you have to go through all the different documents."*

Another participant responded:

> *"Yes, some of the questions covered many parts of the system which lead to a journey on finding information."*

The participants had to answer not only the closed questions next to the guidelines, but also fill-in a form with a list of identified flaws and their description. This resulted in a waste of time and could have only decreased the level of concentration.

First, for future studies, we suggest to minimize the amount of different documents handed out to participants. Second, we suggest minor changes in the format of the guidelines. Along-side each closed question, there should be (i) an obligatory field to specify the location in the architecture, (ii) optional filed for marking a flaw as not applicable, (iii) optional field for notes. Finally, we suggest to remove the additional form with hand-written descriptions of identified design flaws. Having these changes in place would mean the users only work with two documents: one describing the architecture, and one with the guidelines and identified flaws. Automation and tool support would also help reduce this problem.

7.6 Related Work

This section positions the paper in the context of existing literature on catalogs of design flaws, vulnerabilities, architectural bad smells, anti-patterns, and knowledge-based threat analysis techniques.

Catalogs of security design flaws. Da Silva and Cecilia [240] have compiled a catalog of common architectural weaknesses (Common Architectural Weakness Enumeration, CAWE). The authors identify and categorize common types of vulnerabilities rooted in software architecture design and provide mitigations to address such vulnerabilities. Da Silva and Cecilia [240] also analyze the vulnerabilities of four real systems to discover their cause and find that up to 35% vulnerabilities were rooted in the architectural design. Similar to our work, the authors investigate which security patterns are likely to have associated vulnerabilities. However, the proposed catalog is not evaluated with an empirical study. As a result of initiatives launched by IEEE Computer Society, Arce et al. [241] compiled a list of top 10 security design flaws and discuss how to avoid them. The practical examples that showcase the flaws are very useful for understanding the impacts of each flaw. Some catalog entires by Hosseini and Malamas [239] are aligned to top 10 security design flaws. For instance, Flaw 6 (insufficient cryptographic keys management) in [239] relates to the "use cryptography correctly" flaw. Yet, the purpose of the top 10 security design flaws was to raise awareness among software architects about the most common issues that have been the leading cause for security breaches in practice. The purpose of this paper was to re-evaluate detection guidelines and provide improvement suggestions for automating the detection.

Vulnerability databases. Common Vulnerabilities and Exposures (CVE) (launched in 1999) is the largest and most updated vulnerability database. Maintained by the MITRE corporation, it provides a publicly available list of most common security vulnerabilities with unique identifiers. Common Weakness enumeration (CWE) is a community-developed list of common software security weaknesses. CWE aids developers and security practitioners since it serves as a common language for describing security weaknesses in architecture and implementation. It also provides different mitigation and prevention techniques that could be used to eliminate weaknesses.

Architectural bad smells and anti-patterns. The literature on architectural bad smells [234, 236] and anti-patterns [235, 237, 238] collides with our work for what concerns the ambition to find and remove architectural issues that negatively impact system life-cycle properties (extensibility, maintainability, testability, etc.). In contrast, our work investigates security design flaws, that is architectural design decisions that negatively impact the system security. Mo et al. [237] have recently developed an automated detection of 6 architectural anti-patterns and study their impact on error and change-proneness of the related files. The authors analyze 100 industrial software projects with respect to the project structural information and revision history. They find that there are only a few distinct types of anti-patterns that occur in all the projects, where *Unstable Interface* and *Crossing* were by far the largest culprits of error and change-proneness. In contrast to our paper, the authors detect the anti-patterns based on existing implementation. Our work is focused on detecting security design flaws at the level of architectural models. Bouhours et al. [235] introduce a catalog of 'spoiled patterns' and use it to automatically detect their manifestation in software architecture models. They develop a plug-in for the Neptune environment (UML, XML) which supports the instantiation of spoiled patterns and suggest model transformations for re-factoring the design models. Nafees et al. [238] propose a new template for detecting architectural anti-patterns and a catalog of 12 Vulnerability Anti-Patterns (VAP) entries. The authors also provide some examples of the proposed VAP entries. Yet, the catalog has not yet been evaluated empirically. Taibi and Lenarduzzi [236] have conducted interviews with 72 developers to collect a catalog of 11 microservice-specific bad smells. Besides the 11 smells, the authors put forward the importance of carefully analyzing the connections between microservices, especially the connections leading to private data and shared libraries. Similar to this work, the purpose of compiling such a catalog is to help practitioners in the detection of bad architectural decisions. Garcia et al. [234] describe 4 architectural bad smells identified through an in-depth analysis of two industrial systems. The architectural smells are described in detail and are accompanied with UML component diagrams, and a discussion of their impact on quality.

Knowledge-based Architectural Threat Analysis. Architectural threat analysis consists of techniques and methods that are used for systematically analyzing the attacker's profile vis-a-vis assets of value to organizations. Such techniques are often performed on models representing the software architecture of a system. The purpose of analyzing security threats at this stage is to ultimately identify security holes and plan for necessary security solutions. Therefore, we consider existing literature that makes use of knowledge base (threat catalogs, vulnerability data bases, etc.) [9, 11, 44, 45, 124] to perform such analysis as related work. We refer the interested reader to a systematic literature review [208] for a more detailed list of knowledge-based threat analysis techniques.

7.7 Threats to Validity

With respect to *internal validity threats*, we consider the threat of using graduate and doctoral students as participants. Using students during empirical studies has been criticized in the past, as their background knowledge is not that of industrial practitioners. However, studies have shown [187, 189] that the differences between the performance of professionals and graduate students are often limited. To counter this threat we have made sure that the selected participants had sufficient background education to complete the tasks. We also consider the threat of an unrepresentative sample size (in total six participants). In addition, the suggestions for improvement have not been empirically evaluated.

With respect to *external threats* to validity, we consider the threat to generalizability of results. This study is conducted only on one experimental subject, thus the results can not be generalized to other domains. To counter this threat, we plan to conduct more studies, including security experts, and different experimental subjects.

7.8 Conclusion

This paper proposes a catalog of security design flaws accompanied by inspection guidelines for their detection. To evaluate our approach, we present two empirical studies with master students and doctoral candidates. We conduct two experiments investigating the performance of manually applying the inspection guidelines in the context of analyzing a home monitoring system. We also account for the shortcomings of the inspection guidelines and make suggestions for their improvement with respect to the generalization of guidelines, catalog re-organization, and format of documentation. Our results show a relatively high precision (92.6) and productivity (11.5 TP/h). On the other hand, we found that about half of the security design flaws go unnoticed (average recall is 50.4%). We introduce an additional measure to investigate which guidelines posed problems to our participants. We identify three type of problems the participants encountered. First, some guidelines were not general enough to be useful for detecting security design flaws. Second, several closed questions used for detection were overlapping, potentially resulting in loss of time. Lastly, participants have expressed a dislike for the format of documenting the flaws. We suggest simple improvements of the guidelines to support future automation.

Chapter 8

Paper G

This chapter is based on

Automating the Early Detection of Security Design Flaws,

written by

K. Tuma, L. Sion, R. Scandariato, and K. Yskout,

published in

Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), 2020.

Abstract

Security by design is a key principle for realizing secure software systems and it is advised to hunt for security flaws from the very early stages of development. At design-time, security analysis is often performed manually by means of either threat modeling or expert-based design inspections. However, when leveraging the wide range of established knowledge bases on security design flaws (e.g., CWE, CAWE), these manual assessments become too time consuming, error-prone, and infeasible in the context of contemporary development practices with frequent iterations. This paper focuses on design inspection and explores the potential for automating the application of inspection rules to speed up the security analysis.

The contributions of this paper are: (i) the creation of a publicly available data set consisting of 26 design models annotated with security flaws, (ii) an automated approach for following inspection guidelines using model query patterns, and (iii) an empirical comparison of the results from this automated approach with those from manual inspection. Even though our results show that a complete automation of the security design flaw detection is hard to achieve, we find that some flaws (e.g., insecure data exposure) are more amenable to automation. Compared to manual analysis techniques, our results are encouraging and suggest that the automated technique could guide security analysts towards a more complete inspection of the software design, especially for large models.

8.1 Introduction

In the current software development culture, agility and speed are paramount. However, software quality in general, and security in particular, cannot be sacrificed in lieu of a faster pace of development. This is where the "shift left" concept comes into place. Namely, software validation and verification should be applied as early as possible in each agile iteration, including the analysis of the design. Indeed, recent work has highlighted that a large share of vulnerabilities disclosed in industrial control systems had their root cause in the design [242].

At design level, mainstream analysis and validation techniques, like threat analysis and design inspections [9, 11, 127], are heavily based on the use of experts performing manual tasks. Therefore, they do not fit well in the agile paradigm of continuous integration and continuous development [33]. More automation of design-level security techniques is necessary, and the research community has responded to this challenge [43–45, 92]. This paper continues on this research path and explores the automation of design-level security inspection guidelines, which has not been attempted before.

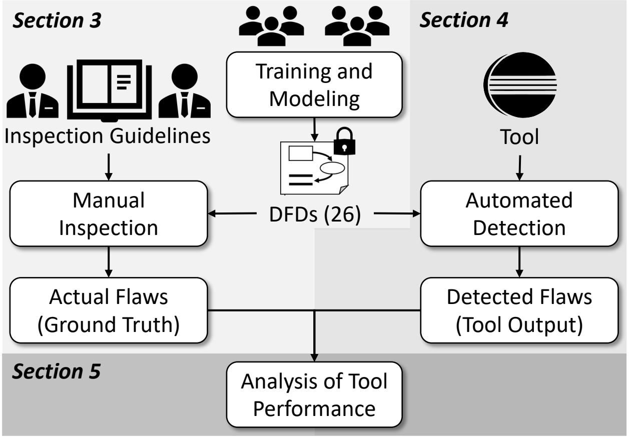


Figure 8.1: Research activities and paper structure

In particular, we select a subset of 5 inspection guidelines from the catalog of Tuma et al. [243] (see Section 8.2.1) and define **a technique to perform the model inspection automatically**. We assume that a software system is modeled as a Data Flow Diagram. This choice of this model type is justified by the fact that DFDs are widely used in the industry for security analysis purposes. For instance, DFDs are central to threat analysis and are, therefore, already available in companies that have a secure software process in place [91]. In addition, a recent case study [15] involving four companies shows that DFDs are used for threat analysis in agile organizations. We use an enriched version of DFDs, which are annotated with additional security information (see Section 8.2.2). As shown in the right-hand side of Figure 8.1, in Section 8.4 we describe how the inspection guidelines have been (i) represented as model query patterns by means of VIATRA and (ii) implemented in a prototype tool as an Eclipse plugin. A match of a model query pattern executed by the tool would correspond to the presence of a security design flaw in the analyzed DFD.

Table 8.1: Security design flaws from the catalog proposed by Tuma et al. [243] (flaws in bold are the focus of this work)

Flaw number and name		Description
1	Missing authentication	An absence of an authentication mechanism in the system.
2	**Authentication bypass**	There exists an entry point with authentication mechanism that can be bypassed.
3	Relying on single factor authentication	The authentication mechanisms rely on the use of passwords.
4	Insufficient session management	Sessions are not managed securely throughout their life cycle.
5	Downgrade authentication	Possibility to authenticate with a weaker (or obsolete) authentication mechanism.
6	**Insufficient crypto key management**	Keys are not managed securely throughout their life cycle.
7	Missing authorization	An absence of an authorization mechanism in the system.
8	Missing access control	An absence of access control in the system.
9	No re-authentication	An absence of re-authentication during critical operations.
10	Unmonitored execution	Uncontrolled resource consumption due to interactions with external entities.
11	No context when authorizing	An absence of conditional checks for access control.
12	Not revoking authorization	An absence of a process for revoking user access.
13	**Insecure data storage**	Storage of sensitive data is in clear or access control mechanisms are weak.
14	Insufficient credentials management	Credentials are not managed securely throughout their life cycle.
15	**Insecure data exposure**	Sensitive data is transported in clear.
16	Use of custom/weak encryption	Generating small keys, using obsolete encryption schemes.
17	Not validating input/data	Absence of validation checks when receiving data from external entities.
18	**Insufficient auditing**	Access to critical resources or operations is not logged.
19	Uncontrolled resource consumption	Uncontrolled resource consumption of internal components.

To evaluate our technique, we need a ground truth, i.e., design models (DFDs) that are labeled with information concerning the security design flaws that are present in each model. Such a data set did not exist and, in general, the lack of validation data has been a recurring challenge in our field of research. As shown in left-hand side Figure 8.1, in Section 8.3 we describe how we have created **a curated data set of 26 security-oriented DFDs** by enrolling 13 modelers who have worked on 4 different software systems, under controlled conditions and with the prescriptions of empirical studies. Additionally, we have employed 2 security experts (co-authors of this paper) to assess the models. The experts have manually applied the 5 inspection guidelines under investigation in this work and have identified the design flaws in all models. The assessment has been performed in an unbiased way, i.e., without any prior knowledge of how the automated technique works. Further, the experts have worked independently and have checked each other's work to a large extent, which provides assurance about the quality of the resulting data set. The data set is now publicly available to the research community and has been used in Section 8.5 in order to validate the automated technique we propose.

Our results (discussed in Section 8.6) show that three inspection guidelines have the promising potential of being amenable to automation. Clearly, these results are valid within the confines of the threats to validity presented in Section 8.7.

8.2 Background

This section provides some background on design flaws, the catalog of inspection guidelines, the Data Flow Diagram (DFD) [244] representation, and its security extensions.

8.2.1 Design Flaws and Inspection Guidelines

We refer to a security design flaw as a weakness in the high-level design of a system (e.g., software architecture), which exposes the system to security threats. Flaws may lead to code defects [245]. This paper relies on a catalog of security design flaws proposed by Tuma et al. [243]. As shown in Table 8.1, the catalog consists of 19 common security design flaws concerning authentication, access control, authorization, availability of resources, integrity, and confidentiality of data. It was compiled by means of a systematic analysis of existing vulnerability database entries from several sources (CVE [87], CWE [88], OWASP [246], and SANS [247]). This study focuses on **five** security design flaws in particular, marked in bold in Table 8.1.

As shown by the example in Listing 8.1, each design flaw specifies an inspection guideline. The guidelines were developed for manually determining the presence of this security design flaw in a software architecture. Each guideline leads the analyst to the identification of certain locations in the model where the flaw could be present. At those locations, the analyst has to evaluate some criteria (rules) in the form of yes/no questions. A '*no*' answer means that a flaw is present. To help the analyst, the criteria sometimes refer to certain security solutions. But, they do not account for all existing security solutions protecting a data property. For instance, the criterion 'Is there any form of

Security Design Flaw 15: Insecure Data Exposure

Flaw description Data is not transferred in a secure way. For example a web application uses the HTTP instead of HTTPS. This leaves the channel vulnerable to eavesdropping, Man In The Middle (MITM) attacks etc.

Inspection guideline to detect this flaw

(i) Locate the valuable information in the model.

(ii) Track them through the architecture to determine where and how they are transferred.

(iii) At each step examine the following:

  (1) Is the reuse of packets prevented (Replay attacks)?

  (2) Is there any form of time-stamping, message sequencing or checksum in the exchanged packages?

  (3) Is the data transferred over an encrypted channel (SSL/TLS)?

Listing 8.1: Inspection guideline for flaw 15 [243]

time-stamping, message sequencing or checksum in the exchanged packages?' does not require cryptographic hashing (as opposed to a simple checksum) to be satisfied. In addition, TLS provides message authentication in addition to encryption. Manually exploring design models in such a way is effort intensive and prone to errors. Therefore, automating this assessment activity is desirable, especially in the context of frequent design iterations where redoing such an assessment is prohibitively expensive.

8.2.2 Data Flow Diagram and Security Extensions

In this work, we automate the inspection of DFDs, which are already extensively used in security threat modeling [9, 190, 208]. The DFD notation is used to graphically represent a system architecture. It highlights the flows of data, showing how the information enters, traverses, and leaves the system. Figure 8.2 depicts a high-level DFD for a social network application. The diagram shows how private users and ad companies (both *external entities*) interact with the system, which is modeled as a set of *processes* for authentication (Authenticate), core business logic (Service Provider) and access to the persistence layer (DB access provider). Data is persisted in two *data stores*: key material is in the Key Storage, while user data is in the Social Network Database.

The regular DFD notation is limited in denoting security-related information, making it hard for practitioners to reason about security at design time [45]. To this aim, the DFD notation has been extended in the literature with security properties [248] and security solutions [249]. As shown in Figure 8.2, the modeler can specify the type of information that is passed around (e.g., sensitive, encrypted data or key material). Furthermore, the modeler can represent the use of security mechanisms:

(i) secure pipes (optionally with client authentication) to protect the confidentiality and integrity of data transmitted over data flows,

(ii) encrypting data in a data store,

(iii) key management solutions (creation, replacement, and destruction),

(iv) secure log of access to data stores, and

(v) authentication.

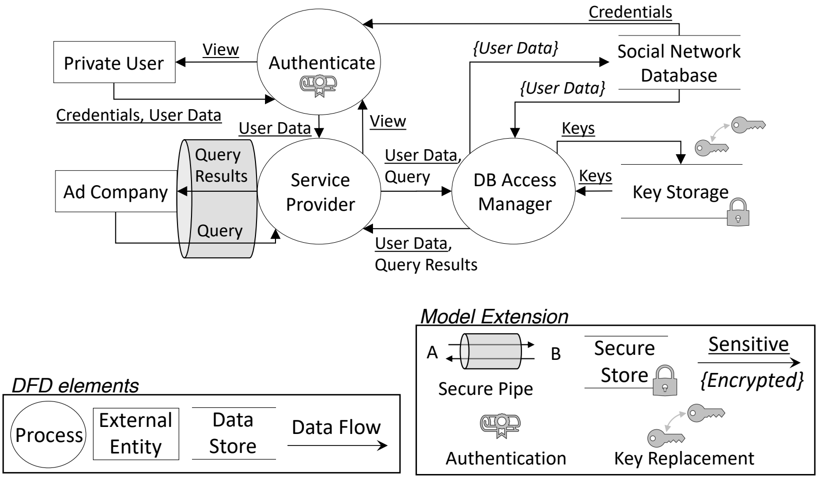


Figure 8.2: A Data Flow Diagram (DFD) of a social network application (with security extensions)

8.3 A Curated Data Set of Design Models and Their Security Flaws

The research field of secure design is plagued by the lack of publicly available ‘case studies’ that could be used to validate new techniques. In order to overcome this shortcoming, we have set up a series of workshops where we asked 13 participants to model a variety of systems using a DFD-like notation in a design tool. The resulting models have been analyzed for security flaws by 2 expert assessors. The workshops have been carried out with scientific rigor in the form of an empirical study (i.e., under controlled conditions) in order to guarantee the quality of the outcome. The outcome of this study is two-fold: (i) the creation of a data set of 26 DFD design models enriched with security solutions and data types, and (ii) for each model, a report of the existing design flaws (for 5 flaws, shown in bold in Table 8.1) and their locations. All the material is publicly available on this paper’s companion website [250].

8.3.1 Study Design

Participants and training. The volunteering participants of this study are 13 academic researchers. All participants finished a higher-level degree in the field of computer science and software engineering and are employed at two universities in two different countries. About half of participants (8/13, herein GROUP A) have a strong background in software design, requirements engineering and modeling. Yet, they are less experienced in software security. The other half (5/13, herein GROUP B) have a deeper understanding of security-related topics, including secure software design and formal methods for security. All the participants received a training session of 1 hour. This **training** session included an introduction to the DFD modeling notation, the extensions to the DFD notation used in this work, a brush-up on concepts related to software security, and a demonstration of the design tool they will use. The same training material has been used at both universities.

Modeled systems. We prepared a brief description (about one page) for 4 different systems. Each description included an explanation of the system functionality and a list of security requirements.

DriveSafe — A smartphone application for achieving safety on the roads collaboratively by continuously updating nearby drivers on current road safety conditions.

BeSocial — A proximity-based collaborative messaging smartphone application to support creating and maintaining virtual chat rooms for nearby users.

PhotoFriends — A media sharing smartphone application to enable users to share photos and build a network of friends.

SmartTex — A collaborative document management web service targeting members of the scientific community to support creating, editing, and compiling LaTeX documents in a collaborative way.

Model creation. In a **randomized** assignment, each participant was given the task to model two of the four systems, by using the DFD notation and its security extension. Individual participants met with the experimenters for a modeling session of about 3 hours. Each participant received a handout package including (i) printed training slides, (ii) a cheat-sheet for the model notation, (iii) a computer with the design tool, and (iv) the descriptions of the systems they had to model. The descriptions are designed in such a way that they can be easily understood in a limited amount of time. Further, the experimenters were available to answer any questions.

Before they started with the task, the participants carried out a short **warm-up** modeling exercise (15 min) to get familiar with the tool. Next, they were given the documentation of their first system. Participants were tasked to read the documentation carefully, and use the tool to create a DFD enhanced with security solutions and data types. To enhance the DFD with security solutions, they instantiated solutions from a provided catalog and bound them to concrete DFD elements. Similarly, they labeled data flows with data types according to a provided data type catalog.

During the modeling session the experimenters took notes and monitored their progress. Finally, the participants were asked to shortly explain their model. After finishing the first model, they received the documentation for the second and repeated the task.

As a result of the modeling sessions, we obtained 26 models [250]. Each DFD model is annotated with labels (e.g., sensitive data) for information assets on the data flows. Also, the models contain elements representing certain security solutions, like encryption on data flows, authentication of external entities, and so on.

Manual model inspection. Two experts manually scrutinized the created models to identify five types of design flaws (in bold in Table 8.1) by applying the inspection guidelines described in Section 8.2. As a form of calibration, the assessors independently inspected four randomly selected models (covering the four different systems) and then they compared their results in a joint session. The discussion of the disagreements resulted in the explicit formulation of common criteria for the subsequent inspections:

(i) If the participant made any mistakes in the use of notation or logical

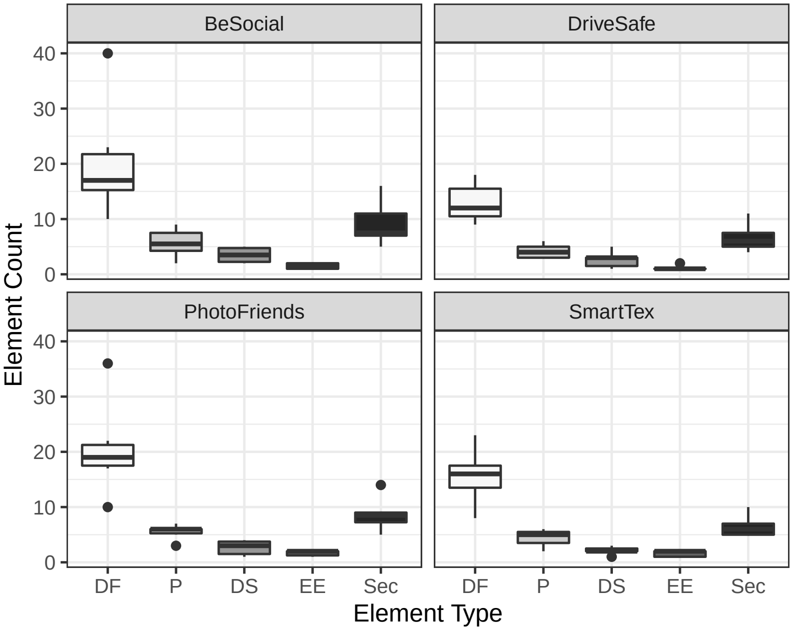


Figure 8.3: Overview of the model sizes per system

mistakes (that is, in case of minor mistakes), the experts agreed to take their intention into account.

(ii) If two inspection rules (for different design flaws guidelines) triggered a violation in the same model location, the experts agreed to report only one flaw for this location (the first time it was found). This is related to the fact that some inspection guidelines overlap, as discussed later in the paper.

(iii) They agreed to only consider assets that are mentioned in the security requirements contained in each system description, despite possible deviations in the created models.

(iv) They agreed to assess each model in its entirety, including any additional logic not required according to the documentation.

After the joint session, the experts independently inspected an equal share of the remaining models, which have been assigned randomly to the assessors (by blocking on the four systems). On average, the experts spent about 30 minutes to manually inspect a single model. In the end, they marked a total of five models as requiring further discussion. These models were handed over to the other assessor for a second inspection. The analysis reports were then compared, and any disagreements resolved.

8.3.2 The Resulting Data Set

The curated data set that emerged from this study can be found online [250]. The data set includes 26 security enriched DFDs, accompanied by expert reports of the flaws identified according to the inspection guidelines. In particular, the flaws are localized on the model and associated to a type (see Table 8.1).

