This is a preprint of the paper submitted and accepted at ECSA Tool & Demo Track 2023. For this track, there will be no formal proceedings. An extended version might be published in the Postproceedings by Springer LNCS.

Tool-Supported Architecture-Based Data Flow Analysis for Confidentiality

Felix Schwickerath[1,2], Nicolas Boltz[1,3], Sebastian Hahner[1,3], Maximilian Walter[1,3], Christopher Gerking[1,3], and Robert Heinrich[1,3]

[1] Karlsruhe Institute for Technology (KIT)
[2] felix.schwickerath@student.kit.edu
[3] {boltz,hahner,maximilian.walter,gerking,heinrich}@kit.edu

Abstract. Through the increasing interconnection between various systems, the need for confidential systems is increasing. Confidential systems share data only with authorized entities. However, estimating the confidentiality of a system is complex, and adjusting an already deployed software is costly. Thus, it is helpful to have confidentiality analyses, which can estimate the confidentiality already at design time. Based on an existing data-flow-based confidentiality analysis concept, we reimplemented a data flow analysis as a Java-based tool. The tool uses the software architecture to identify access violations based on the data flow. The evaluation for our tool indicates that we can analyze similar scenarios and scale for certain scenarios better than the existing analysis.

Keywords: Confidentiality · Software Architecture · Security.

1 Introduction

With increased digitalization, more and more systems and digital services are integrated into our lives. These systems often gather data to enable efficient services, like a purchase history in an online shop. This collected data is then exchanged with other services or systems. For instance, in the case of an online shop, customer data might be shared with payment providers. Often, the collected data contains sensitive data, such as the mentioned payment information or a customer's address. Therefore, there is a need to preserve the data's confidentiality.

Confidentiality is described by ISO 27000 as the property "that information is not made available or disclosed to unauthorized individuals, entities, or processes" [11, Section 3.10]. A system violating confidentiality can result in privacy violations, which can result in costly fines, as seen in the case of H&M [25] or British Airways [2]. However, identifying confidentiality violations can be difficult, because the connected services build a complex network of data flows. Hence, a systematic approach to analyze them is required.

Data flow analyses based on source code, e.g., JOANA [24] or KeY [1], cannot consider context information, such as deployment. However, deployment information can be essential for confidentiality, because the deployment can contain whether the application is deployed on an external cloud provider or

not. In addition, source code analyses cannot be used in early design phases because of their need for existing source code. Analyzing the system early at design time is beneficial, because fixing issues in later phases is usually more costly [23]. Seifermann et al. [17, 20] proposed an architecture-based data flow analysis to analyze systems for confidentiality violations. The approach can consider additional context information, such as the deployment, enabling software architects to analyze confidentiality during early design phases. However, their Prolog-based implementation of the analysis is very hard to maintain and has a high resource (memory) demand, which severely limits the applicability for large systems. Hence, we decided to reimplement the analysis as a Java-based open-source Eclipse plugin[4].

The approach of Seifermann et al. [17, 20] consists of a metamodel and an analysis. We explain the metamodel and the scientific concept for the analysis in Section 2. In Section 3, we describe the reasons for the reimplementation and our expected benefits. In addition, we give insight into the tool architecture and how it relates to the scientific concept. Section 4 explains how our developed tool can be used. We compare the old analysis with our newly developed tool in Section 5. For the investigated scenarios, our comparison shows that we can identify the same violations, and we need fewer resources to analyze larger systems. In the last section, we conclude the paper and discuss future work.

2 Modeling Confidentiality in Software-Architectures

Our analysis approach uses software-architectural models to determine the confidentiality of a software system. Here, we build on the Palladio Component Model (PCM) [15] as Architectural Modeling Language (ADL). Using PCM is beneficial, since it supports security analyses [7, 27, 28] as well as performance and reliability analyses [15], thereby reducing the overall effort required by software architects. PCM was a foundation in the original data flow analysis [20].

To enhance the description of our modeling and analysis, we present the running example of an online shop that is deployed within the European Union [6]. Using this online shop, users can browse through the available inventory of items and select an item to purchase. Here, sensitive information, like the user's address, is sent to the online shop, which encrypts this data and stores it in a database that is deployed outside the EU, as shown in Figure 1. Without encryption, this data flow would violate confidentiality.

