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
University of Gothenburg
Gothenburg, Sweden, 2021

Efficiency and Automation in Threat Analysis of Software Systems

Katja Tuma

ISBN 978-91-8009-154-1 (PRINT)
ISBN 978-91-8009-155-8 (PDF)
ISSN 1652-876X

Technical Report No 191D
Department of Computer Science & Engineering
Division of Software Engineering
University of Gothenburg
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

Abstract **v**

Acknowledgement **vii**

List of Publications **ix**

Personal Contribution **xi**

1 Introduction **1**

1.1 Positioning of Contributions with Respect to the Related Work 3

1.1.1 Threat Analysis of Design Models 3

1.1.2 Automated Security Analysis of Design Models 4

1.1.3 Security Compliance Between Model and Code 6

1.2 Research Focus 8

1.2.1 High manual effort 9

1.2.2 Low recall 10

1.2.3 Disconnect between models and code 11

1.3 Paper Summaries 12

1.3.1 SLR on Threat Analysis (Paper A) 12

1.3.2 STRIDE-per-el vs STRIDE-per-inter (Paper B) 13

1.3.3 Towards Security Threats That Matter (Paper C) . . . 14

1.3.4 STRIDE-per-el vs eSTRIDE (Paper D) 14

1.3.5 Flaws in Flows (Paper E) 15

1.3.6 Detection of Security Design Flaws (Papers F & G) . . 16

1.3.7 Structural Compliance (Paper H) 17

1.3.8 Security Compliance (Paper I) 17

1.4 Discussion 18

1.5 Conclusion and Future Work 26

Bibliography **29**

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-

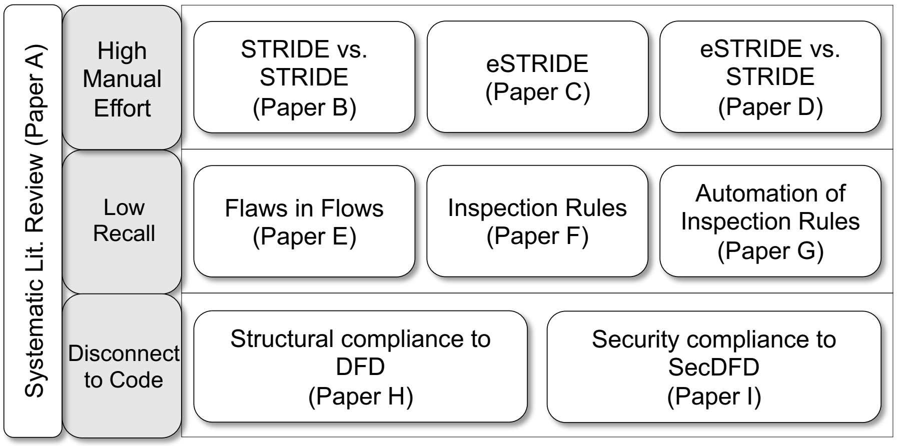
Systematic Lit. Review (Paper A)
High Manual Effort
Low Recall
Disconnect to Code
STRIDE vs. STRIDE (Paper B)
eSTRIDE (Paper C)
eSTRIDE vs. STRIDE (Paper D)
Flaws in Flows (Paper E)
Inspection Rules (Paper F)
Automation of Inspection Rules (Paper G)
Structural compliance to DFD (Paper H)
Security compliance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

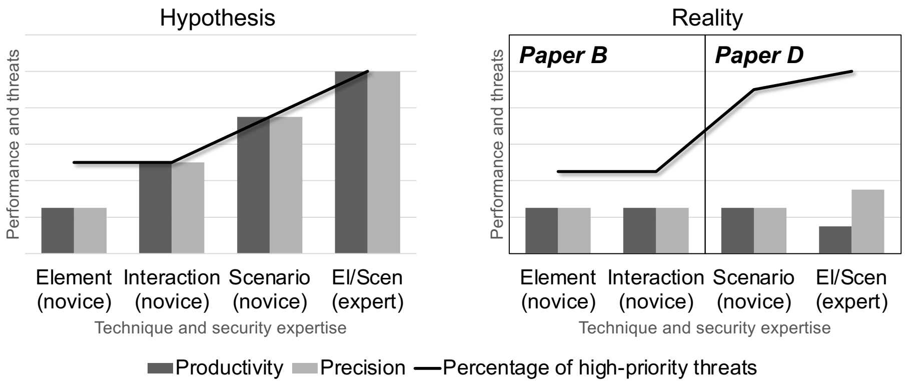

Figure 1.2: Illustration of our hypothesis about technique performance in relation to the cognitive load required for brainstorming threats and security expertise (left) and the reality for three specific techniques (right)

(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

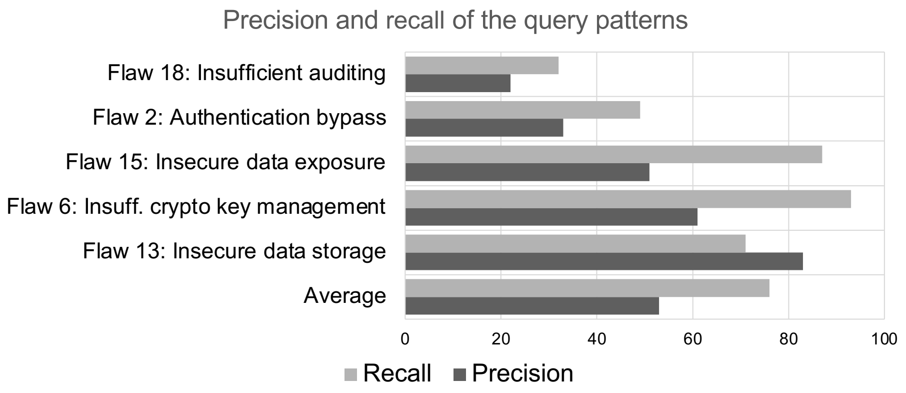

Figure 1.3: Precision and recall of the automated detection of five security design flaws (Paper G)

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,

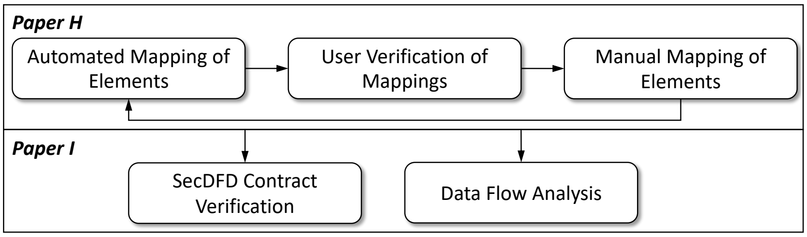
Paper H
Automated Mapping of Elements
User Verification of Mappings
Manual Mapping of Elements
Paper I
SecDFD Contract Verification
Data Flow Analysis

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.

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 implemented 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.