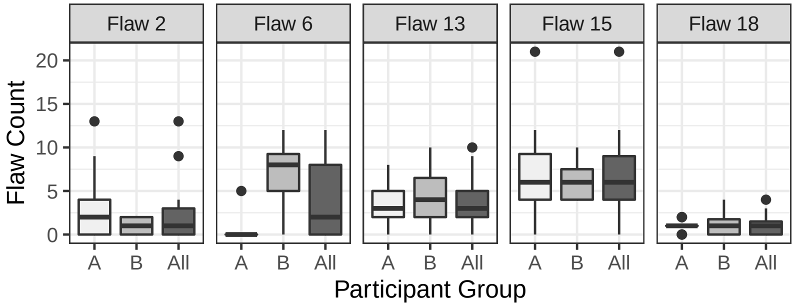


Figure 8.4: The number of flaws per participant group

Statistics about the models. Figure 8.3 depicts the average number of elements used in the models of each system. On average, a model in our data set consists of 17.1 data flow elements, 4.9 processes, 2.7 data stores, 1.5 external entities, and 7.6 security solutions. With respect to element type, the models are fairly uniform across systems. Similar distributions of DFD element types have been observed in the related work [10]. Overall, DriveSafe models are smaller compared to the rest, in particular with respect to the data flows. This is explained by the fact that DriveSafe has a very simple and unidirectional interaction model from the users' perspective.

We also investigate the differences in the created models across participant groups (i.e., the two campuses). The two groups created models of comparable size. Yet, the number of modeled data flow elements varies more within Group B (from 10 to 20). This may indicate that participants of Group A were in fact more experienced in software design modeling and created more homogeneous DFDs.

Statistics about the violations. On average 15.6 flaws are found on a single model. First, we investigated the flaws reported by the assessors to make sure that their analysis was comparable. Overall, the assessors found a similar number of design flaws of each type.

Second, we investigated the flaws for each of the four systems. Slight differences can be observed across the four systems. On average, the DriveSafe models contain the smallest number of flaws (average per model: DriveSafe: 12.5, SmartTex: 15, BeSocial: 17, PhotoFriends: 18.1). As expected, the average number of flaws seems to correlate with the model size. Systems with larger models (BeSocial and PhotoFriends) contain on average more flaws.

Figure 8.4 shows the number of security flaws in models created by Group A, Group B, and both groups together. Notably, the total number of insufficient auditing flaws (Flaw 18), regardless of the group, is much smaller compared to the other flaws. A possible explanation is that, in contrast to the other flaws, every instance of this flaw is only associated to a data store element, of which there are typically just a few in each model (see Figure 8.3). The number of flaws of type 13, 15, and 18 does not differ significantly across groups. This suggests that despite a lesser security background, Group A created similarly (in)secure models with respect to these design flaws. Yet, differences can be observed for what concerns flaw 6 (crypto key management). Often,

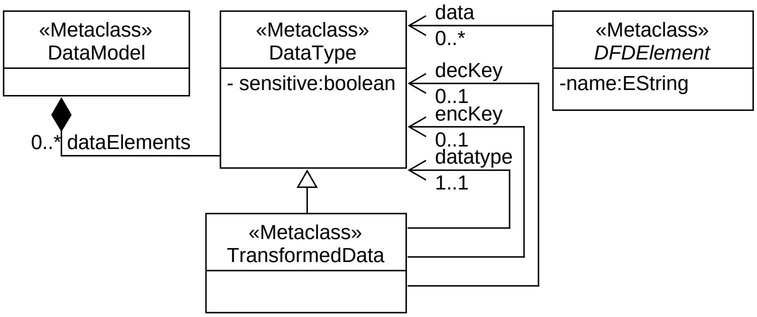


Figure 8.5: The Data Type Meta-Model

the less security-oriented group (Group A) did not model key management explicitly, hence making this inspection guideline not applicable. After the modeling sessions, Group A participants explained that they did not feel confident in their security knowledge to model cryptographic details. Also, only a few flaws of type 2 were identified on models created by Group B (average per group: Group A: 2.8, Group B: 1). Possibly, correctly modeling authentication requires a deeper understanding of security mechanisms.

8.4 Automated Detection of Flaws

This section describes the design and implementation of the automated design flaw detection. First, the required model extensions for the automated detection are presented. Next, we describe how these extensions are leveraged in the security design flaw detection. Finally, the model query patterns are discussed.

8.4.1 DFD Model Extension

The detection of security design flaws relies on the representation of two key concepts in the DFD models:

(i) security solutions, which define existing countermeasures, and
(ii) data types, which specify what type of information is being processed (especially whether it is sensitive or encrypted data).

Security solutions. The design flaw detection leverages information about existing security solutions in the model. More concretely, checking for the presence of a design flaw can incorporate the following knowledge:

(i) the presence of security solutions at correct locations in the model (e.g., the presence of authentication mechanism at the entry points),
(ii) the correct instantiation of these solutions, and
(iii) the appropriateness of the protection provided by the solution with respect to the involved sensitive data.

The existing representation of Sion et al. [249] is used to model the security solutions and capture their effects.

Data types. Data types are an essential concept in security design flaw

Table 8.2: The use of DFD model extensions for flaw detection

Extension Name	**Affected Threat Types**	**Flaws**
Secure pipe	Information disclosure, Tampering, Spoofing	2, 15
Secure pipe with client authentication	Information disclosure, Tampering, Spoofing	2, 15
Authentication	Spoofing	2
Encrypted storage	Information disclosure	13
Key creation	—	6
Key replacement	—	6
Key destruction	—	6
Secure logging	Repudiation	18
Sensitive data	—	2, 13, 15, 18
Encrypted data	Information disclosure	2, 6, 13, 15
Crypto key data	—	2, 6, 13, 15
Session data	—	2

descriptions [251] and are thus required in the models to support their detection. A concrete DFD is extended with a data model, which is a catalog of all data types that are used in the DFD. All elements in the model (i.e., processes, data flows, data stores, external entities) are linked to the relevant data types in the catalog to track how data moves across the system. Furthermore, the data model allows one to express the relationship between an encrypted piece of data and the original data, including the key (data type) used for encryption and decryption. This way, we capture the notion that 'encrypted' data is a transformation of the original (sensitive) data, such that we can still track where sensitive data is sent or stored after it has been encrypted. Figure 8.5 shows the meta-model we created for this study to represent these data types. The encrypted version of data is represented as a *TransformedData* instance.

8.4.2 Leveraging the Extensions for Detection

Table 8.2 illustrates how the DFD model extensions are used for flaw detection. The top part of this table shows the relationship between security solutions, threat types, and flaws. Rather than hard-coding the set of solutions that can impact the detection of a flaw, the detection criteria of flaws 2, 13 and 15 are expressed using a threat type (e.g., 'information disclosure'). The solutions are associated to the threat types that they prevent. Note that the actual relationship that is implemented in the detection logic is more involved, because it also needs to be verified that an instantiated solution prevents the threat at the correct model location to avoid a flaw. For the key management and logging solutions, the detection logic (for flaws 6 and 18) directly checks for their presence.

The bottom part of Table 8.2 shows the data types used in this study, and the corresponding flaws that rely on these data types in their detection logic.

8.4.3 Detecting Flaws

The security design flaws in focus (see Table 8.1) were translated to a set of criteria to enable their detection. Below, we describe how the query patterns detect instances of these flaws in concrete models.

Authentication bypass (Flaw 2). This flaw is detected by first filtering for data flows from an external entity to a process which transfer sensitive data. For each of these data flows the flaw triggers if:

(i) the data is sent without protection against information disclosure; or
(ii) there is no protection against spoofing the external entity.

Further, in the case of session data type, the flaw also triggers when there is no protection against tampering.

Insufficient crypto key management (Flaw 6). This flaw is detected by filtering for DFD elements handling a data of type key. The flaw triggers if:

(i) the key is insecurely distributed (i.e., there is no protection on the flow against information disclosure, tampering, and spoofing on data flows or processes),
(ii) the key is stored insecurely (i.e., there is no protection against information disclosure and tampering on data stores),
(iii) a solution for key creation is missing,
(iv) a solution for key replacement is missing, or
(v) a solution for key destruction is missing.

Insecure data storage (Flaw 13). This flaw is detected by filtering for data stores containing sensitive data and triggers if:

(i) the sensitive data is not encrypted (i.e., is not stored as an 'encrypted' *TransformedData*), or
(ii) there is no solution to protect against information disclosure.

Insecure data exposure (Flaw 15). This flaw is detected by filtering for data flows transferring sensitive data and triggers if:

(i) the sensitive data is not encrypted, or
(ii) there is no solution to protect against information disclosure.

Insufficient auditing (Flaw 18). This flaw is detecting by filtering for data stores containing sensitive data and triggers if there is no solution to provide secure logging of access to this data store.

To avoid biasing the results of this work, the development of the query patterns was carefully isolated from the model creation step, and the manual model inspection (see Section 8.3). First, the implemented query patterns were tested against a separate example system (not part of the data set). Second, implementing and testing the query patterns was completed before the start of participant training. Finally, the experts that performed the manual inspection were not aware of how the automated detection was implemented.

8.4.4 Implementation

This section briefly describes the implementation of the tool provided to the participants, and the detection of security design flaws.

To provide the participants with a tool environment to create the models, we have developed a modeling tool based on the Eclipse platform. The tool uses Ecore[1] to express the meta-model of DFDs, security solutions, and the data types (as discussed earlier). Furthermore, a graphical modeling editor was developed using Sirius.[2]

To detect the security design flaws, the above criteria are implemented with

[1] `https://www.eclipse.org/ecore`

[2] `https://www.eclipse.org/sirius`

```
// Pattern for Security Design Flaw 15
pattern insecureDataExposure(df : DataFlow){
    // only data flows with sensitive data
    DataFlow.data(df,data);
    find sensitiveDataType(data);

    // if sensitive info is not encrypted
    neg find dataEncrypted(data);
    // and there is no appropriate solution
    neg find flowMitigationAgainstInfoDiscl(df);
}

// Helper pattern to find sensitive data
private pattern sensitiveDataType(dataType : DataType) {
    // data type itself is sensitive
    DataType(dataType);
    DataType.sensitive(dataType,true);
} or {
    // data type is transformation of sensitive data
    TransformedData(dataType);
    TransformedData.data(dataType, data);
    find sensitiveDataType(data);
}
```

Listing 8.2: Insecure data exposure query pattern in VIATRA

VIATRA model query patterns[3] (see, for example, Listing 8.2 for the pattern for flaw 15). Every security design flaw is specified as a pattern. These patterns are typed with the meta-model elements and declaratively list the criteria for triggering the flaw. To specify more complex situations, the presence or absence of other *helper* patterns can be used. For example, Listing 8.2 shows how the detection of insecure data exposure can only match if there is sensitive data involved, which can be a data type with the 'sensitive' flag set to true, or a *TransformedData* of sensitive data (determined recursively). The automated detection in concrete user models uses the VIATRA query engine to automatically query the model and provide a list of all the discovered matches in the model, which are exported for subsequent analyses.

8.5 Performance of the Automated Inspection Technique

We have analyzed the 26 models described in Section 8.3 with the automated inspection tool described in Section 8.4.

8.5.1 Research Questions

We measure the performance of the query patterns in terms of precision and recall with respect to the ground truth, i.e., the inspection performed by expert

[3] https://www.eclipse.org/viatra

assessors. Accordingly, we pose two research questions.

*RQ1. What is the **precision** of the automated inspection guidelines (implemented as query patterns) for the detection of five security design flaws?*

We measure true positives (TPs) and false positives (FPs) to calculate the precision $P = \frac{TP}{TP+FP}$. A true positive is a flaw (guideline violation) which is detected by the tool and that matches an actual flaw reported by experts (i.e., part of the ground truth). A detected flaw matches an actual flaw when they have the same type (design flaw ID) and are attached to the same location in the diagram (model element ID). Otherwise, the flaw is considered a false positive.

A high precision would mean that the automated detection produces a low number of false alarms, which makes the technique meaningful in the context of design-level security analysis by focusing the attention of the analyst. As a term of comparison, the precision of manual design analysis techniques is known to be high (e.g., 0.81 in [10]).

*RQ2. What is the **recall** of the automated inspection guidelines (implemented as query patterns) for the detection of five security design flaws?*

To calculate the recall R, we measure false negatives (FNs), and calculate $R = \frac{TP}{TP+FN}$. A false negative is an actual flaw (i.e., part of the ground truth) which is not detected by the tool.

A high recall would mean that the automated detection is able to find most actual flaws that are present in the model, providing assurance to the analyst regarding its completeness. However, we remind that the recall of manual design analysis techniques is known to be low (e.g., 0.36 in [10] and around 0.50 in other studies [210]).

8.5.2 Results

Table 8.3 presents a summary of the performance results. As shown in the last row, the overall average precision of the automated technique is **P=0.53** and the recall is **R=0.76**. As shown in Table 8.4 these results are consistent across the four analyzed systems, i.e., there are only small variations in how the technique performs in different systems. The results for about half of the models (15/26) were inspected by the first author against the ground truth to determine if the FPs of the tool were in fact overlooked flaws by experts. Though we did not find many overlooked flaws in the ground truth, a more systematic quality evaluation would be beneficial for the data set.

> The *first take home message* is that, not surprisingly, it is very hard to attain good performance (precision and recall) when automating the inspection rules. Compared to manual threat analysis techniques (e.g., STRIDE), the precision of our automation is too low to replace expert analysis. However, the higher value of recall is somewhat encouraging, as the automated technique could be used to present an expert with a list of potential issues to sieve through.
> The *second take home message* is that some rules seem to be more promising than others as being amenable to automation. Indeed, the precision and recall differ significantly across the query patterns implementing the 5 inspection guidelines, as shown in Table 8.3.

In the rest of this section we analyze the reasons for false positives and false negatives in the detection of each design flaw. We start from the query patterns

Table 8.3: Precision (P) and recall (R) of the query patterns

Security Design Flaws	TP	FP	FN	P	R
Flaw 2: Authentication bypass	28	58	29	0.33	0.49
Flaw 6: Insufficient key management	56	36	4	0.61	**0.93**
Flaw 13: Insecure data storage	76	16	31	**0.83**	0.71
Flaw 15: Insecure data exposure	166	162	24	0.51	**0.87**
Flaw 18: Insufficient auditing	8	28	17	0.22	0.32
Total	334	300	105	**0.53**	**0.76**

with a lower precision and recall (i.e., flaws 18 and 2—in order of increasing performance) and continue with the query patterns with a slightly better precision and recall (i.e., flaws 15, 16, and 13—in order of increasing performance).

Insufficient auditing (Flaw 18) achieved the worse precision (0.22) and recall (0.32). The inspection guidelines for this flaw dictate an analysis of logging mechanisms for critical resources and operations. One possible explanation for the high number of FPs (28 compared to 8 TPs) is that the participants chose to model assets as sensitive, even when they were not (e.g., "list of user followers" is public in the context of a social network application, but was sometimes labeled as sensitive.). A correct data model is crucial for automated detection since most inspection guidelines suggest focusing on security critical information in the model. Given an incorrect data model, the query pattern was looking for flaws in the wrong locations, producing FPs. During the manual inspection of the results vis-a-vis the ground truth (on 15/26 models), we have marked such FPs to determine their weight. For this flaw, 4 out of 17 FPs were due to mislabeled assets. Therefore, aligning the sensitivity of the modeled assets to the expert analysis would already increase the precision of detecting this flaw.

Authentication bypass (Flaw 2) requires inspecting the entry points of the system to determine if authentication is modeled correctly between the external entities and the processes of the system. In total, there are 57 actual flaws ($TPs + FNs$) of this type. Yet, the query pattern detects 86 flaws ($TPs + FPs$). Out of those, many are FPs (58), and only 28 are TPs. We provide two possible explanations. First, the experts took modelers' intention into account while inspecting the models. If the participants modeled authentication incorrectly, minor mistakes were intentionally overlooked, and the flaw was not reported. The query pattern does not perform any quality check of the diagram, which yields FPs. Second, compared to the query patterns, experts often reported this flaw on different DFD elements. Given that the DFD model is a kind of *directed* graph, our model distinguishes incoming (element is consuming the data) to outgoing data flows (element is sending data). The query patterns report this flaw on the outgoing data flows (i.e., for the data being sent from the external entity), whereas the experts reported this flaw on the incoming data flow (i.e., for the data being consumed by the external entity). This yields both FNs and FPs.

Insecure data exposure (Flaw 15) achieved a high recall (0.87) but performed worse in terms of precision (0.51). Similar to flaw 18, a possible explanation for the high number of FPs (162 compared to 166 TPs and 24 FNs) is an incorrect data model. Here, the assets are traced, and the flaws are reported for each model element, from asset source to asset sink. Thus,

Table 8.4: Overall precision and recall across systems

System	TP	FP	FN	P	R
BeSocial	95	88	24	0.52	**0.80**
DriveSafe	67	59	21	0.53	0.76
PhotoFriends	95	70	32	**0.58**	0.75
SmartTex	77	83	28	0.48	0.73

incorrectly labeled assets may have a larger impact on the precision and recall of the query pattern for detecting design flaw 15. The relatively small number of FNs (24) shows that, at least, violations were not overlooked by this query pattern. This also suggests that the participants over-approximated (rather than under-approximated) the sensitivity of assets.

Insufficient key management (Flaw 6) achieved the highest recall (0.93) but performed worse in terms of precision (0.61). The inspection guidelines for insufficient key management suggest identifying cryptographic keys in the model, and analyzing their distribution, storage, creation, replacement, and destruction. The extensions to the DFD notation enable modeling key creation, replacement, and destruction. These security solutions are linked to the assets (of type 'key') and are checked for presence by the execution of the query pattern. For key distribution and storage, the query pattern leverages helper queries implemented for detecting design flaws 15 and 13, respectively. Therefore, the observed FPs occur for similar reasons to the ones discussed in those flaws.

Insecure data storage (Flaw 13) achieved a fairly acceptable recall (0.71), and a relatively good precision (0.83). We investigated the reason for a sub-optimal number of FNs. One possible explanation is that the inspection rules for security design flaws 13 and 18 overlap. For instance, consider the inspection rule "Is access to data logged?" (from Flaw 13) and "Is access to sensitive data and operations logged?" (from Flaw 18). A systematic application of the inspection rules therefore results in reporting the same violation twice (once for Flaw 13 and once for Flaw 18). During model inspection, experts agreed to report such a flaw only once (the first one they found, which was usually while inspecting for Flaw 13). Instead, missing logs of access trigger the query pattern detecting Flaw 18 (and not Flaw 13). This yields FPs for the pattern detecting Flaw 18, and FNs for the pattern detecting Flaw 13.

8.6 Discussion

This section discusses the construction of the data set, and the challenges specific to automating design flaw inspection.

8.6.1 Creation of the Data Set

During the creation of the data set we have taken additional steps to ensure that the expert assessors calibrated their inspection to achieve repeatable results. Even so, 33% of the reported flaws (over 5 problematic models) were not agreed upon and had to be revisited. The experts had to agree on a common strategy for understanding different requirement interpretations and handling modeling ambiguities. This required more calibration than anticipated.

Requirement interpretations. Early-architecture design models are often created from incomplete system descriptions. Therefore, creating such models means dealing with unknowns and under-specified documentation. If the participants made functional mistakes, the experimenters intervened and warned them to revisit the system description. A systematic assessment of functional correctness was not carried out, as this was seldom the case. But, some security requirements were interpreted differently by our participants. For instance, the requirement: "*In no event, the documents of a customer should be exposed due to a security breach. Hence, the documents have to be stored securely,*" was often understood to require assurance of confidentiality (of documents) for transfer and storage. Another interpretation is that the documents must be stored in a secure storage to which access is logged. In particular, Group A (less security background) often made over-approximations when interpreting security requirements. Different requirements interpretations caused participants to extend the DFD with a different data model and, in consequence, different security solutions. This has an effect on the presence or absence of security design flaws. To understand the model (in particular the rationale for extensions), the assessors had to reconstruct the rationale for the created data model vis-a-vis the requirements.

Modeling ambiguities. Different modelers have a variety of ways to model the same software system with the same requirements. As shown in Figure 8.3, models of the same system can vary in size (e.g., the largest (56) and smallest (17) BeSocial model). These different modeling options have an effect on the presence or absence of security design flaws. For instance, sometimes the participants modeled interactions between the external entity and an authentication process, and between the external entity and all processes representing system functionalities. The participants implicitly assumed sequential and conditional data flows (i.e., the authentication process is invoked first, and only upon success, can the other functionalities be executed). Since the extended notation does not allow a specification of conditional or sequential data flows, this model is ambiguous, and the authentication bypass flaw could be present. The assessors had to interpret the modeler's intention to handle ambiguously modeled DFDs.

To help a manual inspection, we see benefit in (i) operationalizing the guidelines for inspection with reference to element types, and (ii) introducing quality checks for the extended DFD notation.

8.6.2 Automation

In what follows we describe challenges specific to the automation of design flaw detection and discuss how they can be overcome.

Informal notation. The query patterns were developed by translating natural language inspection rules to relations between elements of the extended DFD notation. According to this translation, the query patterns search for concrete diagram element combinations that are incorrect or problematic. Such an implementation can be broad (e.g., checking for the absence of a security solution). Still, this cannot account for all potential modeling options as modelers may apply shortcuts and (un)intentionally circumvent the detection mechanism. For instance, if the model does not contain sensitive assets, the query patterns will not find insecure data exposure flaws on data in transit. Therefore, the security design flaw detection inherits the problems from the DFD modeling ambiguities.

Modeler assumptions. Furthermore, any model-based detection mechanism relies on these models to precisely reflect the modeler's intentions. It may, however, be possible (due to misinterpretations) that the models actually represent a different situation than intended by the modeler. For example, modeling application-level encryption, but specifying the resulting encrypted data as sensitive, will cause automated assessments to consider the encrypted data not to be protected against information disclosure.

Similarly, the modeling concept of data is very open and supports many interpretations of which data types exactly would need to be modeled. Given the reliance of some flaws on the sensitivity of data as a key criterion to determine their applicability, the degree of detail in modeling data and correctly assigning its sensitivity has a considerable impact on the detection, as shown in the example with the encrypted sensitive data above.

Going forward. The query patterns are executed on finished models, aiming to achieve a fully automated design flaw detection. This approach does not explore the potential benefits of providing modeling feedback to the modeler while the model is being constructed. Computer-aided detection could overcome some of the modeling challenges discussed above. For instance, our approach could be extended with an appropriate user interface to guide modelers and alert them for potential security design flaws, continuously. Such guidance can also assist modelers in avoiding modeling ambiguities and ensure a more accurate detection of security design flaws. Finally, our approach can be extended to implement the detection of other security design flaws from the catalog.

8.7 Threats to Validity

8.7.1 Internal Validity

The threats to internal validity that we have identified relate to: (i) the descriptions of the four systems, (ii) the construction of the DFD models, (iii) the extension of the DFD models with the data types, and (iv) the construction of the expert assessment baseline.

Descriptions of the four systems. Some of the security requirements mentioned in the descriptions of the four systems might have required the participants to use security solutions which were not provided (as out-of-the-box extensions) or straightforward to model. This threat also relates to the limited security expertise of some participants. In addition, some security requirements were open to interpretation (as discussed in Section 8.6).

Construction of the models. For the construction of the 26 models there are three concerns. First, the participants had a limited familiarity with the graphical user interface of the modeling tool. To counter this threat, all participants started with a warm-up modeling exercise to ensure they were able to create models and had no remaining questions. Also, at least one author was always present to assist in case any questions or issues arose. Second, the learning effect of working on two systems per participant was controlled by a balanced distribution of the systems. Third and finally, the participant might have perceived some stress in trying to create the models in the foreseen time slots and fatigue due to the length of the session. However, we remark that all participants finished earlier and they could take short breaks if needed.

Extending the models with data types. Since there was no graphical modeling support for adding the data types, participants had to use textual labels on the data flows to specify the data types. These descriptions later had to be included in the model to enable the automated flaw detection patterns to take them into account. We consider the threat of modeling errors that could have been introduced by the authors, when creating the data model from the textual labels. To reduce the impact of errors in the modeling, all models have been checked by two authors.

Construction of the expert assessment. Concerning the assessment of the models by the two experts, we acknowledge that such an assessment could contain errors. For instance, the experts had to interpret the modelers' intentions when assessing the presence of the security design flaws. However, the probability of these errors has been minimized by applying two separate calibration steps between the security experts that performed the assessment.

8.7.2 External Validity

With respect to the generalization of the results, there are two main threats to the external validity. First, the participants might not be representative of industry professionals. All the participants were researchers with modeling experience, while 5 out of 13 participants had security expertise. Second, the results might be specific to the systems used in this paper. To reduce the impact of this threat, we relied on four different system descriptions which were randomly assigned to the different participants. However, these four systems are relatively similar in size because the participants had to be able to create them in a limited amount of time. An evaluation on systems with varying sizes may be useful to evaluate the impact of the model sizes on the effectiveness of the automated detection.

8.8 Related Work

In this section, we position our contributions in the context of the related work on automating security design analysis. We also discuss related security design flaw catalogs and works on automating the detection of architectural bad smells and anti-patterns.