PCM enables us to describe the software architecture of the online shop from different viewpoints. The structure of the software architecture is modeled as multiple components, e.g., representing the shop interface and the database, and connected in the assembly model. The behavior of the system, e.g., calling of services and processing of data, is modeled as *ServiceEffectSpecifications (SEFF)*. User behavior is captured in the usage model, which contains multiple usage scenarios, each describing the service calls by a user. Lastly, the hardware of the

[4] Video demonstration available: `https://www.youtube.com/watch?v=q3WJsMyqJcA`

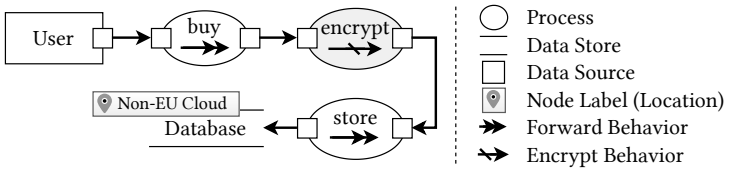

Fig. 1. Data flow diagram that represents the flow of user data in the running example

system is represented in the resource environment, and deployment information is stored in the allocation model.

Seifermann et al. [17, 18, 19, 22] extended PCM to annotate confidentiality-related properties like data sensitivity or encryption and automatically derive a data flow diagram from the architecture. Afterward, the diagram is analyzed in a data flow-based confidentiality analysis [20]. The concept was reused by different architectural analyses targeting uncertainty [5, 8, 9, 29], Industrial IoT [3] or estimating attacker impacts [30].

In the remainder of this section, we briefly describe the concept of the data flow analysis using the running example shown in Figure 1. The automatically derived data flow diagram contains confidentiality-related information that has been extracted from the annotated software architecture model. This includes the behavior of nodes, e.g., encrypting or only forwarding data. Data labels represent the characteristics of the data within the system, e.g., whether the data is encrypted. Node labels represent characteristics of the system itself, e.g., the non-EU deployment location of the database component. All available characteristics are listed in a data dictionary and can be used to define data flow constraints [10]. In our running example, we restrict user data labeled as *personal*, but not labeled as *encrypted*, from flowing to a *non-EU* labeled node.

The data flow analysis checks these constraints using *label propagation* [20]. Data flows through the data flow diagram and can be altered by the nodes' behavior, e.g., by adding the *encrypted* label. In each node, the constraints are examined, taking into account all propagated data labels and also the node's label. In our running example shown in Figure 1, a constraint violation would occur if we remove the *encrypt* node, which is highlighted gray. In this case, the *personal* label would propagate to a *non-EU*-labeled node without the *encrypted* label, which violates confidentiality. This analysis was originally implemented using Prolog. By transforming the data flow diagrams and all of their properties into facts, the Prolog environment can solve queries. Architects can either define their constraints directly in Prolog or by using a domain-specific language [10].

3 Analysis Architecture

The data flow analysis by [20] as described in Section 2 is made up of four steps. Figure 2 shows the analysis steps and their sequential order as an activity diagram. First, the Palladio Component Model (PCM) and analysis-specific models are loaded and references between model elements are resolved. This is done automatically by EMF. Using the information from the models and

annotations, described in Section 2, possible data flows are extracted. As PCM allows developers to model different use cases, the analysis first needs to determine all possible starting points of data flows. For each starting point, the analysis iterates the following calls and adds a node to the data flow for each encountered element. Calls to Service Effect Specifications (SEFFs) defined in interfaces are handled differently: For each call encountered, the analysis adds a calling and returning node in the data flow, as returning values from SEFFs are allowed.

After all data flows are extracted, the analysis propagates the characteristic labels that are defined in the data dictionary model and have been added to the PCM models using annotations. Starting at the first node in the data flow, the analysis evaluates the node characteristics that are present at the given node. Using the node characteristics applicable at the current node and the node characteristics from the previous node, the analysis is able to resolve the defined relationship between inputs/pins and the characteristics of the node. Furthermore, as data characteristics are applied to variables and parameters, the analysis filters the variables with their data characteristics to only include variables that are in scope.

Using the data flows and propagated characteristic labels, data flow constraints can be checked. For example, by comparing propagated data characteristics with defined node characteristics, as described in Section 2.