8.8.1 Automation of Security Design Analysis

Recently, Seifermann et al. [43] presented an approach for automatically analyzing security of data-driven architectures. To this aim, they propose an architectural description language enriched with a data model. They transform the architecture to an operation model, which in turn, is automatically transformed to a logic program, where the security analysis is executed. Similar to our data transformations, Seifermann et al. define data processing operations, which seem to be essential for analyzing confidentiality. But, our detection also considers existing countermeasures. Further, Seifermann et al. demonstrate the analysis with logical rules for detecting unauthorized access. Instead, our work is automating the detection of flaws related to several concerns (namely, authentication, confidentiality, integrity, and accountability).

Almorsy et al. [44] propose an approach for automating security analysis by means of capturing security metrics and vulnerabilities as constraints over a detailed system description model. It is beneficial for the analysis to consider system vulnerabilities and defenses side-by-side. Similarly, our query patterns consider design flaws with respect to the security solutions. In addition, the constraints developed by Almorsy et al. [44] rely on the modeler to provide a model. In particular, the constraint about data tampering seems similar to our query pattern detecting insecure data exposure (15). Yet, the introduced constraints detect attack scenarios (e.g., denial of service) and assess the system's implemented security (e.g., defense-in-depth), rather than security design flaws.

Berger et al. [45] develop graph query rules to check for vulnerabilities in extended DFD models and evaluate them with case studies. Similar to this work, these rules are based on the descriptions of existing repositories (namely, CWE, and CAPEC). In addition, their approach also extends the DFD model with asset sensitivity. Among others, Berger et al. develop queries to check for authentication bypass and clear text transmission of sensitive information (similar to our query patterns for flaws 2 and 15). An important distinction is that our DFD extended notation includes security mechanisms.

UMLsec [57, 252] is a security extension of the Unified Modeling Language. It enables developers to express security relevant information in system specification diagrams. Similar to our approach, UMLsec relies on security extensions to automatically analyze design models. In contrast, our approach is focused on detecting early security design flaws on architectural models, as opposed to evaluating constraints over specification diagrams.

Katkalov et al. [65] developed a model-driven approach (IFlow) for specifying and analyzing information flow properties on UML models. The authors extend the UML model and transform it to a formal model, which is used to refine the UML generated code skeleton. The proposed approach can leverage static analysis to verify information flow properties in the implemented code skeleton, as well as an interactive theorem prover to verify the properties on the formal model. Similar to this work, IFlow requires the developer to provide information about the sources (of confidential information). In contrast, IFlow is a formal approach that analyzes the non-interference property.

Hoisl et al. [91] present an approach for modeling and enforcing object flows in process-driven Service-Oriented Applications (SOAs). The authors provide a metamodel for defining and enforcing secure object flows in process-driven Service-Oriented Architectures and develop model transformations to generate platform specific models. Similar to the data transformations in our data model, they introduce a semantics of control nodes (i.e., fork, join, decision, and merge) to reason about secure object flows. In addition, their approach is used to automatically analyze the confidentiality and integrity of data flows in a model representation. But, this work extends the model notation with security solutions and focuses on the detection of a variety of security design flaws.

Frydman et al. [92] propose an approach accompanied by a tool (AutSEC) for automating threat modeling and risk assessment of software designs. To identify threats in annotated DFDs, the authors introduce identification and mitigation trees. They obtain the DFD annotations by maintaining maps of common DFD element labels (e.g., web server can be mapped to the "Apache" label). Similar to this paper, Frydman et al. extract information about assets and security mechanisms from the user-extended DFDs. Yet, their diagram extensions are based on user-provided labels and map-like data structures. In contrast, this work allows modelers to explicitly model data properties and security solutions in DFDs.

For a more detailed account on automated design analysis techniques, we refer the reader to a systematic literature review [208].

8.8.2 Security Design Flaw Catalogs

We briefly mention the related work on security design flaw catalogs. Santos et al. [253] compiled a catalog of common security architectural weaknesses. Similar to the catalog used in this work, their catalog focuses on design-level security flaws. Arce et al. [241] compiled a list of top 10 security design flaws to raise awareness among software architects about the most common design issues leading to security breaches. Indeed, a few inspection guidelines of the catalog used in this study are in line with this list (e.g., 'use cryptography correctly' is related to our design flaw 6). Nafees et al. [238] propose a template for detecting architectural anti-patterns and a catalog of 12 Vulnerability Anti-Patterns. The purpose of their catalog is to bridge the communication gap between security experts and software developers. We also mention the existing corpora (i.e., CWE, CVE, OWASP, SANS, CAPEC) describing common security weaknesses, vulnerabilities, and mitigations.

8.8.3 Architectural Bad Smells and Anti-Patterns

We briefly mention the related work on detecting architectural bad smells and anti-patterns but remind the reader that none of these approaches analyze the architecture with respect to security. Bouhours et al. [235] introduce a catalog of so called 'spoiled patterns' and automatically detect them in architecture models. To this aim, they extend an existing OWL ontology. Their approach suggests according model transformations to the user. Taibi and Lenarduzzi [236] compile a catalog of 11 bad smells specific to microservice architectures by means of conducting interviews with developers. They emphasize the importance of analyzing microservices that expose private data and shared resources, which is interesting from a security perspective. For a more complete account of existing literature on design smells detection we refer the reader to the mapping study by Alkharabsheh et al. [254].

8.9 Conclusion

This paper has presented three main contributions. First, we have shared with the research community a data set of design models (in a notation based on DFD), created by thirteen participants with a varied background, that model four systems with a varied set of security requirements, and that are annotated with identified design flaws. These models can be used to validate existing and future security techniques. Second, we have attempted to automate five model inspection guidelines for security to detect secure design flaws. These guidelines are meant to be used by security experts and, hence, are difficult to automate, as humans are better suited to execute tasks that involve fuzzy and/or incomplete instructions. Third, we have performed an empirical evaluation that, compared to the ground truth created by manual analysis, shows that automating some of the guidelines is possible with acceptable precision and recall, albeit others are more challenging. Also, our work has pointed out some limitations (e.g., overlaps, unclear rules) in the inspection guidelines themselves. As part of the future work, the results of this paper are being used to improve the quality of the inspection guidelines. Further, we are interested in extending the data set and particularly welcome the contribution of the wider research community.

Chapter 9

Papers H & I

This chapter is based on
Security Compliance Checks between Models and Code based on Automated Mappings,

written by
S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, and R. Scandariato,

published in
Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), 2019

&

Checking Security Compliance between Models and Code,

written by
K.Tuma, S. Peldszus, R. Scandariato, D. Strüber, and J. Jürjens,

submitted to
Journal on Software and Systems Modeling (SoSyM), 2020.

Note that Paper I subsumes Paper H, thus only Paper I is appended in what follows.

Abstract

The verification that planned security mechanisms are actually implemented in the software code is a challenging endeavor. In the context of model-based development, the implemented security mechanisms must capture all intended security properties that were considered in the design models. Assuring this compliance manually is labor intensive and can be error-prone. This work introduces the first semi-automatic technique for secure data flow compliance checks between design models and code. We develop heuristic-based *automated mappings* between a design-level model (SecDFD, provided by humans) and a code-level representation (Program Model, automatically extracted from the implementation) in order to guide users in discovering compliance violations, and hence potential security flaws in the code. These mappings enable an *automated*, and *project-specific* static analysis of the implementation with respect to the desired security properties of the design model.

We contribute with (i) a definition of corresponding elements between the design-level and the implementation-level models and a heuristic-based approach to search for correspondences, (ii) two types of security compliance checks using static code analysis, and (iii) an implementation of our approach as a publicly available Eclipse plugin, evaluated with three studies on open source Java projects. Our evaluation shows that the mappings are automatically suggested with up to 87.2% precision. Further, the two developed types of security compliance checks are relatively precise (average precision is 79.6% and 100%), but may still overlook some implemented information flows (average recall is 65.5% and 94.5%) due to the large gap between the design and implementation. Finally, our approach enables a project-specific analysis with up to 62% less false alarms raised by an existing data flow analyzer.

9.1 Introduction

For decades, organizations have been concerned with the security of their software throughout the entire development process. According to the principle of security by design [7, 255], the analysis of system assets vis-a-vis security threats needs to be carried out already in the design phase of the development process. In this context, threat analysis techniques (e.g., STRIDE [9], attack trees [207], CORAS [25], and threat patterns [124]) aim to identify security threats to software systems by scrutinizing the architectural design. But, empirical evidence shows that existing threat analysis techniques can be labor intensive [10] and lack in automation [208].

Threat analysis is often performed on a graphical representation of the software architecture called *Data Flow Diagram* (DFD) [11, 249]. DFD-like models are extensively used in practice, e.g., in the automotive industry [14], at Microsoft [9], and some agile organizations [15]. Still, the DFD notation is informal and lacks the ability to specify security properties, which are needed to reason about security threats at the design level [256]. To support the detection of problematic information flows at the design level, previous work extends the DFD notation with security-relevant information [90, 249] and security semantics [248]. This work uses one such extended notation, namely the Security Data Flow Diagram, in short, *SecDFD* [248] (cf. Section 9.2).

Once a design model has been analyzed and its security flaws fixed, the results are of limited value if the implementation does not comply with the security properties described in the model. Further, design models tend to be useful during the design phase, but are often ignored after the system is implemented. In particular, empirical evidence shows that only a fraction of open source projects (26% of the investigated projects in [12]) ever update their UML files at least once. Thus, there is a *disconnect* between the architectural design models (containing important security decisions) and the implemented system and its defenses. To be useful for security compliance analyses, an automated connection between the design model and its implementation needs to be established.

Having this connection could also benefit to static code analysis. Indeed, existing static analysis tools may report violations which are afterwards labeled as *false alarms* [94]. All reported violations have to be manually sieved through, and, more importantly, the true violations must be distinguished from the false alarms. This is not a trivial task for static program analysis in general, and in particular it is not trivial for static security analysis (as observed by an industrial experiment in [95]). Making such distinctions can be improved by the contextual information which can be derived from the (connected) design model.

To address these issues, we have proposed an approach to support compliance analysis between models and code, and extend it with security compliance checks in this work. Specifically, we have previously proposed a user-in-the-loop approach (cf. Section 9.2.3 and Peldszus et al. [257]) to support compliance checks between a design-level data flow diagram enriched with security-relevant information (*SecDFD*) and an implementation-level model called *Program Model*, or PM for short (cf. Section 9.2.2). To this aim, we have made the following contributions:

(i) We presented an automated technique for establishing mappings between SecDFDs and PMs (Section 9.3), thereby supporting the discovery of structural compliance violations. The key idea of our technique is twofold. First, we defined a mapping between SecDFD and PM element types, constraining how elements of a concrete system can be mapped to each other. Second, we combined similarity-based matching of element names with structural heuristics (based on data flow properties) to automatically derive suggested mappings.

(ii) We presented an incremental technique where the user is (a) involved in discovering an adequate mapping and (b) able to inspect the planned security properties against the implementation.

(iii) We implemented (Section 9.5) the approach as a publicly available Eclipse plugin and evaluated it (Section 9.6.1) on five open source projects.

In this paper, we present *three extensions* which allow an automated security analysis between SecDFD and its implementation:

(i) We develop static checks to verify security properties (i.e., SecDFD contracts) in the implementation (Section 9.4.1). Specifically, we develop two types of checks: a rule-based check for cryptographic contracts (encrypt and decrypt) and a local data flow check for data processing contracts (forward and join).

(ii) We develop an automated extraction of project-specific sources and sinks of confidential information from the design, which we leverage to reduce the number of false alarms raised by an existing data flow analyzer (Section 9.4.2).

(iii) We extend our implementation (Section 9.5) with (a) automatically executable security compliance checks, (b) additional user-interface functionalities (e.g., the graphical view of the SecDFD, mapping several SecDFD models, etc.), and (c) an improved mapping serialization.

We evaluated the previously proposed approach with an experiment (Section 9.6.1) which showed a high precision and recall of the suggested mappings [257]. Further, we have shown that the user has an impact on the suggested mappings, and can steer the automation. Two studies were conducted to evaluate the extensions presented in this paper (Section 9.6.2 and Section 9.6.3), where we measure the precision and recall of the security contract checks, and the impact of using the SecDFD model to derive project-specific sources and sinks on the number of false alarms raised by an existing data flow analyzer. The security compliance checks developed for the cryptographic process contracts are very precise (average precision is 100%) and rarely overlook implemented cryptographic operations (average recall is 94.5%). In comparison, the local data flow security compliance checks are less precise (average precision is 79.6%) and may overlook more implemented flows (average recall is 65.5%). Considering the vast gap between design models and their implementation, this is still an encouraging result. In addition, we show that the proposed approach enables a project-specific taint analysis with up to 62% less false alarms. Evidently, these results are valid within the bounds of the threats to validity presented in Section 9.7. We position our contributions in the context of the related work in Section 9.8 and present the concluding remarks in Section 9.9.

9.2 Background

This section describes the background on the design-level model, implementation-level model, architectural compliance, and data flow analysis. We consider the Eclipse Secure Storage [258] to illustrate the models considered in this work. The secure storage allows plugins to store and access secret data. This functionality is used, for example, by the Git extension of Eclipse to store user names and passwords [259].

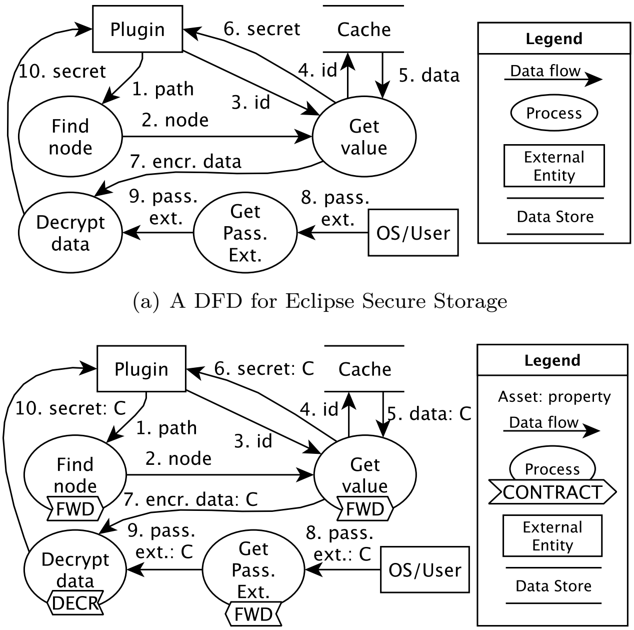
9.2.1 Design-level model (SecDFD)

At design time, the processing of system data can be specified with a variety of notations. Apart from DFDs, frequently used notations are activity diagrams [260] and business process models (BPMN [261]). Our rationale for focusing on DFDs is twofold: First, they are widely applied in practice, specifically, in the automotive industry [14] and at Microsoft [9] as part of their STRIDE methodology. Second, they represent an essential set of concepts necessary for data flow analysis (processes and data flows between them), which be can mapped exhaustively to activity diagrams and business processes, rendering our mapping generation technique also applicable to these model kinds. We introduce our technique for DFDs, but it can be applied to a broad range of modeling languages supporting data flow modeling.

In what follows, we introduce DFDs and an extended notation which allows to include security-relevant information in DFD models, which is required for checking the consistency between planned security and implemented security properties.

Data Flow Diagrams (DFDs). A Data Flow Diagram (DFD) is a graphical representation of the software architecture and the information it handles [9]. It represents how the information enters, leaves, and traverses the system. The DFD consists of processes (active entities), external entities (e.g., 3rd parties), data stores (where information rests), data flows (carrying the exchanged information), and trust boundaries (signaling trust levels). Fig. 9.1(a) depicts a DFD for the Eclipse Secure Storage. The plugin attempts to access a secret by sending a request including path information of where to look for the secret (e.g., a password request for a user name of a Git account). The secure storage queries an internal tree-like data structure to find the corresponding node containing the secret. Next, the cache is queried for the secret value, which can be in clear text (i.e., *secret* on flow 6 in Fig. 9.1(a)) or encrypted (i.e., *encr. data.* on flow 7). If the value is in clear text, the secret is sent to the plugin. In case of an encrypted value, a decrypt operation either fetches the root password from the operating system or prompts the user to provide it. Upon a successful decryption, the secret is sent to the plugin (flow 10 in Fig. 9.1(a)). Though useful for performing architectural threat analysis [210], we do not use trust boundaries in our work.

Security Extension. To capture security properties at the architectural level, we use the Security Data Flow Diagram (SecDFD [248]). SecDFD is a notation that enriches DFD with security concepts to enable a formally grounded information flow analysis, focusing on the confidentiality and integrity of information assets. First, assets can be tagged with a *high* or *low*



(a) A DFD for Eclipse Secure Storage

(b) An excerpt of the SecDFD for Eclipse Secure Storage

Figure 9.1: A DFD (top) and an excerpt of the SecDFD (bottom) for Eclipse Secure Storage

confidentiality label. Second, process nodes can be tagged with security contracts that define how the security properties of assets change upon exiting the node. The SecDFD defines four such contracts.

- Encrypt or Hash contract. The contract for encrypting input asset(s) always results in propagating a low (public) label on the output flow(s).
- Decrypt contract. If the input asset is low decrypting it will result in propagating a low label. However, if the input asset is high decrypting it will result in propagating a high label on the output flow.
- Join contract. The contract for joining two or more assets propagates the label equivalent to the most restrictive input asset. For example, if a confidential asset is joined with a non-confidential assets the asset on the output will be confidential.
- Forward or Copy contract. This contract will copy the labels of the input asset(s) to the output flow(s) carrying the corresponding forwarded asset(s).

Finally, the model elements can be grouped to attacker zones. An attacker zone specifies the groups of elements which can be observed by an attacker of a specific profile. The user can define a hierarchy of attacker zones with different attacker profiles.

Fig. 9.1(b) shows an excerpt (for clarity) of the SecDFD for the Eclipse Secure Storage example. If a plugin requires secret data that is cached encrypted, the user must enter a *password* when prompted (c.f. *pass. ext.* in Fig. 9.1(b)). The externally provided password is then used to decrypt the cached secret

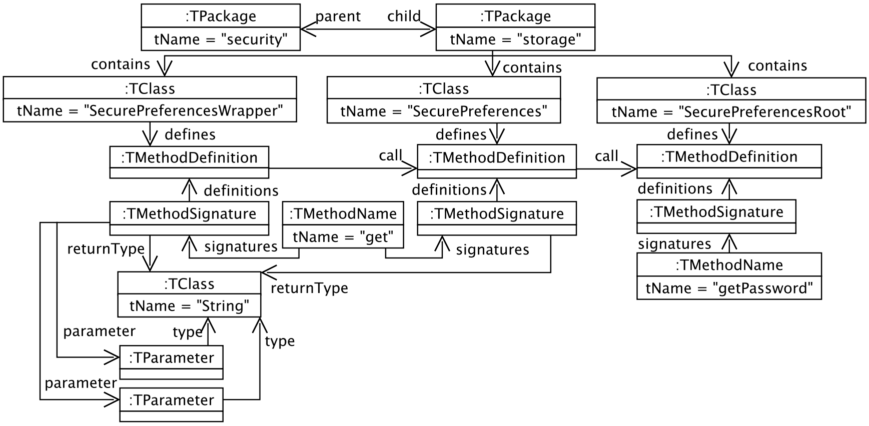


Figure 9.2: Excerpt from the Program Model (PM) of the Eclipse Secure Storage (shown as UML object diagram)

data, and if successful the plugin is allowed to read it. First, the designer must specify that the external password is confidential. Second, the designer needs to specify the process contract (e.g., for process *Decrypt_data*). Since the external password is confidential, it should not be leaked to other plugins running in the environment. These simple extensions allow us to identify such behavior in the model. For instance, the extended notation [248] is shipped with a simple label propagation (using a dept-first search) according to the specified process contracts. Once the labels have been propagated, a static check is executed to determine if any confidential information flows to an attacker zone. In Figure 9.1(b), the Plugin is not a malicious entity (i.e., it is not part of an attacker zone). The developer can manipulate the elements of attacker zones to change the design model and improve security.

The specified security properties can be also propagated from the SecDFD to code using the mappings created by our approach. They can then be used as input for code-level analysis tools, thus enabling compliance checks between planned and implemented security properties (see Sect. 9.3.5). For the concrete syntax and semantics of SecDFD we refer the reader to [248].

9.2.2 GRaViTY Program Model (PM)

To create a mapping between SecDFDs and their concrete implementation we need an easy to analyze representation of the source code. Representations such as abstract syntax trees (AST) contain every detail from the implementation, which makes it hard to analyze for security purposes. Many details about the implementation are not required for our approach. At the same time, important information is not always directly accessible. For example, in the source code files or an AST accesses of fields are not directly visible as access edges between the source and the accessed field, but are access statements within the source to some field with a given name. For our approach, it is only important to know that there is an access to a specific field from some source, but we do not need to know every detail about the circumstances of this access. The Program Model, herein PM (proposed with the *GRaViTY*-framework [13, 262, 263]) creates a more suitable abstraction for security analysis and allows easy queries, which were very useful for our approach. The *GRaViTY*-framework has been used for the evaluation and execution of refactorings [263], for the detection of anti-patterns [264], as well as for the automated design optimization of Java applications [265].

Fig. 9.2 shows an excerpt of the PM created by the *GRaViTY*-framework for the Eclipse Secure Storage example. The figure shows two method calls. The first call is from the method `get(String, String)`, defined in the class `SecurePreferencesWrapper`, to the method `get(String, String, SecurePreferenesContainer)` of the class `SecurePreferences`. The second call is from the called method of the first call to the method `getPassword(String, IPreferencesContainer, boolean)` which is defined in the class `SecurePreferencesRoot`.

On top of the figure we can see the package structure of the program. All packages without a parent can be taken as entry points for a search. Additionally, it is possible to iterate over all types directly. The types (in this case classes) are shown in the second row, each with a reference to the members defined within the type. For the classes `SecurePreferencesWrapper`, `SecurePreferences`, and `SecurePreferencesRoot` only a single method is shown. Methods are represented by a triple of method name, method signature and method definition. This allows an efficient search for specific methods, starting with the method name, continuing with signatures, and finding concrete definitions for them. Method signatures have parameters which have a reference to the type representing the parameters type and a reference to the return type. In the PM excerpt only the parameters of the signature `get(String, String):String` are shown.

A benefit for our mapping from SecDFD to Java implementations is the possibility of an iterative search, starting only with little knowledge about the searched elements–e.g., a method name. The PM allows to start a search with such little information and to find more concrete elements by considering more information like method parameters without iterating over all method definitions defined in the source code.

9.2.3 Compliance

Identifying the differences and equivalences between the planned and the implemented software architecture is the goal of architecture compliance checking. The compliance checks can be based on a static set of rules [266], dynamic monitoring of a running system [267], or a hybrid of both [93]. In our work, we statically check the compliance of design-level models to implementation-level models. Running compliance checks reveals the relations between a set of components of the first (design-level) model and a set of components of the second (implementation-level) model. As outcome, three different types of relations can be discovered.

Convergence. The compliance checks reveal an allowed relation between the implemented components. Convergence indicates that the implementation is compliant with the planned architecture. In the context of the mappings, convergence means that the user has accepted a suggested mapping or has manually defined a mapping. In the context of security properties, convergence means that a planned security contract is implemented at the correct location and no leaks have been detected by a data flow analyzer.

Divergence. The compliance checks reveal a relation between the implemented components that is not allowed. In other words, the implementation diverges and is therefore not compliant to the planned architecture. In this work, divergence means that there are flows of assets in the implementation which have not been defined in a DFD. We look for elements that relate to existing mappings to find the relative parts of the implementation. In the context of security properties, we identify divergence when (i) there exists an implemented data flow which does not comply with the specified security contracts of the process node, or (ii) the analysis with a state-of-the-art data flow analyzer reports a leak of potentially confidential information.

Absence. The compliance checks reveal a relation between design-level components that were not implemented. Absence indicates that the source code is not compliant with the planned architecture due to a missing implementation. In the context of the mappings, absence means that the user finished using our approach, but there are still design-level elements that have not been mapped. In the context of the security properties, absence means that the SecDFD contracts have not been implemented.

9.2.4 Data Flow Analysis

Secure information flow analysis dates back to the 70s, and has been heavily studied ever since [77, 268, 269]. In principle, the idea is to perform static analysis of the program with the goal of showing that if executed, the program does not leak confidential information. Data flow analysis computes the data dependencies (i.e., which variables are dependent) to determine how data propagates in the program. Data flow analyzers take as input an abstracted representation of the code (e.g., abstract syntax tree, control flow graph) to perform the analysis. *Taint analysis* is a kind of information flow analysis where data objects are tainted at the source and tracked to the sink using data flow analysis [269]. It is one of the most used data flow analyses and has even been integrated to some programming languages (e.g., perlsec [270]). *Source methods* are characterized by reading data from a system resource (e.g., remote database or user input) and returning them to the caller. Contrarily, *sink methods* write to system resources. An alarm is raised if a tainted object (i.e., source) flows into a forbidden location (i.e., sink) in the program.

9.3 Enabling Compliance Checks with Automated Mapping Generation

Assuming a correct DFD, the way it is implemented can vary depending on concrete design (e.g., architectural patterns) and implementation specific decisions (e.g., programming language). Therefore, a fully automatic generation of a correct and complete mapping between DFDs and code is not feasible. Yet, a manual specification of the same mapping is inefficient and error-prone. To

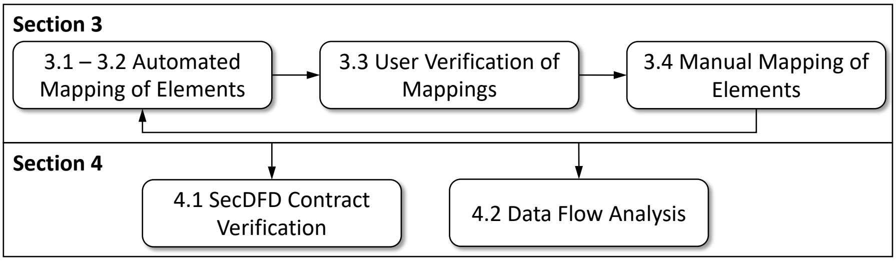

Section 3
3.1 – 3.2 Automated Mapping of Elements
3.3 User Verification of Mappings
3.4 Manual Mapping of Elements
Section 4
4.1 SecDFD Contract Verification
4.2 Data Flow Analysis

Figure 9.3: Semi-automated mappings approach (Section 9.3) and security compliance checks (Section 9.4)

this end, we propose an iterative technique for interactively guiding the user in finding an adequate mapping by combining automated mappings with user decisions as shown in Fig. 9.3. In step 1, mappings between DFD elements and implementation elements are calculated using a heuristic technique. In step 2, these mappings are presented to the user and manually checked by her. In step 3, the user can manually map additional elements. Afterwards the automated mapping is executed again, benefiting from the user input. This process terminates when the user cannot find any additional mapping or finds a violation. Next, the user can perform a security analysis of the SecDFD with respect to the implementation (Section 9.4).

In this section we describe the steps of our technique in detail, including the automated suggestions. In addition, we explain the use of these mappings for compliance checks. In Section 9.3.1, we define the allowed correspondences between DFD and PM element types. In Section 9.3.2, we show how our automated technique in step 1 establishes concrete mappings between DFDs and their implementations by means of a naming- and structure-based heuristics. In Section 9.3.3 and 9.3.4, we explain the interactive steps 2 and 3 of our technique. In Section 9.3.5 we argue how the created mappings can be used for checking general compliance.

9.3.1 Corresponding Elements

As a prerequisite for mapping DFD elements to code elements, we have to define which DFD element can correspond with which code elements.

Assets → types: The assets in a DFD are the elements holding critical data. On the level of implementation, data is usually stored in fields, processed using variables and transmitted using parameters and return values. A single asset can be stored in many different locations at the same time which makes it infeasible to map an asset to every single location. The only property of an asset which only changes rarely in programs, written in an object-oriented languages, is the asset type.

Data stores → types & methods: If we think about data stores like the cache in Fig. 9.1(a) and 9.1(b), it is quite obvious that this could be a field in some class. But it could also be implemented by an operation which, e.g., requests the cached values from an external server by creating HTTP requests. The common trait between these two variants is the type used to store the data

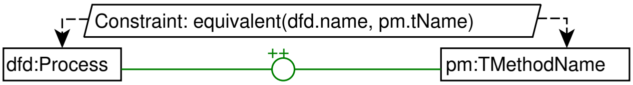


Figure 9.4: Rule describing the name matching for methods

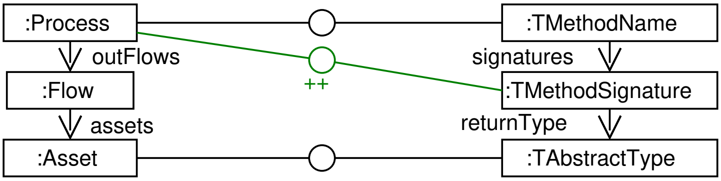


Figure 9.5: Rule for extending name matches based on return types

in. The field has a type which provides getters and setters for using the data store, and the method used to get data from a remote server is implemented in a type. Therefore, we map data stores to types as well as to the methods used for accessing the stored data.

Processes → method(-names): Processes in DFDs describe functionalities which process data, like methods in implementations do. Obviously, these two elements correspond with each other. While a concrete method definition in an implementation contains all details describing the functionality of this method, the processes only have a name describing the functionality. We assume that a developer implementing a process will chose a similar name for the methods implementing this process. This leads us to a correspondence between the names of processes and the names of methods.

Processes + Assets → method parameters: Between processes in a DFD, data can be exchanged using flows, where the exchanged data are represented by assets on the flows. In the methods implementing these processes the same data have to be exchanged. Data between methods in implementations are usually exchanged using parameters and return values. Therefore, we can combine the name mappings between processes and methods with the assets flowing into and out of a process to method parameters giving us the according method signatures.

9.3.2 Semi-automated Mapping

In what follows we discuss the steps of our automatic generation of mappings in detail.

The automated generation of mappings is based on name matchings and structural heuristics, which are sequentially executed and complement each other. For illustration, we formalize two of our mappings using graph rules, using a notation inspired by algebraic graph transformation [271] (explained below). The other mappings can be formalized in a similar way.

Name matching. First, the names of elements from a DFD are mapped to the according names in the implementation. Asset and data store names are mapped to the names of types and process names are mapped to the names of methods. Fig. 9.4 shows a rule for mapping processes from a DFD to method names from a PM. A correspondence (visualized as circle connecting the corresponding elements) between a process and a method name is created (denoted by `++`) if the constraint at the top of the rule holds. In this case the names of the two elements on the left and right of the rule have to be equivalent. The precise definition of this equivalence is described in what follows.

Names, both in a DFD and in a Java implementation, are usually build by concatenating multiple words. For example, a Java method name `getPassword` consists of the word *get* and *password*. These words can vary slightly in the names of the corresponding DFD processes (e.g., in plural form, *passwords* instead of *password*). In addition the style of word concatenation can differ. In Java usually the camel case (`getPassword`) is used, whereas in DFDs this is not a prescribed style, so underscores may also be used (*Get_Passwords*).

To deal with these issues, we first split the strings at frequently used delimiters and upper-case characters. This gives us for the example the sets of words *[get, Password]* and [Get, Passwords]. Then we compare the lower-case versions of the words with each other using a fuzzy compare based on the Levenshtein distance [97]. The Levenshtein distance is a measure of the minimal amount of characters which have to be removed, added or flipped to change one word into the other one. For the given example this distance is zero and one as the first word is already identical and only the character *s* has to be added to change *password* into *passwords*. We accept different distances between words for considering them as identical according to the length of the words to be compared.

Finally, a DFD process is usually implemented in multiple methods, typically having slightly more concrete names. For example besides the method `getPassword`, there might also be an additional method `internalGetPasswod` involved in the implementation of the process *Get_Passwords*. But the DFD process name might also contain additional information – e.g. the process *get_Passwords_External* of the DFD in Fig. 9.1(a). To address this challenge, we compare all words from the two names with each other and count the similar words. If this number reaches an threshold of more than half the number of the average words of the compared names, we consider the names sufficiently equal.

For the example DFD in Fig. 9.1(a) and the PM excerpt in Fig. 9.2 we get a name match between the *Get_Value* process and the two method names `get` and `getPassword` as well as a match between the process *Get_Passwords_External* and the method name `getPassword`. While two of this matches are expected, the match between *Get_Value* and `getPassword` is unexpected and should be dropped in the following steps.

Extending name matches to method signatures. For every method name, multiple signatures may exist. Even if our name matches were always perfectly correct, this would not imply that all signatures with this name are the ones corresponding to a given process. For example, besides the relevant signature `getPassword(String, IPreferencesContainer, boolean):PasswordExt`, there might be a second signature `getPassword():char[]` defined in the Java standard library which is never used in the implementation. To identify the relevant signatures, we use data flow information about assets flowing into and out of a process. Information flowing into a process has to be passed to the implementation of the process, for example, as a parameter value. Likewise, information leaving a process can leave it over return values and parameters. Accordingly, we can use the mapped assets to identify relevant signatures. For every signature, we count how many mapped assets are compatible with the parameters and return types of the existing signatures. If we have at least one match we consider this signature for further mappings.

A rule for extending a process mapping based on an asset flowing out of a process is shown in Fig. 9.5. On top of the rule we can see an existing mapping between a process and a method name, as e.g. created by the rule shown in Fig. 9.4. A mapping to one of the signatures having this name is created if there is an mapping between an asset flowing out of the process and a type which is the return type of the signature.

If we look at the return type of the signature `get(String, String):String` and assume that the *secret* asset from Fig. 9.1(a) has been mapped to the class `java.lang.String` we'll accept this signature as corresponding with the process *Get_Value*. The other method name corresponding with this process was `getPassword`. The return type of this method signature is `PasswordExt` and also no parameter type is matching to an asset. Accordingly, we don't create a correspondence.

Finding implementations of signatures. The last step is to find concrete implementations of a signature corresponding with the process. For every signature there might be several concrete implementations, all of which do not necessarily correspond to the process. We make use of the flows between different processes to find the concrete definitions.

If there is a flow from one process to another process, this does not only mean that there has to be a signature which has the capability to return or receive the according asset. There also has to be a definition of this signature which is called from a definition in the other process. Therefore, we search for two kinds of data flows between the concrete definitions of the signatures found before.

[A] Parameters passed by a call from the source of a flow to to the target of the flow.

[B] Return values returned along a call from the target of a flow to the source of the flow.

The flow between two such definitions is not necessarily a single direct call between the two definitions. There can also be multiple definitions in between forwarding data. For example we can see in Fig. 9.2 a call between the methods `get(String, String, SecurePreferenesContainer):String` and `getPassword( String, IPreferencesContainer, boolean):PasswordExt` but in the DFD in Fig. 9.1(a) there is no flow between the processes *Get_Value* and *Get_Passwords_External*, they have been mapped to. In the implementation the `get` method forwards the return value of `getPassword` to a call of method `decrypt` which has been mapped to the process *Decrypt_data*. Matching this intermediate to one of the two involved processes is non-trivial. However, if we found such a flow, we can definitely assume that we found two definitions implementing at least parts of the two processes.

The intermediate definitions can be partly mapped to one of the two processes by considering the internal coupling in a process. For every pair of signatures mapped to the same process, we look for pairs of definitions calling each other. For example, this is the case for the definition of the signature `internalGetPassword`, which is called by `getPassword(String, IPreferencesContainer, boolean):PasswordExt`.

Cleanup. After matching assets and processes we have to decide which matches are most likely to be correct and, therefore, should be presented to the user. For that reason, we introduce a certainty score for our mappings. This score is calculated with respect to the quality of the underlying name matching as well as the coupling of matched elements with other matched elements. For every DFD element we only present mappings whose score is higher or equal to the median score of all mappings for this element.

The mappings sorted out in this step are not presented to the user, but may be discovered later again in the interactive process – based on future matches, which might have a coupling to the elements that are now discarded.

9.3.3 User Verification of Mappings

The mappings created in the previous step are now presented to and verified by the user. For every asset-, data store-type and process-definition mapping the user can preform three actions.
Accept: The user can accept the mapping. From then, the mapping cannot be discarded by the optimization step of the automated mapping approach anymore, and all mappings coupled to this mapping obtain a higher certainty score.
Reject: The user can reject the mapping. From then, this mapping is never presented to the user again and it is not considered anymore for extending it to other mappings. All other mappings to which the rejected mapping has been extended will be removed, too, but might be presented to the user again.
Tolerate: The user can choose to ignore some suggested mappings. Mappings that are not explicitly accepted or rejected are suggested again and can be re-assessed in future iterations.

Mappings accepted or rejected by the user allow the heuristic to automatically discard related mappings that have only been found by following up the rejected mapping. This is how the search space is reduced in the next automated iteration. Conversely, manually accepting mappings can lead to the score of related mappings being increased and, for this reason, allow to propose new mappings which haven not been considered as correct ones before. Anyhow, a limitation of our heuristic is that they cannot detect mappings which are outside of the search space created by the initial name mappings. We are overcoming this limitation in our approach by including user feedback as described in what follows.

9.3.4 Manual Mapping of Elements

To increase the search space, an additional user step is conducted after the user manually verified the automatically created mappings (or at least a part of them). In this step, the user has to add at least one new mapping to give additional input to the automated mapping algorithm. The selection of this manually mapped element can have a large impact on the efficiency of the following automated steps.

9.3.5 Compliance of Models and Code

The mappings can be used to perform compliance checks. In what follows we describe the check developed to determine if the implementation corresponds with the specification in the DFD.

The correspondence checks take place while the mappings are created. Using the proposed approach, we check for the three kinds of correspondences introduced in Section 9.2.3:

Convergence. All DFD elements which have been mapped to implementation elements and have not been rejected are allowed to be mapped. Following the definition of convergence, the convergences between the DFDs and the code are described by the set of all allowed mappings.

Divergence. Elements present in the code, but not specified in the DFD represent a divergence between the DFD and code. To help the user discovering divergences, it is possible to show all flows from members mapped to one process to other members not mapped to this process. If the target of such a flow has not been mapped to any process, there seems to be a divergence. But, a divergence also arises if there is a flow between two processes in the code that has not been specified on the DFD. If an critical asset is communicated along such a flow this is not only a divergence from the intended design but a security violation.

Absence. If we are neither able to map a DFD element to the code automatically and the user is not able to map the same element when asked, we discover an absence of specified functionality in the code. Assuming correctness of DFD models, we only have to consider this one direction of absence (concerning the opposite direction, see *divergence*).

Using these checks, a developer or code reviewer can detect a compliance issue between an DFD and the implementation at hand. However, regarding security, these checks are not precise enough: They might not reveal flows of confidential assets into parts of the program that are not supposed to take place – e.g., if a developer uses a full representation of an object, instead of a stripped one. To this end, we can perform more sophisticated security checks, as described in what follows.

9.4 Security Compliance with Static Program Analysis

After the user creates the mappings using our approach (Section 9.3.5), she can use them to verify the security of the implemented systems. Besides precise security checks that are addressed in the main part of this section, the created mappings can also be used to address security on an organizational level.

One approach to achieve a secure system is to structure it into different security levels where only some parts have to be maintained by security experts, e.g., this kind of structure can be used to isolate subjects for manual security code reviews. Unfortunately, such a structure also might erode and increase the effort required for maintaining security [272]. To detect such an erosion security metrics have been defined [273, 274]. These metrics (as many other security checks) need information about security critical parts of the system, therefore their application is often not possible.

In what follows, we demonstrate how we can transfer security related information from the design-time security models to the implementation using the created mappings. As example, we use the *Critical Design Proportion* metric, specifying the ratio between security critical and not security critical classes [274]. To calculate this metric, we have to classify all classes as security critical or not security critical. Even though the assets are mapped to types, they do not necessarily represent security critical classes, e.g., the class `String` is used to represent both secret assets but also other data. We can derive the security critical classes from the mapping by first identifying the security critical methods and afterward the classes defining these methods. These are exactly all methods mapped to a process in the SecDFD that is processing an asset tagged as confidential.

While security metrics can make security maintenance controllable and demonstrate how the information in the SecDFD can be leveraged, they do not allow to actively detect and prevent security violations. For this reason, the main part of this work rather focuses on automating a security analysis of the SecDFD with respect to the implementation (see Fig. 9.3). First, the developer can automatically verify if the specified SecDFD contracts are implemented. Second, she can automatically extract project-specific sources and sinks and perform a data flow analysis. The provided feedback of compliance violations and potential leaks may cause her to revisit the implementation, and reflect the changes in the SecDFD. First, we discuss the verification of the specified SecDFD contracts in the implementation. Second, we reveal how using our approach helps in reducing false alarms raised by data flow analysis.

9.4.1 Verification of Specified SecDFD Contracts

We developed static checks to verify the compliance of the implementation to the SecDFD encrypt, decrypt, forward, and join contracts. We assume an existing mapping between the SecDFD and the implementation before executing the checks.

Encrypt and Decrypt contracts. When executed, all encrypt and decrypt process contracts will be checked against the implementation. For each process with such a contract, we collect all the mapped method implementations that call at least one method signature performing an encrypt or decrypt operation. If at least one such method implementation exists, we consider that the process contract has been implemented, and mark it as *convergence*. If no such method implementation has been mapped to this process, we consider that the process contract has not been implemented, and mark this occurrence as *absence*.

We provide a list of well known methods that are called during cryptographic operations. We compiled this list by inspecting the Java standard security library, and packaged it together with the plugin. In addition, the user is able to add project-specific methods to this list (at runtime) via the user interface. We remark that state-of-the-art static analysis tools (e.g., SonarCube[1]) maintain similar rules for checking implemented encryption logic, but with our approach users can verify their expectation regarding the planned security.

Forward and Join contracts. The forward and join contracts at the SecDFD level describe local data flows within a process that have to be present in the implementation. To check if the specified contracts have been implemented, we propose a two-step procedure introduced in what follows. First, we extract the relevant asset-communicating flows from the implementation (i-flows).

[1] `https://www.sonarqube.org`

Second, we compare the implemented flows with the expected flows specified in the SecDFD (d-flows).

The main challenges in checking forward and join contracts are that one process can be realized by multiple methods but there are also many methods that do not belong to any process but interact with multiple processes. Furthermore, an asset in the SecDFD can be realized by different types in the implementation. For example, the encrypted data (encr. data) in Fig. 9.1(b) is realized by instances of the Java classes `String` and `CryptoData`. In addition, a single type in the implementation can be used to create instances of different assets. This is especially a problem for frequently used types like strings that can be used to represent nearly every asset as shown before.

In Algorithm 2, we show the pseudo code for the extraction of the implemented flows (i-flows) for a given process. We define an i-flow as a pair of the target of the i-flow and a set of the sources of the i-flow. The inputs to this algorithm are the process for which we want to extract the implemented flows and the mapping described in Section 9.3.1.

First, we retrieve the methods implementing the process from the mapping. For each method, we search for the relevant incoming and outgoing flows in the implementation. To this aim, we implement operations *inFlows* and *outFlows* which collect all flows into the parameters of the methods and all incoming or outgoing return flows. Next, we filter the collected flows in lines 3–8 and 10–14. For the forward and join check only the flows that can be used to communicate assets from the SecDFD are relevant. This means that the type communicated along a data flow has to be mapped to an asset. Accordingly, we filter out the flows which communicate unmapped types. At this point it is not important which assets can be communicated along the single data flow.

After filtering, for every outgoing flow we perform a backward search in line 18 and check in line 19 if we found reachable incoming flows (sources). The pair of the found sources and the target represent one i-flow, that is added to the result set *i*. If exactly one incoming data flow is propagated to the outgoing data flow, we found an *implemented forward*, and if multiple incoming data flows are propagated to an outgoing data flow, we found an *implemented join*. Note that we only consider patterns with one outgoing flow. If there are contracts in the DFD with multiple outgoing flows, they have to be split into multiple contracts. Finally, we return all found i-flows.

After we extracted the i-flows, we compare them to the expectations from the SecDFD using Algorithm 3. The input to this algorithm are the process, the mapping, and the extracted i-flows. The output is a set of identified violations (absence and divergence).

The algorithm is again based on two steps. First, we collect all possible matches between the i-flows and the expected flows from the SecDFD contracts (d-flows). We consider the implementation of a contract to be *convergent* with the SecDFD if and only if there exists a bidirectional one-to-one mapping between the d-flow of the contract and an i-flow. We call this property a biunique mapping. But, the matches are usually not biunique because of the overlapping asset type mappings, therefore we have to reduce the initial set of matches to a set of biunique mappings in the second step.

To collect the matches we iterate over every contract and every outgoing asset of the contract in lines 2 and 5. For each of these pairs we select i-flows if

```
Input  : Process $p$, Mapping $m$
Output : I-Flows $i$
$methods \leftarrow m$.methods($p$)
$in \leftarrow$ inFlows($methods$)
foreach flow $\in$ in do
    $type \leftarrow$ communicatedType($flow$)
    if m.$mapping($type$) = \emptyset$ then
        remove $flow$ from $in$
    end
end
$out \leftarrow$ outFlows($methods$)
foreach flow $\in$ out do
    $type \leftarrow$ communicatedType($flow$)
    if m.$mapping($type$) = \emptyset$ then
        remove $flow$ from $out$
    end
end
$i \leftarrow \{\}$
foreach target $\in$ out do
    $sources \leftarrow$ reachableBwd($target$, $out$)
    if sources $\neq \emptyset$ then
        add ($sources$,$target$) to $i$
    end
end
return $i$
```

Algorithm 2: Algorithm for the Extraction of the I-Flows i for a given Process p

```
Input   : I-Flows $i$, Process $p$, Mapping $m$
Output : Violations $v$
$v \leftarrow \{\}$
$matches \leftarrow \{\}$
foreach contract $\in$ fwdJoinContracts(p) do
    $inAssets \leftarrow contract$.inAssets()
    foreach outAsset $\in$ contract.outAssets() do
        $flows \leftarrow \{\}$
        foreach  iflow $\in$ i do
            $type \leftarrow$ communicatedType(iflow.trg())
            if  outAsset $\in$ m.mapping(type) and $\forall$ s $\in$ iflow.src() :
             (m.mapping( communicatedType(s)) $\cap$ inAssets) $\neq \emptyset$ then
                add iflow to flows
            end
        end
        if  flows $= \emptyset$ then
            add "Absence: Not implemented" to $v$
        end
        add (contract, outAsset)$\rightarrow$flows to matches
    end
end
$solution \leftarrow$ findSolution($matches$)
if  solution $= \emptyset$ then
    add "Divergence: No biuniqe assignment" to $v$
else
    foreach  flow $\in$ (matches $\setminus$ solution.flows()) do
        add "Divergence: Not in DFD" to $v$
    end
end
return $v$
```

Algorithm 3: Algorithm Checking the Implemented Flows i for a given Process p against the Specified Contracts

their possible outgoing assets contain the expected asset and if for every incoming flow at least one possible asset is contained in the set of expected incoming assets (see line 9 in Algorithm 3). If no such i-flow exists, the contract is not implemented (for this outgoing asset) and we detect a *divergence* (lines 13 and 14).

After collecting all possible matches, we have to find a biunique solution within the created mappings between the d-flows and the i-flows. This is implemented in the function *findSolution*. The easiest implementation is to iteratively assign i-flows to d-flows and to check if a solution is still possible. If so, we can assign the next i-flow to a d-flow, else, we have to backtrack. If we cannot find such a solution, we report a violation as there is at least one not implemented contract and we detected an *absence* (lines 20 and 21). If we found a solution, all specified contracts have been implemented and we found a *convergence*. However, all i-flows that are not part of the solution are still reported as violation as they are unspecified forwards or joins of assets and represent a *divergence*.

9.4.2 Optimized Data Flow Analysis

To perform a data flow analysis, the developer needs to identify the sources and sinks of secret data in the implementation. More importantly, to perform a meaningful and precise data flow analysis, the sources and sinks must be *identified correctly*. For instance, we have found the standard substring method in Java (`java.lang.String.substring(int, int):String`) as one of the sink method signatures in an existing list of identified sinks.[2] This will result in many false alarms raised by the analyzer, since it seems unlikely that data can leave the system through this method and it is a very common operation over strings in Java. Dually, overlooking an important source may result in overlooking true leaks. Though some sources and sinks can be extracted from library APIs [79], finding project-specific sources still remains a challenge. In addition, many data flow analyzers work with a flat security policy. Specifically, they raise an alarm if there is an access path between *any* of the source methods and *any* of the sink methods. But, certain tainted data might be expected to flow to some sinks (e.g., writing an encrypted password to local storage) but not others. If all the tainted objects are treated equally, the analyzer raises false alarms. In response to this challenge, we aim to automatically extract project-specific sources and sinks *for each SecDFD asset*.

Project-specific sources. The SecDFD requires the user to specify confidential assets, thus their source element (in the model) can easily be determined. There are three possible types of source elements: an external entity, a data base, or a process. If the asset source is an *external entity* and it is mapped to method definitions, their signatures are collected as sources. But, if a mapping of the external entity does not exist (e.g., for the entity Plugin from Fig. 9.1(a) and 9.1(b)), the signatures of the mapped method definitions of the processes reading from that entity are collected instead. If the asset source is a *data store*, it can be mapped to methods or types. First, the signatures of method definitions mapped to the data store (if any) are collected. Second, if the data store is mapped to a type (e.g., a Class), the signatures of method definitions defined by this class are also collected, but only if the return type matches the asset type. Finally, an asset source can be a *process* element (e.g., a random

[2] `https://github.com/secure-software-engineering/SuSi`

number generator). If there is no process contract with this particular asset on the output, then the signatures of the method definitions mapped to the process are collected. But, the asset may originate in the process as a result of a transformation (e.g., a join of two assets). In this case, the assets on the contract inputs are *traced backwards* reaching either an external entity, a data store, or a process with no contracts impacting the traced asset. The signatures of the method definitions mapped to the traced element are collected as sources.

Allowed sinks. We collect the sink method signatures from [79] (excluding methods of Android specific packages) and exclude the allowed sinks. The allowed sinks are maintained *for each* confidential asset. These are method implementations mapped to SecDFD elements where the confidential asset exits the system (i.e., external entities and data stores). For example, the secret flowing into the Plugin (data flow 10 in Fig. 9.1(b)) is expected to flow there. Therefore, we consider the Plugin as an allowed sink. However, since the Plugin can not be mapped to the implementation, we instead consider the method implementations mapped to the Decrypt data process as allowed sinks.

Attacker zones. The SecDFD allows the user to specify attacker zones, which denote what elements are observable by the attacker. For each asset, we collect signatures of all the method definitions mapped to elements of attacker zones and add them to the list of sinks and (if needed) remove them from the allowed sinks. In this way, the user is able to influence the security policy of the SecDFD, and perform an analysis assuming over-exposed components or APIs. This kind of what-if analysis can be useful to identify the impact of a security mitigation on the design level.

9.5 Tool Support

In this section we give a quick overview of the implemented tool and describe how to work with the tool using the provided user interface. We show how to create a mapping between a SecDFDs and its implementation and how to verify the implementation for security compliance with the SecDFD.

9.5.1 Implementation

The approach is implemented and packaged as a publicly available Eclipse plugin [275]. The architecture of our implementation is shown in Fig. 9.6. Reused components and external libraries are shown in dark gray. Components developed for [257] and adapted as part of this work are shown in light gray. Entirely new components are shown in white. Our implementation is structured according to the two main contributions of this work. First, we have the semi-automated creation of mappings realized in the component `Mapping`, and second, the security compliance checks realized in the `SecurityChecks` component.

Semi-automated mappings. For the creation of mapping suggestions we implemented the name matches and the patterns shown in Section 9.3 in hand written-java code. The implementation leverages an existing implementation for modeling SecDFDs using an Xtext DSL with editor support [248]. Also, we use an existing plugin for generating the Program Model from Java source code [13]. The SecDFD and the PM are accessed through the Java APIs provided by the components realizing the models.

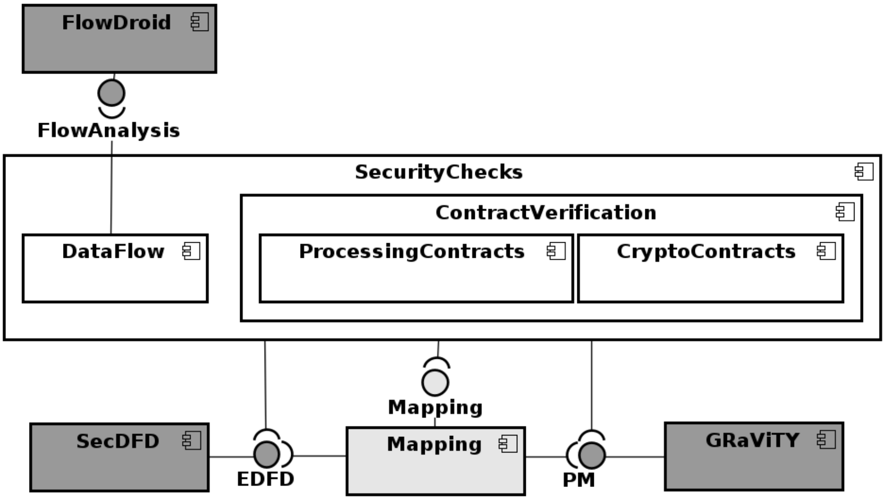


Figure 9.6: Architecture of the implementation

Using the Eclipse API, an integration into the Java source code editor is provided. For working with the SecDFD the textual editor is provided by the `SecDFD` component. In addition, we provide a graphical Editor based on the Sirius framework.[3] For showing the proposed mappings to the user, we registered a view in the Eclipse IDE. As a single system is usually described in multiple SecDFDs, we extended the implementation of this view to support multiple SecDFDs at the time. Details on how the user interacts with our implementation are presented in Section 9.5.2. Created mappings can be accessed through a wizard that shows all SecDFDs within a project as well as all existing mappings.

Access to the mappings is provided to other components though a `Mapping` interface, e.g., for the verification of SecDFD contracts. This interface allows to query the mappings in both directions, for mappings to a given SecDFD element and mappings to PM elements for a single or multiple SecDFDs.

Security checks. The implementation of the security compliance checks is following the structure of Section 9.4 and is separated into two components. One component for performing optimized data flow analyses (`DataFlow`) and one for the verification of the contracts specified in a SecDFD (`ContractVerification`).

In this work, we perform the data flow analysis using FlowDroid [96], a state-of-the-art taint analyzer for Android applications, but also applicable to Java programs. The 2.7.1 release of FlowDroid was obtained from [276] and is imported as a library in our plugin.

FlowDroid raises an alarm if and only if an object flows from a predefined list of *source* methods (i.e., these objects are tainted) into *sink* methods (i.e., they violate the security policy). The sources and sinks must be identified and are passed as parameters to the analyzer. To simply the analysis, FlowDroid relies on capabilities of the Soot compiler framework [277] which converts Java

[3] https://www.eclipse.org/sirius/

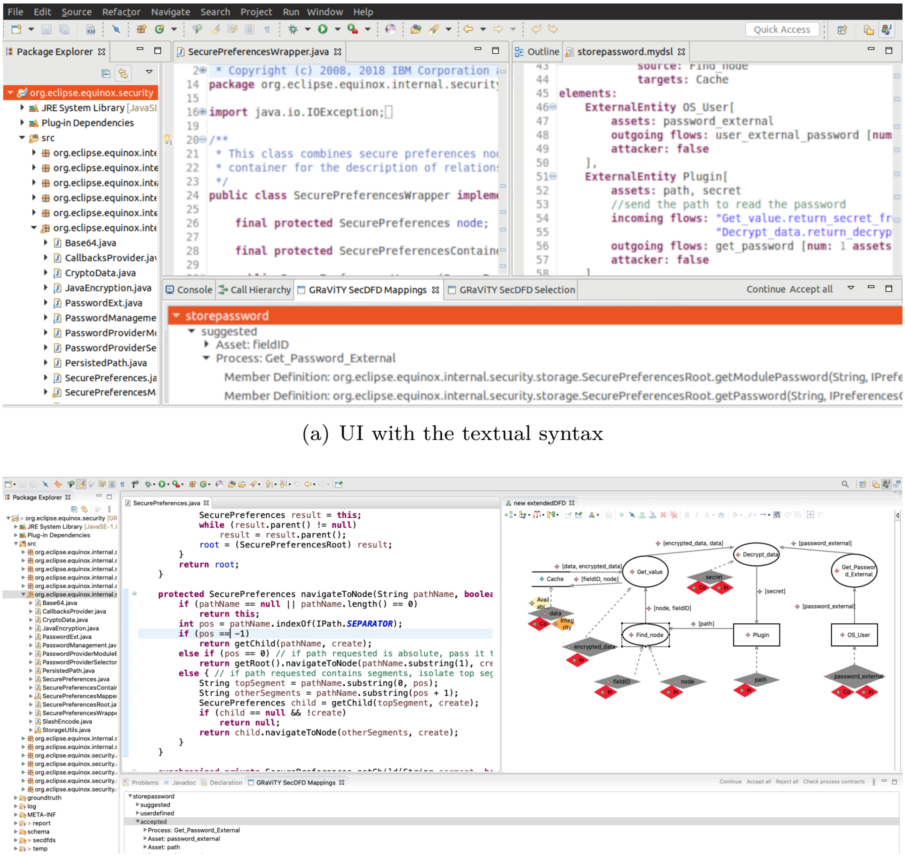
bytecode into the Jimple [278] intermediate code representation. This makes the analysis in FlowDroid precise as it is flow-sensitive (the call graph is aware of the order of statements) and context-sensitive (the call graph is enriched with the context of the callees). In addition, the Jimple representation is able to handle Java reflection, but only for reflective calls where the types of all referenced classes are known. The analysis in FlowDroid is also object-sensitive (i.e., the call graph distinguishes method invocations on different object instances) since it uses access paths as taint abstractions. In general, taint analyzers consider only explicit flows for performance reasons [279], but FlowDroid also supports tracking implicit flows and shows high performance results on benchmarks (86% precision and 93% recall on DroidBench [96]). We refer the interested reader to [280] for more details. The `DataFlow` component of our implementation executes FlowDroid over its Java API. Following Section 9.4.2, we execute FlowDroid for every asset in the SecDFD taking its set of allowed sinks and possible sources into account.

The contract verification is again split into two sub-components. One for the verification of the forward and join contracts (`ProcessingContracts`) and one for the verification of the encrypt and decrypt contracts (`CryptoContracts`). In both sub-components we implemented the checks as introduced in Section 9.4.1 using hand-written Java code.

9.5.2 Using the Tool

The target audience of the tool are software developers with training in the principles of software architecture. After the installation of the required packages, the program is started as a running Eclipse instance. First, the developers import the desired Java project. Second, they manually create one or several SecDFDs for representing the high-level architecture and security properties of the project. They can do so with a textual or graphical syntax (one can be generated from the other). Fig. 9.7 shows screenshots of the user interface after this step is completed. On the left hand side of the figures, users can see the Package Explorer. The top two windows are used for displaying the source code (left) and the SecDFD (right). The bottom windows are used for displaying and defining the mappings. Next, using context menu entries, the developers trigger the automated generation of a PM from the source code, and start the first iteration of the semi-automated process for mapping the SecDFD elements to source code elements (see Sect. 9.3).

At the start of each iteration, the developers are shown a list of suggested mappings. Since one DFD element is usually mapped to several program elements, the results are grouped by the DFD elements. For each DFD element, the list of mapped PM elements is shown, each with its path in the source code. The developers can interact with the tool by accepting, rejecting, and manually defining mappings. A suggested mapping is accepted or rejected with a right-click on the entry and selecting *accept* or *reject*, respectively. Once a mapping is accepted, corresponding in-line markers are created on the SecDFD and in the source code. Double-clicking a mapping will open the correct source file and navigate to the correct line in the file. Accepted mappings can always be rejected. If all the suggested mappings are correct, the developers can select *accept all*. Rejected mappings will never be suggested again. Manual



```
 * Copyright (c) 2008, 2018 IBM Corporation a
package org.eclipse.equinox.internal.security

import java.io.IOException;

/**
 * This class combines secure preferences noc
 * container for the description of relation
 */
public class SecurePreferencesWrapper impleme

    final protected SecurePreferences node;

    final protected SecurePreferencesContaine
```

```
            source: Find_node
            targets: Cache
elements:
    ExternalEntity OS_User[
        assets: password_external
        outgoing flows: user_external_password [num
        attacker: false
    ],
    ExternalEntity Plugin[
        assets: path, secret
        //send the path to read the password
        incoming flows: "Get_value.return_secret_fr
                        "Decrypt_data.return_decryp
        outgoing flows: get_password [num: 1 assets
        attacker: false
    ]
```

storepassword
- suggested
  - Asset: fieldID
  - Process: Get_Password_External
    - Member Definition: org.eclipse.equinox.internal.security.storage.SecurePreferencesRoot.getModulePassword(String, IPrefe
    - Member Definition: org.eclipse.equinox.internal.security.storage.SecurePreferencesRoot.getPassword(String, IPreferences(

(a) UI with the textual syntax

```
        SecurePreferences result = this;
        while (result.parent() != null)
            result = result.parent();
        root = (SecurePreferencesRoot) result;
    }
    return root;
}

protected SecurePreferences navigateToNode(String pathName, boolea
    if (pathName == null || pathName.length() == 0)
        return this;
    int pos = pathName.indexOf(IPath.SEPARATOR);
    if (pos == -1)
        return getChild(pathName, create);
    else if (pos == 0) // if path requested is absolute, pass it t
        return getRoot().navigateToNode(pathName.substring(1), cre
    else { // if path requested contains segments, isolate top seg
        String topSegment = pathName.substring(0, pos);
        String otherSegments = pathName.substring(pos + 1);
        SecurePreferences child = getChild(topSegment, create);
        if (child == null && !create)
            return null;
        return child.navigateToNode(otherSegments, create);
    }
}
```

storepassword
- suggested
- userdefined
- accepted
  - Process: Get_Password_External
  - Asset: password_external
  - Asset: path

(b) UI with the graphical syntax

Figure 9.7: Screenshots of the UI in Eclipse

Table 9.1: Projects considered in the evaluation

	source code			DFD
project	lloc	classes	methods	elements
jpetstore	1,221	17	277	47
ATM simulation	2,290	57	225	85
Eclipse Secure Storage	2,900	39	330	41
CoCoME	4,786	120	512	44
iTrust	28,133	423	3,691	31

definition works by right-clicking and selecting *Map Selection to SecDFD* on source code elements. At the end of the iteration, developers can either stop or select *continue* to trigger a new search refining the present mapping.

Finally, the developers can execute security compliance checks by pressing a button. The contract violations and leaks identified by FlowDroid are presented to the user with error and warning markers on the SecDFD model. At any moment, the developers can extend the list of project-specific methods signatures for cryptographic operations, and execute the checks again. Similar to manual definition, they can right-click the source code elements and select the appropriate menu item.

9.6 Evaluation

The evaluation of our approach has three parts. First, we conducted an experiment to evaluate the automated mapping creation (Section 9.6.1). Next, we conducted experiments to evaluate the verification of SecDFD contracts in implementation (Section 9.6.2), and the optimization of data flow analysis by extracting project-specific sources and sinks from SecDFDs (Section 9.6.3).

Table 9.1 depicts the characteristics of five open source Java projects used in our studies.

Jpetstore [281]. This is a web application built on top of MyBatis 3, Spring and the Stripes Framework. This is an example with very few classes, implementing the basic functionalities of a web store. In principle, the users are able to create their accounts, browse, and order goods online. Jpetstore has been designed as minimal demonstration application for MyBatis, which should have a good design and documentation. The developers tried to strictly follow the MVC pattern.

ATM simulation [282]. This is a simulation for an ATM machine developed for academic purposes. The ATM simulation implements the main procedure of a control system. Upon start-up a new session is initiated, and the users are able to insert their card and PIN number. The session continues upon a correct PIN entry, and provides the users with the option of a withdrawal, deposit, balance inquiry, and money transfer. After a completion of desired transactions, the ATM returns the card and optionally prints the receipt.

Eclipse Secure Storage [258]. As described in Section 9.2, Eclipse Secure Storage is used for ensuring secure storage and management of sensitive data within the developer's Eclipse workspace. The secure storage allows for plugins to authenticate and have controlled access to workspace resources.

CoCoME [283]. CoCoMe is a platform for collaborative empirical research on information system evolution [284]. This platform helps engineers manage different aspect of software evolution, such as the system life-cycle, versioning artifacts, and comprehensive evolution scenarios. The implemented system is a cash register.

iTrust [285]. This example is a web application for hospitals which allows the hospital's staff to manage medical records of patients, based on 55 use cases. The example originally stems from a course project, has been maintained by the Realsearch research group at North Carolina State University, and was used as an evaluation example in research papers before [286]. Detailed requirements describing different activities are available [285]. However, the available requirements and use cases mostly describe very simple tasks and only a few of them are realized in the implementation.

9.6.1 Evaluation of Mappings

The purpose of this study was to evaluate the correctness of the suggested mappings. In what follows, we briefly describe the design of the experiment, the projects, and the results.

Design of study. We conduct this experiment with all five open source projects from Table 9.1. To evaluate the correctness of the suggested mappings, we set up an experiment to compare a ground truth of manually created mappings with the generated mappings for each of the five considered projects. The iterative approach involves the user to guide the generation of mappings in the desired direction. As per this design choice, we intentionally investigate the correctness of the automated mappings and the impact of the user separately. Consequently, the evaluation aims to answer the following research questions.

RQ1. What is the correctness of the automated mappings generated by the plugin? We measured correctness in terms of *precision* and *recall* (dependent variables). Conventionally, precision ($TP/(TP + FP)$) is measured as a ratio between the true positives (i.e., correct mappings) and all generated mappings (including the false mappings). A true positive TP is a correct mapping between the source code and the DFD element which is listed in the ground truth. A false positive FP is a mapping between the source code and DFD element that is not listed in the ground truth. Recall ($TP/(TP + FN)$) is measured as a ratio between the true positives and all correct mappings (including the overlooked mappings). A false negative FN is a mapping between the source code and the SecDFD element which is present in the ground truth, but has not been identified.

RQ2. What is the impact of the user on the correctness of mappings? The implementation automatically derives trivial mappings from the user defined mappings, raising the recall before a new iteration starts. Therefore, the impact of the user defined mappings is measured as the difference in recall before, and after the added mappings.

Execution. The experiment was executed by the first and second author. The authors worked on the projects individually and compared their results at each step. First, the authors created the SecDFDs for all five projects models manually. To this aim, the authors inspected all available documentation (including the source code) and reverse engineered a high-level architecture. Second, a ground truth was created for each SecDFD by following the execution

Table 9.2: Results of the mapping after each iteration

		automated		manual			
project	it.	precision[%]	recall[%]	accept+u	($\sum$)	reject	recall[%](Δ)
jpetstore	1	56.1	51.1	23 + 3	(26)	18	57.8 (+6.7)
	2	96.4	60.0	1 + 3	(30)	1	66.7 (+6.7)
	3	96.8	66.7	0 + 5	(35)	1	77.8 (+11.1)
	4	97.4	82.2	2 + 3	(40)	1	88.9 (+6.7)
	5	100	93.3	2 + 3	(**45**)	0	**100** (+6.7)
ATM simulation	1	72.0	40.0	18 + 3	(21)	7	46.7 (+6.7)
	2	67.6	51.1	2 + 5	(28)	11	62.2 (+11.1)
	3	70.5	68.9	3 + 5	(36)	11	80.0 (+11.1)
	4	76.6	80	0 + 4	(40)	13	88.9 (+8.9)
	5	95.5	93.3	2 + 3	**(45)**	2	**100** (+6.7)
Eclipse sec. storage	1	73.0	90.5	40 + 1	(41)	14	92.9 (+2.4)
	2	67.7	**100**	1 + 0	(**42**)	12	—
CoCoME	1	27.9	77.3	17 + 1	(18)	44	81.8 (+4.5)
	2	86.4	90.5	1 + 1	(20)	2	90.9 (+0.4)
	3	90.9	83.3	0 + 2	(**22**)	4	**100** (+16.7)
iTrust	1	23.5	80.0	8 + 1	(9)	26	90.0 (+10.0)
	2	81.8	90.0	0 + 1	(**10**)	2	**100** (+10.0)

of the modeled scenarios and manually mapping the executed methods and transferred data to the processes and assets of the according step. The ground truth is a JSON file with a list of correspondence mappings between the elements of the SecDFD and a uniquely identifiable location of the source code element. Third, the implemented plugin was used to find the automated mappings in several iterations. Each iteration included accepting, rejecting the automated mappings, and defining mappings manually by highlighting elements in the source code and specifying the corresponding SecDFD elements. After each iteration the precision and recall of the automated mappings were logged.

Results. This study shows promising results for guiding the user in the discovery of compliance violations. In particular, Table 9.2 shows measurements of high precision and recall only after a few iterations for realistic Java projects. Each iteration consists of an automated, and a manual (user input) phase. We present the precision and recall for the automatically suggested mappings in each iteration. We also depict the amount of manually accepted, user defined, the sum of all accepted and user defined, rejected mappings, and the impact of the user defined mappings on recall (in that order). Notice that the later iterations make use of the manually defined mappings.

RQ1. We start by reporting the correctness of the automated mappings in the first iteration. The average precision of the first iteration is 50.5%. On average, the recall of the first iteration is 69.8%. Yet, both the precision and the recall increase after the first iteration. On average, the final precision and recall of the automated phase are very good (87.2% and 92%, respectively).

The average difference between the recall of the second iteration and the the user-impacted recall of the first iteration (last column in Table 9.2) is 4.5%. This means that on average, the automated search was able to increase the recall between the first and second iteration by 4.5%. On the other hand, the average difference between the user-impacted recall of the second iteration and the recall of the third iteration is minimal. This means that, the automated search was not able to increase the recall significantly between the second and third iteration.

RQ2. On average, the user accepted less (7) mappings then they rejected (9.6), and defined only 2.6 mappings manually. However, in three cases (jpetstore, ATM simulation, Eclipse Secure Storage) the user accepted more mappings then rejected. This means that the user could quickly scan the suggested mappings and eliminate the ones that are obviously wrong. Overall, adding a few mappings manually resulted in a more fruitful next iteration. For instance, adding three mappings manually in the first iteration of evaluating the ATM simulation resulted in two new correct mappings (see accepted mappings of the second iteration).

On average, the user impact on the recall was an increase of 7.9%. This means that the users were indeed able to guide the discovery of compliance violations. Further, the users had a larger impact on increasing the recall in later iterations compared to the automated search (7.9% vs 4.5%). Notice, that on average 75% of all correct mappings (TP) are suggested to the user and do not have to be manually defined.

9.6.2 Evaluation of the SecDFD Contract Verification

In this section, we evaluate if the proposed contract checks (Section 9.4.1) can effectively detect convergence, absence and divergence between the planned security properties and the implemented security mechanisms.

Design of study. In this part of the evaluation, we focus on the effectiveness of the SecDFD contract verification to answer the following research question.

RQ1. How effective is the proposed approach in the verification of contracts? It is important to evaluate if the proposed checks can effectively be used in the context of realistic projects. To this aim, we have used open source Java projects, as opposed to illustrative projects. Further, as we are interested in the effectiveness of the proposed compliance checks, we execute the evaluation for all process contracts, encrypt, decrypt, forward, and join. We evaluate the approach with perfectly compliant SecDFDs (i.e., verification results only include convergences, and there are no absence or divergence violations) and with SecDFDs with injected process contracts. In case of the fully compliant SecDFDs, all the detected compliance violations are false positives (FPs). Injecting the process contracts allows us to measure expected compliance violations (e.g., an absence of a join contract), which we mark as true positives (TPs). If the expected compliance violation is not found (according to the injected contract), we mark it as a false negative (FN). Finally, if we find unexpected compliance violations we mark them as false positives (FPs). As a term of measure, we adopt the well-understood precision ($TP/(TP + FP)$) and recall ($TP/(TP + FN)$) of detected compliance violations.

Execution. As subjects of this evaluation we use two subjects from the introduced test corpus, the Eclipse secure storage and iTrust. For both projects, we created one additional SecDFD. In what follows, we refer to the new SecDFDs as Eclipse 2 and iTrust 2. The two SecDFDs created for the study in Section 9.6.1 are Eclipse 1 and iTrust 1. As the created SecDFDs (all four) have been reverse engineered from the implementations, these are perfectly compliant.

First, we apply the contract verification to the two projects. We expect to detect no divergences or absences between the SecDFD and the implementation.

Afterward, we inject violations into the systems and check if these are detected. The violations are injected by adding random contracts to the SecDFDs that are not implemented. After every injection, we execute the contract verification and check if the expected violation has been detected, if additional false alarms have been raised, or if expected convergences are not detected any longer. We generate injections of all contract types (encrypt, decrypt, forward, and join). Regardless of the contract type, we inject all possible contracts that have not been specified on the initial SecDFD.

New encrypt and decrypt contracts can be injected independently of each other. An encrypt contract can be injected to every process that has no encrypt contract in the initial SecDFD and a decrypt contract to every process that has no decrypt contract. Accordingly, it can happen that we inject a decrypt contract to a process that has already an encrypt contract and the other way around.

For the injection of forward and join contracts, we inject for every process of a SecDFD all possible contracts that are not already specified. To do so, we calculate all possible combinations with one outgoing flow. To calculate the combinations we consider all incoming and outgoing assets. For instance, for a process with two incoming and two outgoing assets (and no specified forward, or join contract), we inject 6 possible contracts. Every incoming asset can be forwarded to every outgoing asset (4 forward contracts) and the pair of incoming assets can be joined with both outgoing assets as target (2 join contracts). If a combination is equivalent to an existing contract, it is omitted.

Results. The results of the evaluation are in favor of using our approach to execute security compliance checks between design and implementation.

For the execution of the verification on the fully compliant SecDFDs, we achieved 100% precision and recall. But, the effectiveness of the proposed contracts must also be studied in the context of imperfectly mapped SecDFDs. In what follows, we discuss the effectiveness of the approach in detecting absences of specified contracts. Tables 9.3 and 9.4 depict the results of the contract verification based on the injected contracts. We show the results per SecDFD and overall.

For evaluating the verification of encrypt and decrypt contracts, we injected 200 additional encrypt and decrypt contracts into the SecDFDs. Most injected contracts (except 11) were correctly detected as absent. The 11 undetected absent contracts belong to the same SecDFD (of the iTrust project). After investigating them, we noticed that all of them have been injected into processes that already have a encrypt or decrypt contract. The reason for this defect is that the project-specific specified signature (in the list of well-known cryptographic operations) for encryption is also specified for decryption. As iTrust uses a crypto-function on which a parameter is used for specifying whether a encryption or decryption should be performed, this is a correct classification. Since, we only check for at least one method call for encrypt/decrypt, we can not detect an absence in this particular case.

To evaluate the forward and join checks we injected 232 contracts into the

SecDFDs. In contrast to the cryptographic contracts verification, the results presented in Table 9.4 paint a more diverse picture. On the one hand, the processing contracts verification reaches a very good precision (98.21% and 87.01%) and recall (70.51% and 82.71%) on the iTrust project. On the other, the verification performs below par on the Eclipse secure storage project. In addition, there is a huge difference between the two SecDFDs on the Eclipse secure storage.

In particular, the verification did not work for the SecDFD shown in Fig. 9.1(b) (Eclipse 1). There are two reasons for this poor performance.

First, external entities are not part of the system and can not be mapped to elements from the system. For example, the external entity *Plugin* in Fig. 9.1(b) represents an arbitrary plugin installed into the Eclipse instance that is unknown to the Eclipse secure storage. This arbitrary plugin accesses the secure storage using a Java API specified on implementation level. Similarly, the data can be stored in a cloud, to which access is controlled via an API. In such cases we attempt at guessing possible incoming flows by considering, e.g., every parameter of the methods mapped to a process as possible source but also all returns of called methods that have not been mapped to any process. For instance, the *Get_value* process (Eclipse 1) is heavily interacting with an external entity and data store which results in very many guesses weakening the results.

Second, despite the reduction when extracting flows (described in Section 9.4.1), the overlapping asset types caused both FPs and FNs. In example, this communication of *Get_value* is implemented by mainly using assets whose mappings are overlapping (mainly strings). In general, representing sensitive objects with string values is prevalent in Eclipse secure storage. This also effected the performance of the processing contracts verification on the second SecDFD (Eclipse 2). Yet, the verification still achieves a recall and precision of 50%. This happened because the asset types of injected contracts overlapped with the asset types of the implemented contracts. For instance, consider two existing and fulfilled forwards of assets that are both mapped to the type *String*. On Fig. 9.1(b) for instance, these are the forward of *id* on the *Get_value* process and the forward of the *data* to *encr. data.* [4] In addition to these expected forwards, there are some additional uses of strings that are not representing assets, e.g., a parameter representing a default value in the implementation of the *Get_value* process. Now we inject a join of *id* and *data* to *encr. data.*. As the default value is a guessed flow, we could easily ignore it before this injection but now it exactly contributes to the injected join contract and we have to report this contact as convergence. However, we cannot any longer report the forward of *data* as convergence as the flow pattern is now mapped to the injected join contract. Accordingly, we now report a false divergence. In this case, at least the user would have been warned about a violation but the information about the assets was not entirely correct.

As the iTrust project does not have as many overlapping asset-type mappings and the SecDFDs have less external entities, the results are much better for this subject. Again, the missed violations are mainly due to overlapping asset mappings.

Overall, the contract verification is fairly precise (80%) and reaches the recall of more than 65%. Generally, the contract verification works and is able to bridge the huge gap between early design models and concrete implementations.

[4] Note that the *Get_value* encrypts the *data* only if it is stored in plain, else it forwards it.

Table 9.3: Results of evaluating the cryptographic contracts verification

	Eclipse		iTrust		
	1	2	1	2	Overall
TPs	12	48	59	70	189
FPs	0	0	0	0	0
FNs	0	0	11	0	11
precision	100%	100%	100%	100%	**100%**
recall	100%	100%	84.28%	100%	**94.5%**

Table 9.4: Results of evaluating the processing contracts verification

	Eclipse		iTrust		
	1	2	1	2	Overall
TPs	1	29	55	67	152
FPs	0	28	1	10	39
FNs	14	29	23	14	80
precision	100%	50.88%	98.21%	87.01%	**79.58%**
recall	6.67%	50%	70.51%	82.71%	**65.52%**

Though, it suffers from overlapping mappings. Also, missing API specification of the system (i.e., issue of mapping external entities), has a negative impact on the performance of the contract verification.

9.6.3 Evaluation of Optimized Data Flow Analysis

The purpose of this study is to evaluate whether using our approach helps to reduce the number of false alarms raised by an existing data flow analyzer.

Design of study. We investigate the performance of an analysis with FlowDroid [96] initialized with project-specific sources and sinks. To this aim, we built three configurations of sources and sinks. Apart from the first configuration (Plain), we execute the analyzer *for each SecDFD asset* separately. This experiment was conducted with two projects from Table 9.1, namely, Eclipse Secure Storage [258] and iTrust [285]. To the best of our knowledge, both projects are free of data flow leaks. Therefore, all the reported leaks by the analyzer are by default labeled as false alarms (FPs). We pose one research question.

RQ1. To what extent can the mapped design model (with our approach) be used to reduce the number of false alarms raised by a data flow analyzer?

To answer the research question, we have set up three configurations of sources and sinks.

Plain. We execute the analyzer with the list of source signatures shipped with FlowDroid [79](herein Default sources) and sink signatures (herein Default sinks). Apart from Java method signatures, this list contains signatures of methods specific to Android source packages. We removed such signatures to avoid unnecessarily searching for them with FlowDroid. Note, that this reduced the list of source signatures from 18,077 to 1,229 and sink signatures from 8,315 to 1,310. As a result of this filtering, the Android SQL database API (SQLite) was also removed. To analyze Java projects, we manually added signatures from the Java SQL API to the above list of sources and sinks.

Partly Opt. We execute the analyzer (for each confidential asset) with project-specific source signatures (herein SecDFD sources) and Default sinks. The SecDFD sources are extracted per SecDFD asset, as described in Section 9.4. Note that the SecDFD sources are extracted independently, and therefore may not include any of the Default sources.

Fully Opt. We execute the analyzer (for each confidential asset) with SecDFD sources and without allowed sink signatures (herein SecDFD sinks). The list of allowed sink signatures is extracted per SecDFD asset, as described in Section 9.4. The SecDFD sinks are obtained by *removing* the allowed sink signatures from the Default sinks.

The results are compared in only terms of the number of FPs, as no actual leaks (TPs) exist in the analyzed projects. In addition, we measure the number of extracted project-specific source signatures, and the number of removed sink signatures. A false alarm (FP) is a detected leak with a *unique pair of source and sink method signatures*, regardless of the access path where the leak is detected. The rationale for counting unique signature pairs is that comparing access paths would be computationally expensive and not useful for the purpose of this study. For instance, consider an implementation of a function where the number of recursive calls depends on a conditional. In this case, at least two access paths (when the conditional evaluates to `true` and `false`) are detected. But the DFD does not specify such level of detail, thus we can not distinguish between the access paths of the detected data leaks. The false alarms are aggregated per SecDFD, to enable comparison with the Plain configuration.

As we execute the analysis for each SecDFD asset, we measure the project specific sources and sinks in the same manner. Specifically, to measure the number of project-specific sources we count each discovered source signature per SecDFD asset. Similarly, to observe the number of times we are able to remove an allowed sink, we count each signature which has been removed for a unique asset.

Listing 9.1: Configuration of FlowDroid used in this study

```
Infoflow result = new Infoflow("", false, null);
result.setSootConfig((options, config) -> {
config.setCallgraphAlgorithm(CallgraphAlgorithm.AutomaticSelection);
config.setImplicitFlowMode(ImplicitFlowMode.AllImplicitFlows);
config.setAliasingAlgorithm(AliasingAlgorithm.FlowSensitive);
config.setStopAfterFirstKFlows(100);
});
result.setTaintWrapper(new EasyTaintWrapper(Collections.emptyMap()));
return result;
```

Execution. Both projects used in this study include two SecDFDs, representing two different scenarios. Listing 9.6.1 shows how we configured FlowDroid for all our executions. This configuration was set-up to achieve the best performance and most conservative analysis, in accordance with the literature [280]. We configure FlowDroid to use the default call-graph construction algorithm (SPARK). In addition, we have enabled implicit flow tracking and flow-sensitive aliasing. Note that, without tracking implicit flows, Fully Opt. produces no false alarms, while Plain still reports many. Finally, we limit the static analysis to the projects, excluding third-party libraries (line 11 in Listing 9.6.1),

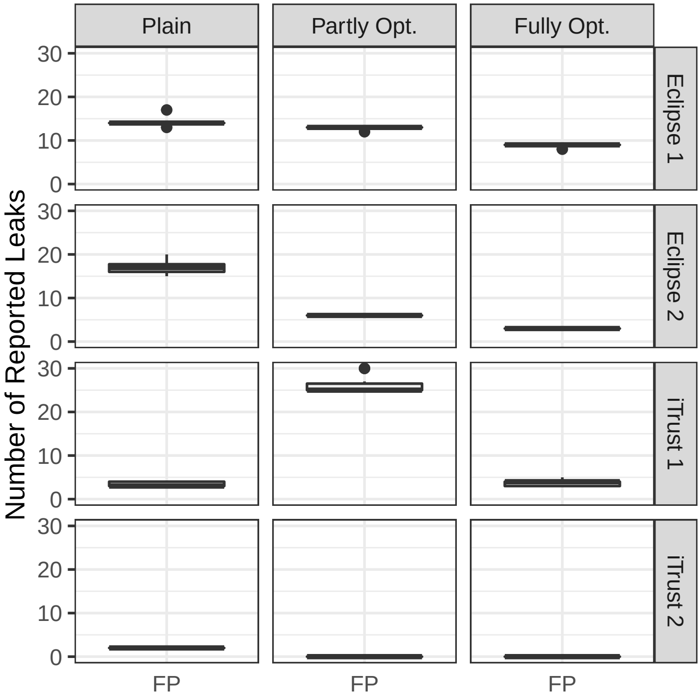


Figure 9.8: False alarms (FPs) raised by the analyzer after three configurations of sources and sinks per SecDFD (Eclipse Secure Storage on top, iTrust on bottom)

and stop the analyzer after identifying 100 leaks per run. We have implemented and executed the experiments using the JUnit Plugin Test framework with a limit of 6 GB of memory consumption (for each execution of the analyzer). The amount of allowed memory and the maximum number of identified leaks were determined empirically. We have executed random parts of the experiment with different configurations repeatedly and didn't get different results.

Results. Fig. 9.8 shows the false alarms raised by the analyzer after three configurations per SecDFD model. The average number of false alarms is aggregated per project in Table 9.5 and the change in the number of false alarms is presented. The main takeaway of the evaluation is that using our approach we were able to a) extract project-specific sources of secret data, and b) reduce the number of false alarms (up to 62%) raised by the data flow

Table 9.5: Average false alarm reduction for the different configurations (aggregated per project)

Configuration	FPs on Eclipse		FPs on iTrust		Overall
Plain	15.65		2.7		9.18
Partly Opt.	9.45	(↓ 60%)	13.1	(↑ 485%)	11.28
Fully Opt.	5.95	(↓ 37%)	1.9	(↓ 85%)	3.93
Total		(↓ **62**%)		(↓ 30%)	(↓ **57**%)

analyzer. First we discuss the reduction with only project-specific sources. Second we discuss the reduction with removing allowed sinks.

RQ1. Our measurements from the Partly Opt. configuration show that deriving project-specific sources from the SecDFD is possible and can reduce the number of FPs. For instance, in case of Secure Storage we achieved an average 60% reduction of false alarms (Table 9.5). However, adding project-specific sources can also lead to a rise in false alarms (as observed on iTrust). The number of project-specific sources is realistic considering the project size (e.i., 11 for Secure Storage and 10 for iTrust). In addition, the project-specific source methods are in fact accessing sensitive resources (e.g., the `org.eclipse.equinox.internal.security.storage. SecurePreferences.get(String, String, SecurePreferenceContainer):String` is called when fetching the cashed confidential credentials). But, the derived sources depend heavily on the mappings. Since iTrust is implemented with the dynamic Java Server Pages, FlowDroid can not analyze the entire behavior of the program. Therefore, we are only able to reduce the number of FPs after removing allowed sinks.

We found that the number of FPs can be further reduced by removing allowed sinks from the list of sinks passed to the analyzer (Fully Opt. configuration). We have been able to remove 3 sinks (all from `java.lang` package) for Eclipse Secure Storage and 36 sinks (all from `java.sql` package) for the iTrust project. These sinks were included in the previous configurations, but were derived in this configuration as allowed for certain SecDFD assets. In particular, we observed a further 37% average reduction of FPs for the Eclipse Secure Storage project, when comparing the analysis results to the previous configuration (Partly Opt.).Compared to the first configuration (Plain), considering only project-specific sources and removing allowed sinks reduced the number of false alarms on average by 62%. As project-specific sources were hard to find for the iTrust project, we compare the analysis results to the initial configuration (Plain). Removing the allowed sinks in iTrust reduced the number of FPs on average by 30%.

9.7 Threats to Validity

Our evaluation is subject to a number of threats.

The main threat to *external validity* is our selection of samples, based on a limited number of open source projects, partially originating from a teaching context. Regarding the validity of the studies conducted to evaluate the security compliance checks, the open source projects do not contain well-known data flow leaks, thus we consider them secure. The rationale for our selection was the manual effort that was required for creating the ground truth of our technique, a full mapping between high-level DFD elements and low-level program elements. However, as a result, the generalizability of the results to larger project in other domains is limited. To mitigate this threat, the considered projects were chosen to be representative for realistic projects by providing a good documentation, including architectural information (such as, wikis, use cases, scenarios, requirements, state charts, and the like). The available documentation enabled building good design models, close to the intended architecture. Further, we partly mitigate this threat by experimenting with contract injections as part of our evaluation. We plan to extend the evaluation in the future to include a more comprehensive set of projects.

Regarding *internal validity*, the main threat of our evaluation is researcher bias. In absence of pre-existing ground truths and design models, the ground truth and design models for our evaluation were created manually by the authors, possibly introducing a risk of creating a biased result. To mitigate this threat, the ground truths and the design-level models were carefully discussed between all authors. The created models and ground truths are of similar size and complexity and are available online [275].

With respect to *construct validity* we consider the threat of misinterpreting divergence, absence, and convergence compliance violations in the context of design-level models, implementation-level models, and violations detected by static code analysis. However, to the best of our knowledge, our interpretations are in-line with the existing literature [93].

9.8 Related Work

First, we discuss two most related works with respect to security compliance of DFDs, and leveraging specifications to optimize data flow analysis. Similar to our work, these approaches are difficult to classify as forward or reverse engineering solutions. Next, we position our work in the context of *forward* and *reverse engineering* literature.

More than a decade ago, Abi-Antoun et al. [71] proposed conformance checks between the implementation and DFDs. The authors automatically extract a DFD (i.e., the *source DFD*) from the implementation. Next, the user specifies a mapping (using Reflexion Models) between a manually created *high-level DFD* and the source DFD, which is then used to uncover inconsistencies. The notion of extracting the source DFD is similar to our extraction of the implemented data flows. In contrast to the mappings with Reflexion Models, our mappings are semi-automated using heuristics. Further, the security analysis in [71] is performed on the level of the DFD, while our security compliance checks are developed by means of static code analysis. To the best of our knowledge, this work is the sole attempt at implementing security compliance checks between the SecDFD and its implementation.

Recently, static code analysis techniques have been developed to assure GDPR compliance of the implemented systems with respect to privacy specifications [279, 287, 288]. Most relevant to our work, is the proposed approach by Ferrara et al. [279] which uses the privacy policy to fine tune and execute a taint analysis. The authors evaluate the approach by executing a prototype analysis on a benchmark application. Deriving the sources and sinks from the privacy policy is similar to our idea of maintaining allowed sinks for each SecDFD asset. But, the required GDPR policy needs to be specified on the level of implementation (e.g., concrete fields as sources, and API method signatures for sinks). In contrast, our approach can derive project-specific sources and allowed sinks from the design, and also performs security compliance checks with respect to the design model.

UML models have been extensively studied in the context of **forward**

engineering solutions for checking security compliance.

Muntean et al. [64] extend the UML statecharts with security annotations (such as source function, sink function, declassified parameter, etc.), generate the source code in C, and implement static checks (using the Smtcodan engine) to detect data flow violations. Similar to our work, the authors leverage security information from the design to execute a static analysis, and lift the detected violations back to the user (they display them with sequence diagrams). However, compared to DFDs, the gap between statecharts and source code is smaller (e.g., DFDs can not express conditional data flows, or sequence of data flows). Further, our approach with correspondence mappings can be used on existing projects (no code generation is necessary).

IFlow [65] is an approach for specifying and analyzing information flow properties in distributed Java applications. The proposed approach extends the UML model with information flow properties, and uses it to generate a Java code skeleton, and transform it to a formal model supporting an interactive theorem prover. The Java code skeleton (and manually completed program) can be checked for standard information flow properties, such as non-interference, using an existing framework (i.e., JOANA). Similar to this work, IFlow requires the developer to provide the security information in the model, and leverages an existing static analyzer. But, IFlow is model-driven and analyzes non-interference in a more formal setting.

Fourneret et al. [62] combine model-based security analyses using UMLsec [255] with the generation of security tests. Security properties are specified and verified on UML state machines. These models are then used to generate tests for the implemented system. In contrast to us the considered state machines have to be very close to the implementation. Further, Ramadan et al. [63] use model transformation to automatically generate security-annotated UML class models [255] from security-annotated BPMN models.

For the classical **reverse engineering** scenario from source code to UML class models, Peldszus et al. [289] propagate hand-crafted security annotations from source code to the corresponding elements in automatically extracted class models.

Scoria [66] is a semi-automated approach for extracting and analyzing the Owner Object Graph annotated with security properties (i.e., SecGraph) to find security flaws in the architecture. First, The SecGraph is extracted from a manually annotated implementation. Second, software architects can optionally refine the SecGraph with additional annotations. Finally, software architects can design queries to analyze the SecGraph. Similar to our work, Scoria is an iterative semi-automated approach analyzing security on abstracted code representation. However, our work does not rely on code annotations, and executes the security compliance checks by means of static analysis.

Jasser [70] recently proposed an approach for analyzing system behavior and detecting its discordance with a set of useful security rules. The security rules (modeled as Linear Time Logic (LTL) properties) are expressed with a controlled natural language for describing architectural constraints. The system behavior is extracted by means of dynamic analysis, using aspect-oriented programming. Finally, before the security rules can be executed, the source-level elements are manually mapped to the architectural elements. On a high-level, the idea of our work relates Jassers approach, in that, an abstracted representation of code is mapped to a higher-level model to analyze security compliance. In comparison, our approach supports an automated discovery of such mapping, and studies the compliance of *static* security properties in the implementation.

Manual security reviews can be aided by automated static (or hybrid) program analysis. Static Application Security Testing (SAST) [290] tools aim to analyze the program code of a software component and automatically report the violations to developers, removing the need for security experts reviewing large code bases. Our approach relates to such mechanisms in that it leverages static code analysis to evaluate security of an implemented system. But, the SAST analyzes security of the implementation, while our approach focuses on analyzing the compliance of implemented security to the intended (designed) security. Further, SAST tools need to still be configured by security experts, whereas our approach automatically derives project-specific sources and sinks from the SecDFD model.

Duarte et al. [291] propose to use context information of execution sequences for the extraction of labeled transition system models from source code. While the authors motivate their approach with the need for correspondence between models and code, they only discuss the possibility to analyze the models using existing tooling. The compliance checks introduced by Duarte et al. [291] are performed similarly to the checks developed in this work. In contrast, our approach supports compliance checks between models and code. Regarding the preparation for compliance checks, Duarte et al. reverse engineer models that can be checked or compared to existing models. In contrast, we recreate a mapping between existing models and their implementation. This already includes a comparison with the existing models.

Beyond the security scope of this work, conformance checking is generally a well-studied topic in model-driven engineering. Paige et al. [292] use meta-models as the common reference point to enable conformance checks between diagrams representing different views on a system. Diskin et al. [293] present a framework for global consistency checks of heterogeneous models based on constraints. By supporting the explicit specification of overlaps between the considered models, they avoid the need for a global meta-model. Expanding on this work, König and Diskin [294] improve the efficiency of this approach by supporting an early localization of relevant parts of the models whose consistency is to be checked. Reder and Egyed [295] propose an efficient approach to consistency checking based on predefined consistency rules. Estanol et al. [296] developed an approach to check the conformance of process implementation to UML and OCL models by translating them into petri-nets, and executing existing conformance checking techniques. However, none of these works address security compliance checking between design and its implementation.

9.9 Conclusion and Future Work

This work has introduced a novel approach for tackling the problem of automating the code-level verification of planned security mechanisms. In particular, we have developed a solution with tool support for executing security compliance checks between an abstract design model (the SecDFD) and its implementation (in Java). To this aim, we developed a user-in-the-loop approach for finding corresponding elements based on heuristically computed suggestions. Once defined, the correspondence mappings are leveraged for an automated security analysis of the implementation against the design. First, two types of security compliance checks are executed: a rule-based check for a set of cryptographic operations, and a local data flow check for data processing contracts specified in the model. Second, the mapped design is leveraged to initialize and execute a state-of-the-art data flow analyzer over the entire Java project. The results of the compliance checks (convergence, absence, and divergence) are lifted to the attention of the user via the user interface of our tool.

Our approach was evaluated with three studies on open source Java projects, focused on assessing the performance from different angles. First, our evaluation has shown a high precision (87.2%) of the automated suggestions of mappings. Second, the rule-based security compliance checks are very precise (100%) and rarely overlook implemented cryptographic operations (recall is 94.5%). In addition, the local data flow checks are fairly precise (79.6%), but may overlook some implemented flows (recall is 65.6%), due to the large gap between the design and implementation. Finally, our approach enables a project-specific data flow analysis with up to 62% less false alarms.

Regarding future improvements, we note that extending the SecDFD with strongly typed assets could improve the performance of the security compliance checks. Strongly typed SecDFD assets could be mapped to the implementation more precisely, which would make the local data flow checks cleaner. In addition, the missing mappings to the external entities could be better approximated by relying on parsed API specifications (e.g, JavaDoc). Finally, the evaluation of the security checks could be improved by including more open source projects, especially projects with well-known data leaks.

Bibliography

[1] C. Y. Jeong, S.-Y. T. Lee, and J.-H. Lim, "Information security breaches and it security investments: Impacts on competitors," *Information & Management*, vol. 56, no. 5, pp. 681–695, 2019.

[2] "UK's ICO fines British Airways a record £183M over GDPR breach that leaked data from 500,000 users," https://techcrunch.com/2019/07/08/uks-ico-fines-british-airways-a-record-183m-over-gdpr-breach-that-leaked-data-from-500000-users/, accessed: 2020-11-18.

[3] "Report: Data Breach in Biometric Security Platform Affecting Millions of Users," https://www.vpnmentor.com/blog/report-biostar2-leak/, accessed: 2020-11-18.

[4] "Report: Estimated 24,000 Android apps expose user data through Firebase blunders," https://www.comparitech.com/blog/information-security/firebase-misconfiguration-report/, accessed: 2020-11-18.

[5] G. McGraw, *Software security: building security in.* Addison-Wesley Professional, 2006, vol. 1.

[6] N. Daswani, C. Kern, and A. Kesavan, "Secure design principles," *Foundations of Security: What Every Programmer Needs to Know*, pp. 61–76, 2007.

[7] C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, and K. Togashi, "Secure Design Patterns," Carnegie-Mellon University Pittsburgh, Software Engineering Institute, Tech. Rep., 2009.

[8] S. Migues, J. Steven, and M. Ware, "Building security in maturity model 11 (BSIMM11)," https://www.bsimm.com, accessed: 2020-10-29.

[9] A. Shostack, *Threat Modeling: Designing for Security.* John Wiley & Sons, 2014.

[10] R. Scandariato, K. Wuyts, and W. Joosen, "A descriptive study of microsoft's threat modeling technique," *Requirements Engineering*, vol. 20, no. 2, pp. 163–180, 2015.

[11] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen, "A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements," *Requirements Engineering*, vol. 16, no. 1, pp. 3–32, 2011.

[12] R. Hebig, T. H. Quang, M. R. Chaudron, G. Robles, and M. A. Fernandez, "The Quest for Open Source Projects that Use UML: Mining GitHub," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS).* ACM, 2016, pp. 173–183.

[13] S. Peldszus *et al.*, "GRaViTY Program Model," 2020. [Online]. Available:

http://gravity-tool.org

[14] G. Macher, E. Armengaud, E. Brenner, and C. Kreiner, "A review of threat analysis and risk assessment methods in the automotive context," in *Proceedings of the International Conference on Computer Safety, Reliability, and Security.* Springer, 2016, pp. 130–141.

[15] K. Bernsmed and M. G. Jaatun, "Threat modelling and agile software development: Identified practice in four norwegian organisations," in *Proceedings of the International Conference on Cyber Security and Protection of Digital Services (Cyber Security).* IEEE, 2019, pp. 1–8.

[16] M. N. Anwar, M. Nazir, and A. M. Ansari, "Modeling security threats for smart cities: A stride-based approach," in *Smart Cities—Opportunities and Challenges.* Springer, 2020, pp. 387–396.

[17] J. Lee, S. Kang, and S. Kim, "Study on the smart speaker security evaluations and countermeasures," in *Advanced Multimedia and Ubiquitous Engineering.* Springer, 2019, pp. 50–70.

[18] C. Paule, T. F. Düllmann, and A. Van Hoorn, "Vulnerabilities in continuous delivery pipelines? a case study," in *Proceedings of the International Conference on Software Architecture Companion (ICSA-C).* IEEE, 2019, pp. 102–108.

[19] J. Sanfilippo, T. Abegaz, B. Payne, and A. Salimi, "Stride-based threat modeling for mysql databases," in *Proceedings of the Future Technologies Conference.* Springer, 2019, pp. 368–378.

[20] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer, "Stride-based threat modeling for cyber-physical systems," in *Proceedings of the PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe).* IEEE, 2017, pp. 1–6.

[21] M. Abomhara, M. Gerdes, and G. M. Køien, "A stride-based threat model for telehealth systems," *NISK Journal*, pp. 82–96, 2015.

[22] M. V. Dumitru, D. Dumitrescu, and C. Raiciu, "Can we exploit buggy p4 programs?" in *Proceedings of the Symposium on SDN Research*, 2020, pp. 62–68.

[23] M. A. Naagas and T. D. Palaoag, "A threat-driven approach to modeling a campus network security," in *Proceedings of the International Conference on Communications and Broadband Networking*, 2018, pp. 6–12.

[24] D. Magin, R. Khondoker, and K. Bayarou, "Security analysis of openradio and softran with stride framework," in *Proceedings of the International Conference on Computer Communications and Applications (ICCCN)*, vol. 38, 2015.

[25] M. S. Lund, B. Solhaug, and K. Stølen, *Model-driven risk analysis: the CORAS approach.* Springer Science & Business Media, 2010.

[26] C. Alberts, A. Dorofee, J. Stevens, and C. Woody, "Introduction to the octave approach," Pittsburgh, PA, Carnegie Mellon University, Tech. Rep., 2003.

[27] ——, "Octave-s implementation guide, version 1.0," Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Tech. Rep., 2005.

[28] R. A. Caralli, J. F. Stevens, L. R. Young, and W. R. Wilson, "Introducing octave allegro: Improving the information security risk assessment process," Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Tech. Rep., 2007.

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

[30] K. Buyens, B. De Win, and W. Joosen, "Empirical and statistical analysis of risk analysis-driven techniques for threat management," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES).* IEEE, 2007, pp. 1034–1041.

[31] J. Selin, "Evaluation of threat modeling methodologies," Master's thesis, JAMK University of Applied Sciences, https://www.theseus.fi/bitstream/handle/10024/220967/Selin_Juuso.pdf?isAllowed=y&sequence=2, 5 2019.

[32] D. Verdon and G. McGraw, "Risk analysis in software design," *IEEE Security & Privacy Magazine*, vol. 2, no. 4, pp. 79–84, 2004.

[33] D. S. Cruzes, M. G. Jaatun, K. Bernsmed, and I. A. Tøndel, "Challenges and experiences with applying microsoft threat modeling in agile development projects," in *Proceedings of the Australasian Software Engineering Conference (ASWEC).* IEEE, 2018, pp. 111–120.

[34] S. Mauw and M. Oostdijk, "Foundations of attack trees," in *Proceedings of the International Conference on Information Security and Cryptology*, vol. 3935. Springer, 2005, pp. 186–198.

[35] G. Sindre and A. L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements Engineering*, vol. 10, no. 1, pp. 34–44, 2005.

[36] K. Beckers, D. Hatebur, and M. Heisel, "A problem-based threat analysis in compliance with common criteria," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES).* IEEE, 2013, pp. 111–120.

[37] D. Hatebur and M. Heisel, "Problem frames and architectures for security problems," in *Proceedings of the International Conference on Computer Safety, Reliability, and Security (SAFECOMP).* Springer, 2005, pp. 390–404.

[38] L. Lin, B. Nuseibeh, D. Ince, and M. Jackson, "Using abuse frames to bound the scope of security problems," in *Proceedings of the International Conference on Requirements Engineering (RE).* IEEE, 2004, pp. 354–355.

[39] B. Schneier, "Attack trees," Dr Dobb's Journal, v.24, n.12, 1999.

[40] H. S. Lallie, K. Debattista, and J. Bal, "A review of attack graph and attack tree visual syntax in cyber security," *Computer Science Review*, vol. 35, p. 100219, 2020.

[41] "Sustainable Application Security microsofts new threat modeling tool," https://blog.secodis.com/2016/07/06/microsofts-new-threat-modeling-tool/, accessed: 2017-05-15.

[42] L. Sion, D. Van Landuyt, K. Yskout, and W. Joosen, "Sparta: Security & privacy architecture through risk-driven threat assessment," in *Proceedings of the International Conference on Software Architecture (ICSA).* IEEE, 2018.

[43] S. Seifermann, R. Heinrich, and R. Reussner, "Data-driven software architecture for analyzing confidentiality," in *Proceedings of the International Conference on Software Architecture (ICSA).* IEEE, 2019, pp. 1–10.

[44] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE

Press, 2013, pp. 662–671.

[45] B. J. Berger, K. Sohr, and R. Koschke, "Automatically extracting threats from extended data flow diagrams," in *Proceedings of the International Symposium on Engineering Secure Software and Systems.* Springer, 2016, pp. 56–71.

[46] J. B. Hong, D. S. Kim, C.-J. Chung, and D. Huang, "A survey on the usability and practical applications of graphical security models," *Computer Science Review*, vol. 26, pp. 1–16, 2017.

[47] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *Proceedings of the Symposium on Security and Privacy.* IEEE, 2002, pp. 273–284.

[48] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the Conference on Computer and Communications Security.* ACM, 2006, pp. 336–345.

[49] D. Xu and K. E. Nygard, "Threat-driven modeling and verification of secure software using aspect-oriented petri nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 265–278, 2006.

[50] C. Gerking and D. Schubert, "Component-based refinement and verification of information-flow security policies for cyber-physical microservice architectures," in *Proceedings of the International Conference on Software Architecture (ICSA).* IEEE, 2019, pp. 61–70.

[51] G. T. Leavens, T. Wahls, and A. L. Baker, "Formal semantics for sa style data flow diagram specification languages," in *Proceedings of the 1999 ACM Symposium on Applied Computing*, ser. SAC '99, 1999, pp. 526–532.

[52] P. G. Larsen, N. Plat, and H. Toetenel, "A formal semantics of data flow diagrams," *Form. Asp. Comput.*, vol. 6, no. 6, pp. 586–606, Dec. 1994.

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

[54] P. H. Nguyen, M. Kramer, J. Klein, and Y. Le Traon, "An extensive systematic review on the model-driven development of secure systems," *Information and Software Technology*, vol. 68, pp. 62–81, 2015.

[55] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14 516–14 542, 2017.

[56] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, and A. Sousa, "A model-driven traceability framework for software product lines," *Software & Systems Modeling*, vol. 9, no. 4, pp. 427–451, 2010.

[57] J. Jürjens, "Umlsec: Extending uml for secure systems development," in *Proceedings of the International Conference on The Unified Modeling Language.* Springer, 2002, pp. 412–425.

[58] ——, "Model-based security testing using umlsec: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 93–104, 2008.

[59] B. Best, J. Jurjens, and B. Nuseibeh, "Model-based security engineering of distributed information systems using umlsec," in *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE

Computer Society, 2007, pp. 581–590.

[60] J. Jürjens, "Using umlsec and goal trees for secure systems development," in *Proceedings of the Symposium on Applied Computing.* ACM, 2002, pp. 1026–1030.

[61] J. Jürjens and P. Shabalin, "Tools for secure systems development with uml," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 527–544, 2007.

[62] E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jurjens, and P. Yousefi, "Model-Based Security Verification and Testing for Smart-cards," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES).* IEEE, 2011, pp. 272–279.

[63] Q. Ramadan, M. Salnitri, D. Strüber, J. Jürjens, and P. Giorgini, "From Secure Business Process Modeling to Design-Level Security Verification," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS).* ACM/IEEE, 2017, pp. 123–133.

[64] P. Muntean, A. Rabbi, A. Ibing, and C. Eckert, "Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code," in *Proceedings of the International Conference on Software Quality, Reliability and Security-Companion (QRS-C).* IEEE, 2015, pp. 128–137.

[65] K. Katkalov, K. Stenzel, M. Borek, and W. Reif, "Model-driven development of information flow-secure systems with iflow," in *Proceedings of the International Conference on Social Computing.* IEEE, 2013, pp. 51–56.

[66] R. Vanciu and M. Abi-Antoun, "Finding Architectural Flaws using Constraints," in *Proceedings of the International Conference on Automated Software Engineering (ASE).* IEEE, 2013, pp. 334–344.

[67] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE, 2002, pp. 187–197.

[68] P. Olson and M. Randevik, "Secarchunit: Extending archunit to support validation of security architectural constraints," Master's thesis, Chalmers University of Technology, https://masterthesis.cms.chalmers.se/content/secarchunit-extending-archunit-support-validation-security-architectural-constraints, 4 2020.

[69] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, "Characterizing the usage, evolution and impact of java annotations in practice," *IEEE Transactions on Software Engineering*, 2019.

[70] S. Jasser, "Enforcing Architectural Security Decisions," in *Proceedings of the International Conference on Software Architecture (ICSA).* IEEE, 2020, pp. 35–45.

[71] M. Abi-Antoun, D. Wang, and P. Torr, "Checking threat modeling data flow diagrams for implementation conformance and security," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, 2007, pp. 393–396.

[72] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *Proceedings of the Symposium on Foundations of Software Engineering.* ACM, 1995, pp. 18–28.

[73] S. Charalampidou, A. Ampatzoglou, E. Karountzos, and P. Avgeriou, "Empirical studies on software traceability: A mapping study," *Journal of Software: Evolution and Process*, p. e2294, 2020.

[74] A. Velasco and J. Aponte, "Automated fine grained traceability links recovery between high level requirements and source code implementations," *ParadigmPlus*, vol. 1, no. 2, pp. 18–41, 2020.

[75] A. Act, "Health insurance portability and accountability act of 1996," *Public law*, vol. 104, p. 191, 1996.

[76] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[77] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[78] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[79] S. Arzt, S. Rasthofer, and E. Bodden, "SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks," University of Darmstadt, Tech. Rep. TUDCS-2013-0114, 2013.

[80] B. Kitchenham, S. Charters, D. Budgen, P. Brereton, M. Turner, S. Linkman, M. Jørgensen, E. Mendes, and G. Visaggio, "Guidelines for performing systematic literature reviews in software engineering," in *Technical report, Ver. 2.3 EBSE Technical Report. EBSE.* sn, 2007.

[81] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering.* ACM, 2014, p. 38.

[82] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering.* Springer, 2008, pp. 285–311.

[83] D. H. Jonassen, "Toward a design theory of problem solving," *Educational technology research and development*, vol. 48, no. 4, pp. 63–85, 2000.

[84] A. A. A. Jilani, A. Nadeem, T. hoon Kim, and E. suk Cho, "Formal representations of the data flow diagram: A survey," in *Proceedings of the Advanced Software Engineering and Its Applications (ASEA)*, 2008.

[85] D. M. Volpano and G. Smith, "A type-based approach to program security," in *Proceedings of the International Joint Conference Theory and Practice of Software Development*, 1997, pp. 607–621.

[86] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles." *JCS*, 2009.

[87] "CVE - Common Vulnerabilities and Exposures," Available from MITRE, 2020. [Online]. Available: https://cve.mitre.org

[88] "CWE - Common Weakness Enumeration," Available from MITRE, 2020. [Online]. Available: https://cwe.mitre.org

[89] S. Barnum, "Common attack pattern enumeration and classification (capec) schema description," *Cigital Inc, http://capec. mitre. org/documents/documentation/CAPEC_Schema_Descr iption_v1*, vol. 3, 2008.

[90] B. J. Berger, K. Sohr, and R. Koschke, "Extracting and analyzing the im-

plemented security architecture of business applications," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 285–294.

[91] B. Hoisl, S. Sobernig, and M. Strembeck, "Modeling and enforcing secure object flows in process-driven soas: an integrated model-driven approach," *Software & Systems Modeling*, vol. 13, no. 2, pp. 513–548, 2014.

[92] M. Frydman, G. Ruiz, E. Heymann, E. César, and B. P. Miller, "Automating risk analysis of software design models," *The Scientific World Journal*, vol. 2014, pp. 248–259, 2014.

[93] L. De Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[94] K. Goseva-Popstojanova and A. Perhinschi, "On the Capability of Static Code Analysis to Detect Security Vulnerabilities," *Information and Software Technology (IST)*, vol. 68, pp. 18–33, 2015.

[95] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static Code Analysis to Detect Software Security Vulnerabilities-does Experience Matter?" in *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2009, pp. 804–810.

[96] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[97] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[98] T. Antignac, R. Scandariato, and G. Schneider, "Privacy compliance via model transformations," in *Proceedings of the European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2018, pp. 120–126.

[99] "MS Windows NT kernel description," Node-RED: Low-code programming for event-driven applications, accessed: 2020-11-19.

[100] Z. Li, P. Liang, and P. Avgeriou, "Architectural technical debt identification based on architecture decisions and change scenarios," in *Proceedings of the Working Conference on Software Architecture (WICSA)*. IEEE, 2015, pp. 65–74.

[101] "Bsimm7," https://go.bsimm.com/hubfs/BSIMM/BSIMM7.pdf, (Accessed on 12/08/2017).

[102] J. Whittle, D. Wijesekera, and M. Hartong, "Executable misuse cases for modeling security concerns," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2008, pp. 121–130.

[103] C. Raspotnig and A. Opdahl, "Comparing risk identification techniques for safety and security requirements," *Journal of Systems and Software*, vol. 86, no. 4, pp. 1124–1151, 2013.

[104] C. Y. C. Cheung, "Threat modeling techniques," Faculty of Technology, Policy and Management, Delft University of Technology, Tech. Rep., Nov. 2016. [Online]. Available: http://www.safety-and-security.nl/uploads/cfsas/attachments/SPM5440%20%26%20WM0804TU%20-%20Threat%20modeling%20techniques%20-%20CY%20Cheung.pdf

[105] P. Torr, "Demystifying the threat modeling process," *IEEE Security & Privacy*, vol. 3, no. 5, pp. 66–70, 2005.

[106] "Owasp," https://www.owasp.org/index.php/Main_Page, (Accessed on 01/29/2018).

[107] "Octave — cyber risk and resilience management — the cert division," https://www.cert.org/resilience/products-services/octave/, (Accessed on 01/29/2018).

[108] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, 2007.

[109] "Core rankings portal - computing research & education," http://www.core.edu.au/conference-portal, (Accessed on 12/04/2017).

[110] "Excellence in research for australia — australian research council," http://www.arc.gov.au/excellence-research-australia, (Accessed on 01/25/2018).

[111] X. Yuan, E. B. Nuakoh, I. Williams, and H. Yu, "Developing abuse cases based on threat modeling and attack patterns," *Journal of Software (JSW)*, vol. 10, no. 4, pp. 491–498, 2015.

[112] P. Wang, K.-M. Chao, C.-C. Lo, and Y.-S. Wang, "Using ontologies to perform threat analysis and develop defensive strategies for mobile security," *Information Technology and Management*, vol. 18, no. 1, pp. 1–25, 2017.

[113] I. Williams, "Evaluating a method to develop and rank abuse cases based on threat modeling, attack patterns and common weakness enumeration," Ph.D. dissertation, North Carolina Agricultural and Technical State University, 2015.

[114] J. Taylor, *Introduction to error analysis, the study of uncertainties in physical measurements.* University Science Books, 1997.

[115] F. Taylor, "Paul e. meehl: Clinical versus statistical prediction. a theoretical analysis and a review of the evidence (book review)," *The International Journal of Psycho-Analysis*, vol. 37, p. 490, 1956.

[116] L. M. Osbeck and B. S. Held, *Rational intuition: Philosophical roots, scientific investigations.* Cambridge University Press, 2014.

[117] D. Kahneman, *Thinking, fast and slow.* Macmillan, 2011.

[118] D. Kahneman and G. Klein, "Conditions for intuitive expertise: a failure to disagree," *American psychologist*, vol. 64, no. 6, p. 515, 2009.

[119] R. M. Dawes, "The robust beauty of improper linear models in decision making," *American psychologist*, vol. 34, no. 7, p. 571, 1979.

[120] C. B. Haley, R. C. Laney, and B. Nuseibeh, "Deriving security requirements from crosscutting threat descriptions," in *Proceedings of the International Conference on Aspect-Oriented Software Development.* ACM, 2004, pp. 112–121.

[121] Y. Chen, B. Boehm, and L. Sheppard, "Value driven security threat modeling based on attack path analysis," in *Proceedings of the Annual Hawaii International Conference on System Sciences (HICSS).* IEEE, 2007, pp. 280a–280a.

[122] T. Hilburn, M. Ardis, G. Johnson, A. Kornecki, and N. R. Mead, "Software assurance competency model," Carnegie-Mellon University Pitts-

burgh, Software Engineering Institute, Tech. Rep., 2013.

[123] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 2009.

[124] T. Abe, S. Hayashi, and M. Saeki, "Modeling security threat patterns to derive negative scenarios," in *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1. IEEE, 2013, pp. 58–66.

[125] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Attack–defense trees," *Journal of Logic and Computation*, vol. 24, no. 1, pp. 55–87, 2014.

[126] K. Beckers, I. Côté, S. Faßbender, M. Heisel, and S. Hofbauer, "A pattern-based method for establishing a cloud-specific information security management system," *Requirements Engineering*, vol. 18, no. 4, pp. 343–395, 2013.

[127] M. S. Lund, B. Solhaug, and K. Stølen, "A guided tour of the coras method," in *Model-Driven Risk Analysis.* Springer, 2011, pp. 23–43.

[128] O. El Ariss and D. Xu, "Modeling security attacks with statecharts," in *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS.* ACM, 2011, pp. 123–132.

[129] C. O. Encina, E. B. Fernandez, and A. R. Monge, "Threat analysis and misuse patterns of federated inter-cloud systems," in *Proceedings of the European Conference on Pattern Languages of Programs.* ACM, 2014, p. 13.

[130] G. Elahi and E. Yu, "A goal oriented approach for modeling and analyzing security trade-offs," *Conceptual Modeling-ER 2007*, pp. 375–390, 2007.

[131] H. Mouratidis and P. Giorgini, "Secure tropos: a security-oriented extension of the tropos methodology," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 02, pp. 285–309, 2007.

[132] G. Elahi, E. Yu, and N. Zannone, "A vulnerability-centric requirements engineering framework: analyzing security attacks, countermeasures, and requirements based on vulnerabilities," *Requirements Engineering*, vol. 15, no. 1, pp. 41–62, 2010.

[133] H. Mouratidis, P. Giorgini, and G. Manson, "Modelling secure multiagent systems," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems.* ACM, 2003, pp. 859–866.

[134] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, 2008.

[135] S. T. Halkidis, N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Architectural risk analysis of software systems based on security patterns," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 3, pp. 129–142, 2008.

[136] J. McDermott, "Abuse-case-based assurance arguments," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC).* IEEE, 2001, pp. 366–374.

[137] J. McDermott and C. Fox, "Using abuse case models for security requirements analysis," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC).* IEEE, 1999, pp. 55–64.

[138] A. Van Lamsweerde *et al.*, “Engineering requirements for system reliability and security,” *NATO Security Through Science Series D-Information and Communication Security*, vol. 9, p. 196, 2007.

[139] A. Van Lamsweerde, “Elaborating security requirements by construction of intentional anti-models,” in *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE Computer Society, 2004, pp. 148–157.

[140] A. Van Lamsweerde and E. Letier, “Handling obstacles in goal-oriented requirements engineering,” *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 978–1005, 2000.

[141] P. Karpati, G. Sindre, and A. Opdahl, “Visualizing cyber attacks with misuse case maps,” *Requirements Engineering: Foundation for Software Quality*, pp. 262–275, 2010.

[142] P. Karpati, A. L. Opdahl, and G. Sindre, “Harm: Hacker attack representation method,” in *Proceedings of the International Conference on Software and Data Technologies.* Springer, 2010, pp. 156–175.

[143] L. Liu, E. Yu, and J. Mylopoulos, “Security and privacy requirements analysis within a social setting,” in *Proceedings of the International Requirements Engineering Conference.* IEEE, 2003, pp. 151–161.

[144] T. Li, E. Paja, J. Mylopoulos, J. Horkoff, and K. Beckers, “Security attack analysis using attack patterns,” in *Proceedings of the International Conference on Research Challenges in Information Science (RCIS).* IEEE, 2016, pp. 1–13.

[145] I. A. Tøndel, J. Jensen, and L. Røstad, “Combining misuse cases with attack trees and security activity models,” in *Proceedings of the International Conference on Availability, Reliability, and Security (ARES).* IEEE, 2010, pp. 438–445.

[146] M. Jackson, *Problem frames: analysing and structuring software development problems.* Addison-Wesley, 2001.

[147] “Owasp,” https://www.owasp.org/index.php/Main_Page, (Accessed on 11/09/2017).

[148] P. Saitta, B. Larcom, and M. Eddington, “Trike v. 1 methodology document [draft],” https://www.octotrike.org/papers/Trike_v1_Methodology_Document-draft.pdf, 2005.

[149] N. Medvidovic and R. N. Taylor, “Software architecture: foundations, theory, and practice,” in *Proceedings of the International Conference on Software Engineering-Volume 2.* ACM/IEEE, 2010, pp. 471–472.

[150] R. A. Martin, “Common weakness enumeration,” *Mitre Corporation*, 2007.

[151] “Category:attack - owasp,” https://www.owasp.org/index.php/Category:Attack, 2018, (Accessed on 20/07/2018).

[152] “Category:vulnerability - owasp,” https://www.owasp.org/index.php/Category:Vulnerability, 2018, (Accessed on 20/07/2018).

[153] O. Sheyner and J. Wing, “Tools for generating and analyzing attack graphs,” in *International Symposium on Formal Methods for Components and Objects.* Springer, 2003, pp. 344–371.

[154] J. Siegmund, N. Siegmund, and S. Apel, “Views on internal and external validity in empirical software engineering,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, vol. 1. IEEE, 2015,

pp. 9–19.

[155] V. Mohan and L. B. Othmane, "Secdevops: Is it a marketing buzzword?-mapping research on security in devops," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES).* IEEE, 2016, pp. 542–547.

[156] A. A. Ur Rahman and L. Williams, "Security practices in devops," in *Proceedings of the Symposium and Bootcamp on the Science of Security.* ACM, 2016, pp. 109–111.

[157] M. Abi-Antoun and J. M. Barnes, "Analyzing security architectures," in *Proceedings of the International Conference on Automated Software Engineering (ASE).* ACM, 2010, pp. 3–12.

[158] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW).* IEEE, 2017, pp. 46–53.

[159] A. Sharma and R. Bawa, "A comprehensive approach for agile development method selection and security enhancement," *International Journal of Innovations in Engineering and Technology*, vol. 6, pp. 36–44, 2016.

[160] M. Mohsin, M. U. Sardar, O. Hasan, and Z. Anwar, "Iotriskanalyzer: A probabilistic model checking based framework for formal risk analytics of the internet of things," *IEEE Access*, 2017.

[161] I. Agadakos, C.-Y. Chen, M. Campanelli, P. Anantharaman, M. Hasan, B. Copos, T. Lepoint, M. Locasto, G. F. Ciocarlie, and U. Lindqvist, "Jumping the air gap: Modeling cyber-physical attack paths in the internet-of-things," in *Proceedings of the Workshop on Cyber-Physical Systems Security and Privacy.* ACM, 2017, pp. 37–48.

[162] D. Geneiatakis, I. Kounelis, R. Neisse, I. Nai-Fovino, G. Steri, and G. Baldini, "Security and privacy issues for an iot based smart home," in *Proceedings of the International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* IEEE, 2017, pp. 1292–1297.

[163] O. Mavropoulos, H. Mouratidis, A. Fish, and E. Panaousis, "Asto: A tool for security analysis of iot systems," in *Proceedings of the International Conference on Software Engineering Research, Management and Applications (SERA).* IEEE, 2017, pp. 395–400.

[164] G. Macher, H. Sporer, R. Berlach, E. Armengaud, and C. Kreiner, "Sahara: a security-aware hazard and risk analysis method," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition.* EDA Consortium, 2015, pp. 621–624.

[165] S. V. E. S. S. Committee *et al.*, "Sae j3061-cybersecurity guidebook for cyber-physical automotive systems," *SAE-Society of Automotive Engineers*, 2016.

[166] ISO, "Road vehicles – Functional safety," 2011.

[167] G. Martins, S. Bhatia, X. Koutsoukos, K. Stouffer, C. Tang, and R. Candell, "Towards a systematic threat modeling approach for cyber-physical systems," in *Proceedings of the Resilience Week (RWS).* IEEE, 2015, pp. 1–6.

[168] I. . T. Committee *et al.*, "Analysis techniques for system reliability-procedure for failure mode and effects analysis (fmea)," *IEC 60812*,

2006.

[169] IEC, “Fault tree analysis (FTA) ,” 2006.

[170] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[171] D. Mellado, C. Blanco, L. E. Sánchez, and E. Fernández-Medina, “A systematic review of security requirements engineering,” *Computer Standards & Interfaces*, vol. 32, no. 4, pp. 153–165, 2010.

[172] D. Mellado, E. Fernández-Medina, and M. Piattini, “A common criteria based security requirements engineering process for the development of secure information systems,” *Computer Standards & Interfaces*, vol. 29, no. 2, pp. 244–253, 2007.

[173] P. Salini and S. Kanmani, “Survey and analysis on security requirements engineering,” *Computers & Electrical Engineering*, vol. 38, no. 6, pp. 1785–1797, 2012.

[174] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt, “A comparison of security requirements engineering methods,” *Requirements engineering*, vol. 15, no. 1, pp. 7–40, 2010.

[175] D. Muñante, V. Chiprianov, L. Gallon, and P. Aniorté, “A review of security requirements engineering methods with respect to risk analysis and model-driven engineering,” in *Proceedings of the International Conference on Availability, Reliability, and Security (ARES).* Springer, 2014, pp. 79–93.

[176] O. Daramola, Y. Pan, P. Karpati, and G. Sindre, “A comparative review of i*-based and use case-based security modelling initiatives,” in *Proceedings of the International Conference on Research Challenges in Information Science (RCIS).* IEEE, 2012, pp. 1–12.

[177] S. Kriaa, L. Pietre-Cambacedes, M. Bouissou, and Y. Halgand, “A survey of approaches combining safety and security for industrial control systems,” *Reliability Engineering & System Safety*, vol. 139, pp. 156–178, 2015.

[178] R. Latif, H. Abbas, S. Assar, and Q. Ali, “Cloud computing risk assessment: a systematic literature review,” in *Future Information Technology.* Springer, 2014, pp. 285–295.

[179] Y. Cherdantseva, P. Burnap, A. Blyth, P. Eden, K. Jones, H. Soulsby, and K. Stoddart, “A review of cyber security risk assessment methods for scada systems,” *Computers & Security*, vol. 56, pp. 1–27, 2016.

[180] É. Dubois, P. Heymans, N. Mayer, and R. Matulevičius, “A systematic approach to define the domain of information system security risk management,” in *Intentional Perspectives on Information Systems Engineering.* Springer, 2010, pp. 289–306.

[181] M. Riaz, M. Sulayman, and H. Naqvi, “Architectural decay during continuous software evolution and impact of ‘design for change’ on software architecture,” in *Proceedings of the International Conference on Advanced Software Engineering and Its Applications.* Springer, 2009, pp. 119–126.

[182] K. Tuma, R. Scandariato, M. Widman, and C. Sandberg, “Towards security threats that matter,” in *Proceedings of the Computer Security: International Workshop on the Security of Industrial Control Systems & of Cyber-Physical Systems (CyberICPS).* Springer, 2017, pp. 47–62.

[183] G. McGraw, S. Migues, and J. West, “Building security in maturity

model (BSIMM),” https://www.bsimm.com, accessed: 2017-08-25.

[184] “Empirical study: Threat modeling,” https://sites.google.com/site/empiricalstudythreatanalysis/, accessed: 2017-08-25.

[185] C. Wohlin, M. Höst, and K. Henningsson, “Empirical research methods in software engineering,” in *Empirical Methods and Studies in Software Engineering.* Springer, 2003, pp. 7–23.

[186] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, “Issues in using students in empirical studies in software engineering education,” in *Proceedings of the International Software Metrics Symposium.* IEEE, 2003, pp. 239–249.

[187] P. Runeson, “Using students as experiment subjects–an analysis on graduate and freshmen student data,” in *Proceedings of the International Conference on Empirical Assessment in Software Engineering*, 2003, pp. 95–102.

[188] M. Höst, B. Regnell, and C. Wohlin, “Using students as subjects—a comparative study of students and professionals in lead-time impact assessment,” *Empirical Software Engineering*, vol. 5, no. 3, pp. 201–214, 2000.

[189] I. Salman, A. T. Misirli, and N. Juristo, “Are students representatives of professionals in software engineering experiments?” in *Proceedings of the International Conference on Software Engineering-Volume 1.* IEEE Press, 2015, pp. 666–676.

[190] M. Howard and S. Lipner, *The security development lifecycle.* Microsoft Press Redmond, 2006, vol. 8.

[191] K. Wuyts, R. Scandariato, and W. Joosen, “Empirical evaluation of a privacy-focused threat modeling methodology,” *Journal of Systems and Software*, vol. 96, pp. 122–138, 2014.

[192] K. Labunets, F. Massacci, F. Paci *et al.*, “An experimental comparison of two risk-based security methods,” in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement.* IEEE, 2013, pp. 163–172.

[193] A. L. Opdahl and G. Sindre, “Experimental comparison of attack trees and misuse cases for security threat identification,” *Information and Software Technology*, vol. 51, no. 5, pp. 916–932, 2009.

[194] P. Karpati, A. L. Opdahl, and G. Sindre, “Experimental comparison of misuse case maps with misuse cases and system architecture diagrams for eliciting security vulnerabilities and mitigations,” in *Proceedings of the International Conference on Availability, Reliability and Security (ARES).* IEEE, 2011, pp. 507–514.

[195] P. Karpati, G. Sindre, and R. Matulevicius, “Comparing misuse case and mal-activity diagrams for modelling social engineering attacks,” *International Journal of Secure Software Engineering (IJSSE)*, vol. 3, no. 2, pp. 54–73, 2012.

[196] M. H. Diallo, J. Romero-Mariona, S. E. Sim, T. A. Alspaugh, and D. J. Richardson, “A comparative evaluation of three approaches to specifying security requirements,” in *Proceedings of the Working Conference on Requirements Engineering: Foundation for Software Quality*, 2006.

[197] G. Stoneburner, C. Hayden, and A. Feringa, “Engineering principles for information technology security (a baseline for achieving security),” BOOZ-ALLEN AND HAMILTON INC MCLEAN VA, Tech. Rep., 2001.

[198] P. H. Meland, I. A. Tøndel, and J. Jensen, "Idea: Reusability of threat models-two approaches with an experimental evaluation," in *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS).* Springer, 2010, pp. 114–122.

[199] R. Scandariato, J. Walden, and W. Joosen, "Static analysis versus penetration testing: A controlled experiment," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2013, pp. 451–460.

[200] "Holisec: Holistiskt angreppssätt att förbättra datasäkerhet," http://www2.vinnova.se/sv/Resultat/Projekt/Effekta/2009-02186/HoliSec-Holistiskt-angreppssatt-att-forbattra-datasakerhet/, accessed: 2017-06-14.

[201] H. Yu and C.-W. Lin, "Security concerns for automotive communication and software architecture," in *Proceedings of the Conference on Computer Communications Workshops (INFOCOM WKSHPS).* IEEE, 2016, pp. 600–603.

[202] A. Van Lamsweerde, *Requirements engineering: From system goals to UML models to software.* Chichester, UK: John Wiley & Sons, 2009, vol. 10.

[203] "E-safety vehicle intrusion protected applications," http://www.evita-project.org/index.html, accessed: 2016-11-25.

[204] "Heavens: Healing vulnerabilities to enhance software security and safety," http://www.vinnova.se/sv/Resultat/Projekt/Effekta/HEAVENS-HEAling-Vulnerabilities-to-ENhance-Software-Security-and-Safety/, accessed: 2016-11-25.

[205] "Connected vehicle reference implementation architecture," http://local.iteris.com/cvria/, accessed: 2017-8-25.

[206] T. Rauter, N. Kajtazovic, and C. Kreiner, "Asset-centric security risk assessment of software components," in *Proceedings of the International Workshop on MILS: Architecture and Assurance for Secure Systems*, 2016.

[207] V. Saini, Q. Duan, and V. Paruchuri, "Threat modeling using attack trees," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 124–131, 2008.

[208] K. Tuma, G. Calikli, and R. Scandariato, "Threat analysis of software systems: A systematic literature review," *Journal of Systems and Software*, vol. 144, pp. 275–294, 2018.

[209] A. Karahasanovic, P. Kleberger, and M. Almgren, "Adapting threat modeling methods for the automotive industry," in *Proceedings of the Embedded Security in Cars Conference (ESCAR)*, 2017, pp. 1–10.

[210] K. Tuma and R. Scandariato, "Two architectural threat analysis techniques compared," in *Proceedings of the European Conference on Software Architecture (ECSA).* Springer, 2018, pp. 347–363.

[211] T. D. Oyetoyan, D. S. Cruzes, and M. G. Jaatun, "An empirical study on the relationship between software security skills, usage and training needs in agile settings," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES).* IEEE, 2016, pp. 548–555.

[212] K. Yskout, T. Heyman, D. Van Landuyt, L. Sion, K. Wuyts, and W. Joosen, "Threat modeling: from infancy to maturity," in *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2020, pp. 9–12.

[213] A.-A. O. Affia, R. Matulevičius, and A. Nolte, "Security risk management in e-commerce systems: A threat-driven approach," *Baltic Journal of Modern Computing*, vol. 8, no. 2, pp. 213–240, 2020.

[214] M. Mollaeefar, A. Siena, and S. Ranise, "Multi-stakeholder cybersecurity risk assessment for data protection," in *Proceedings of the International Conference on Security and Cryptography (SECRYPT).* Springer, 2020, pp. 341–348.

[215] R. Jabangwe and A. Nguyen-Duc, "Siot framework: Towards an approach for early identification of security requirements for internet-of-things applications," *e-Informatica Software Engineering Journal*, vol. 14, no. 1, pp. 77–95, 2020.

[216] R. Stevens, D. Votipka, and E. Redmiles, "The battle for new york: A case study of applied digital threat modeling at the enterprise level," in *Proceedings of the Conference on Security Symposium.* USENIX Association, 2018, pp. 621–637.

[217] K. Wuyts, R. Scandariato, and W. Joosen, "Empirical evaluation of a privacy-focused threat modeling methodology," *Journal of Systems and Software*, vol. 96, 2014.

[218] M. Balliu, D. Schoepe, and A. Sabelfeld, "We are family: Relating information-flow trackers," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS).* Springer, Cham, 2017.

[219] J. A. Goguen and J. Meseguer, "Security policies and security models." in *S&P*, 1982.

[220] K. Tuma, "Flaws in flows," https://github.com/ktkatjat/flaws-in-flows.g it, 2018.

[221] B. Berger, K. Sohr, and R. Koschke, "Automatically extracting threats from extended data flow diagrams," in *Proceedings of the Engineering Secure Software and Systems (ESSoS)*, 2016.

[222] A. van den Berghe, K. Yskout, W. Joosen, and R. Scandariato, "A model for provably secure software design," in *Proceedings of the International FME Workshop on Formal Methods in Software Engineering.* IEEE Press, 2017, pp. 3–9.

[223] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *Proceedings of the Symposium on Operating Systems Principles.* ACM, 2009, pp. 321–334.

[224] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif 3.0: Java information flow," Jul. 2006. [Online]. Available: http://www.cs.cornell.edu/jif

[225] K. J. Biba, "Integrity considerations for secure computer systems," MITRE Corp., Tech. Rep., 1977.

[226] S. Jasser, K. Tuma, R. Scandariato, and M. Riebisch, "Back to the drawing board," in *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP).* Springer, 2018.

[227] G. Rasool and D. Streitferdt, "A survey on design pattern recovery techniques," *International Journal of Computer Science Issues*, vol. 8, no. 2, 2011.

[228] M. Guerriero, D. A. Tamburri, and E. Di Nitto, "Defining, enforcing and checking privacy policies in data-intensive applications," in *Proceedings*

of the International Conference on Software Engineering for Adaptive and Self-Managing Systems. ACM, 2018, pp. 172–182.

[229] T. D. Breaux, H. Hibshi, and A. Rao, "Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements," *Requirements Engineering*, vol. 19, no. 3, pp. 281–307, 2014.

[230] T. Abdellatif, L. Sfaxi, R. Robbana, and Y. Lakhnech, "Automating information flow control in component-based distributed systems," in *Proceedings of the International Symposium on Component Based Software Engineering.* ACM, 2011, pp. 73–82.

[231] G. McGraw, "Six tech trends impacting software security," *Computer*, no. 5, pp. 100–102, 2017.

[232] C. Ebert and K. Shankar, "Industry trends 2017," *IEEE Software*, vol. 34, no. 2, pp. 112–116, 2017.

[233] J. A. Wang, H. Wang, M. Guo, L. Zhou, and J. Camargo, "Ranking attacks based on vulnerability analysis," in *Proceedings of the Hawaii International Conference on System Sciences.* IEEE, 2010, pp. 1–10.

[234] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Proceedings of the International Conference on the Quality of Software Architectures.* Springer, 2009, pp. 146–162.

[235] C. Bouhours, H. Leblanc, and C. Percebois, "Bad smells in design and design patterns," *The Journal of Object Technology*, vol. 8, no. 3, pp. 43–63, 2009.

[236] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[237] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, 2019.

[238] T. Nafees, N. Coull, I. Ferguson, and A. Sampson, "Vulnerability anti-patterns: a timeless way to capture poor software practices (vulnerabilities)," in *Proceedings of the Conference on Pattern Languages of Programs.* The Hillside Group, 2017, p. 23.

[239] D. Hosseini and K. Malamas, "Design flaws as security threats," Master's thesis, Chalmers University of Technology and University of Gonthenburg, http://publications.lib.chalmers.se/records/fulltext/250250/250250.pdf, 6 2017.

[240] J. C. da Silva Santos, "Toward establishing a catalog of security architecture weaknesses," Master's thesis, Rochester Institute of Technology, https://scholarworks.rit.edu/theses/9004, 5 2016.

[241] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield *et al.*, "Avoiding the top 10 software security design flaws," *IEEE Computer Society Center for Secure Design (CSD), Tech. Rep*, 2014.

[242] D. Gonzalez, F. Alhenaki, and M. Mirakhorli, "Architectural security weaknesses in industrial control systems (ics) an empirical study based on disclosed software vulnerabilities," in *Proceedings of the International Conference on Software Architecture (ICSA).* IEEE, 2019, pp. 31–40.

[243] K. Tuma, D. Hosseini, K. Malamas, and R. Scandariato, "Inspection

guidelines to identify security design flaws," in *Proceedings of the International Workshop on Designing and Measuring CyberSecurity in Software Architecture (DeMeSSA)*. ACM, 2019, pp. 116–122.

[244] T. DeMarco, *Structured Analysis and System Specification*. Yourdon, 1979.

[245] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Proceedings of the International Conference on Quality Software*. IEEE, 2010, pp. 23–31.

[246] OWASP, "OWASP Top Ten Project," 2017. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP%5FTop%5FTen%5FProject

[247] SANS, "SANS Top 25 Software Errors," 2011. [Online]. Available: https://www.sans.org/top25-software-errors/

[248] K. Tuma, M. Balliu, and R. Scandariato, "Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis," in *Proceedings of the International Conference on Software Architecture (ICSA)*. IEEE, 2019, pp. 191–200.

[249] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen, "Solution-aware Data Flow Diagrams for Security Threat Modelling," in *Proceedings of the Annual ACM Symposium on Applied Computing*. ACM, 2018, pp. 1425–1432.

[250] "Security Design Flaw Detection: Companion Web-Site," Available from Google Sites, 2020. [Online]. Available: https://sites.google.com/view/companion-web-site/

[251] L. Sion, K. Tuma, K. Yskout, R. Scandariato, and W. Joosen, "Towards automated security design flaw detection," in *Proceedings of the International Workshop on Security Awareness from Design to Deployment*, ser. SEAD '19. IEEE, 2019, pp. 49–56.

[252] J. Jürjens and P. Shabalin, "Automated verification of umlsec models for security requirements," in *Proceedings of the International Conference on the Unified Modeling Language*. Springer, 2004, pp. 365–379.

[253] J. C. Santos, K. Tarrit, and M. Mirakhorli, "A catalog of security architecture weaknesses," in *Proceedings of the International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 220–223.

[254] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, "Software design smell detection: a systematic mapping study," *Software Quality Journal*, vol. 27, no. 3, pp. 1069–1148, 2019.

[255] J. Jürjens, *Secure Systems Development with UML*. Springer, 2005.

[256] S. Faily, R. Scandariato, A. Shostack, L. Sion, and D. Ki-Aries, "Contextualisation of Data Flow Diagrams for Security Analysis," *arXiv preprint*, no. arXiv:2006.04098, 2020, presented at GraMSec, collocated with the Computer Security Foundations Symposium (CSF).

[257] S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, and R. Scandariato, "Secure Data-flow Compliance Checks between Models and Code Based on Automated Mappings," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2019, pp. 23–33.

[258] Eclipse Contributors, "Eclipse Documentation – Secure Storage," 2020. [Online]. Available: https://help.eclipse.org/2020-06/topic/org.eclipse.p

latform.doc.user/reference/ref-securestorage-start.htm

[259] T. Wolf, N. Dahyabhai, M. Sohn *et al.*, “EGit – User Guide,” 2019. [Online]. Available: https://wiki.eclipse.org/EGit/User%5FGuide

[260] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert, “UML Superstructure Specification,” Object Management Group (OMG), OMG Standard formal/2017-12-05, 2017, version 2.5.1.

[261] Axway Software, BizAgi Ltd., Bruce Silver Associates, IDS Scheer, International Business Machinesand MEGA International, Model Driven Solutions, Object Management Group, Oracle, SAP AG, Software AG Inc., TIBCO, and Unisys, “Business Process Model And Notation (BPMN),” Object Management Group (OMG), OMG Standard formal/13-12-09, 2014, version 2.0.2.

[262] S. Peldszus, G. Kulcsár, and M. Lochau, “A Solution to the Java Refactoring Case Study using eMoflon,” in *Transformation Tool Contest (TTC)*, 2015, pp. 118–122.

[263] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, “Incremental Co-Evolution of Java Programs based on Bidirectional Graph Transformation,” in *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: virtual machines, languages, and tools (PPPJ).* ACM, 2015, pp. 138–151.

[264] ——, “Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching,” in *Proceedings of the International Conference on Automated Software Engineering (ASE).* IEEE, 2016.

[265] S. Ruland, G. Kulcsár, E. Leblebici, S. Peldszus, and M. Lochau, “Controlling the Attack Surface of Object-Oriented Refactorings,” in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE).* Springer, 2018, pp. 38–55.

[266] J. Knodel and D. Popescu, “A Comparison of Static Architecture Compliance Checking Approaches,” in *Proceedings of the Working Conference on Software Architecture (WICSA).* IEEE, 2007, pp. 12–12.

[267] D. Ganesan, T. Keuler, and Y. Nishimura, “Architecture Compliance Checking at Run-time,” *Information and Software Technology (IST)*, vol. 51, no. 11, pp. 1586–1600, 2009.

[268] J. Bacon, D. Eyers, T. F.-M. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch, “Information Flow Control for Secure Cloud Computing,” *IEEE Transactions on Network and Service Management*, vol. 11, no. 1, pp. 76–89, 2014.

[269] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, “Static Analysis of Android Apps: A Systematic Literature Review,” *Information and Software Technology (IST)*, vol. 88, pp. 67–95, 2017.

[270] Perl::DOC, “Perl Language Reference,” 2020. [Online]. Available: https://perldoc.perl.org/index-language.html

[271] H. Ehrig, G. Rozenberg, and H.-J. Kreowski, *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific, 1999, vol. 3.

[272] D. L. Parnas, “Software Aging,” in *Software Fundamentals.* Addison-

Wesley Longman Publishing Co., Inc., 2001, pp. 551–567.

[273] K. Sultan, A. En-Nouaary, and A. Hamou-Lhadj, "Catalog of Metrics for Assessing Security Risks of Software Throughout the Software Development Life Cycle," in *Proceedings of the International Conference on Information Security and Assurance (ISA).* IEEE, 2008, pp. 461–465.

[274] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-oriented Class Designs," in *Proceedings of the Australian Software Engineering Conference (ASWEC).* IEEE, 2010, pp. 55–64.

[275] S. Peldszus, K. Tuma, D. Strüber, R. Scandariato, and J. Jürjens, "Implementation and Evaluation Data," 2020. [Online]. Available: https://github.com/SvenPeldszus/GRaViTY-SecDFD-Mapping

[276] S. Arzt, M. Benz, M. Miltenberger, A. Ashraf *et al.*, "FlowDroid Release Site," 2020. [Online]. Available: https://github.com/secure-software-engineering/FlowDroid/releases

[277] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android Taint Flow Analysis for App Sets," in *Proceedings of the International Workshop on the State of the Art in Java Program Analysis (SOAP).* ACM, 2014, pp. 1–6.

[278] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java Bytecode for Analyses and Transformations," McGill University, Tech. Rep., 1998.

[279] P. Ferrara, L. Olivieri, and F. Spoto, "Tailoring Taint Analysis to GDPR," in *Proceedings of the Annual Privacy Forum (APF).* Springer, Cham, 2018, pp. 63–76.

[280] S. Arzt, "Static Data Flow Analysis for Android Applications," Ph.D. dissertation, Technische Universität Darmstadt, 2017.

[281] MyBatis, "JPetStore," 2020. [Online]. Available: http://www.mybatis.org/jpetstore-6/

[282] R. C. Bjork, "ATMExample," 2020. [Online]. Available: http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/

[283] R. Jung, R. Heinrich, E. Taspolatoglu, and T. Pöppke, "CoCoME," 2020. [Online]. Available: https://github.com/cocome-community-case-study

[284] R. Heinrich, K. Rostami, and R. Reussner, "The Cocome Platform for Collaborative Empirical Research on Information System Evolution," Karlsruhe Institute of Technology, Tech. Rep. 2016,2, 2016.

[285] A. Meneely, B. Smith, and L. Williams, "iTrust Electronic Health Care System Case Study," 2020. [Online]. Available: https://github.com/ncsu-csc326/iTrust

[286] J. Bürger, D. Strüber, S. Gärtner, T. Ruhroth, J. Jürjens, and K. Schneider, "A Framework for Semi-automated Co-evolution of Security Knowledge and System Models," *Journal of Systems and Software*, vol. 139, pp. 142–160, 2018.

[287] M. Fan, L. Yuy, S. Chenz, H. Zhouy, X. Luoy, S. Li, Y. Liuz, J. Liu, and T. Liu, "An Empirical Evaluation of GDPR Compliance Violations in Android mHealth Apps," *arXiv preprint*, no. arXiv:2008.05864, 2020.

[288] K. Hjerppe, J. Ruohonen, and V. Leppänen, "Annotation-based Static Analysis for Personal Data Protection," in *IFIP International Summer School on Privacy and Identity Management.* Springer, 2019, pp. 343–358.

[289] S. Peldszus, D. Strüber, and J. Jürjens, "Model-Based Security Analy-

sis of Feature-Oriented Software Product Lines," in *Proceedings of the International Conference on Generative Programming (GPCE).* ACM, 2018.

[290] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security Testing: A Survey," in *Advances in Computers.* Elsevier, 2016, vol. 101, pp. 1–51.

[291] L. M. Duarte, J. Kramer, and S. Uchitel, "Using Contexts to Extract Models from Code," *Software and Systems Modeling (SoSyM)*, vol. 16, pp. 523–557, 2017.

[292] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-based Model Conformance and Multiview Consistency Checking," *Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 3, p. 11, 2007.

[293] Z. Diskin, Y. Xiong, and K. Czarnecki, "Specifying Overlaps of Heterogeneous Models for Global Consistency Checking," in *Proceedings of the International Workshop on Model-Driven Interoperability*, 2010, pp. 165–179.

[294] H. König and Z. Diskin, "Efficient Consistency Checking of Interrelated Models," in *Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA).* Springer, Cham, 2017, pp. 161–178.

[295] A. Reder and A. Egyed, "Incremental Consistency Checking for Complex Design Rules and Larger Model Changes," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS).* Springer, 2012, pp. 202–218.

[296] M. Estañol, J. Munoz-Gama, J. Carmona, and E. Teniente, "Conformance checking in uml artifact-centric business process models," *Software and Systems Modeling (SoSyM)*, vol. 18, no. 4, pp. 2531–2555, 2019.