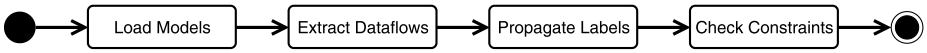

Fig. 2. Analysis architecture as performed activities.

The Prolog-based analysis of [20] realized the extraction of data flows and propagation of labels, by first transforming the PCM models to an explicit DFD metamodel notation, then transforming the DFD elements to Prolog statements and rules. Data flow constraints are checked by defining Prolog queries that are unique to the modeled system and defined data dictionary model. As one DFD element with characteristics is transformed into multiple Prolog statements, the Prolog code grows exponentially with the model size. The exponential growth results in high demand of memory, as the whole Prolog program needs to be fully loaded by the Prolog interpreter.

Additionally, the formulation of constraints and the debugging of issues can become complex due to Prolog. A DSL, as proposed by Hahner et al. [10], can help users to formulate queries to the Prolog model, but the added step of indirection makes it even more difficult to extend the Prolog-based analysis. As the analysis is made up of multiple chained transformations and intermediate model representations, the maintenance of the analysis is made even harder.

Due to the aforementioned reasons, our reimplementation realizes all steps of the analysis using Java. Our reimplementation is based on the current PCM version and does not consider plugins labeled as *incubation*. We extract data flows and represent them in simple ordered lists called `ActionSequence`. An `ActionSequence` is made up of `ActionSequenceElements`, each representing a node in a data flow. We propagate the characteristic labels for each `ActionSequence` individually, by iterating the contained elements and saving the result of the

```
1  var analysis = new DataFlowAnalysisBuilder().build(); // simplified
2  analysis.initializeAnalysis();
3  var allSequences = analysis.findAllSequences();
4  var propagationResult = analysis.evaluateDataFlows(allSequences);
5
6  for (var sequence : propagationResult) {
7    var violations = analysis.queryDataFlow(sequence, node -> {
8      if (node.hasNodeCharacteristic("ServerLocation", "nonEU")) {
9        return node.getAllDataFlowVariables().stream().anyMatch(v ->
10         v.hasDataCharacteristic("DataSensitivity", "Personal") &&
11         !v.hasDataCharacteristic("Encryption", "Encrypted"));
12     }
13     return false;
14   });
15 }
```

Listing 1.1. Code snippet showing how to initialize and how to use the analysis

propagation for each node in the corresponding `ActionSequenceElement`. Doing so not only eliminates the requirement of using logical programming languages, but also removes the need for both transformations and intermediate model representations of the Prolog-based analysis. Thereby, we drastically simplify and reduce maintenance effort. We also create `ActionSequences` with immutable elements, ensuring that no data is shared between `ActionSequences`. This separation of `ActionSequences` allows for the parallelization of the extraction of data flows, propagation of labels, and evaluation of constraints in the future.

Additionally, data and node characteristics are propagated independently of the constraint of the analysis. Due to this reason, our reimplementation is able to evaluate multiple constraints without propagating characteristic labels again. Compared to the Prolog-based analysis, this drastically improves the performance when analyzing a system model for multiple constraints.

4 Tool Application

The Java-based re-implementation of the data flow analysis is available as open source tool based on Eclipse Ecore and the Eclipse Modeling Tools. Documentation and installation guidance of our tool can be found in our repository [14]. We also provide example models that are used as test models to ensure the analysis produces correct results compared to the Prolog-based analysis. Listing 1.1 demonstrates the usage of the analysis using the running example. We provide a *builder* to set up the analysis with required inputs, which is simplified in line 1. After initializing the analysis in line 2, all possible data flows, i.e., sequences, are extracted from the architectural model in line 3. In line 4, we propagate all annotated labels through these data flows. After the label propagation, we search for constraint violations starting in line 6. For each possible data flow in the modeled software architecture, we test each data flow node for a predicate that represents the constraint.

In our running example, we only look for nodes that are located outside the EU in line 8. In lines 9 to 11, we evaluate whether one of these nodes receives *personal* data that has no *encrypted* label. If this is the case, for any node in any possible data flow, the data flow constraint is violated and the violation is collected. After the execution, the variable *violations* contains a list of all constraint-violating nodes within the modeled software system. If no violation has been found, the list remains empty.

5 Evaluation

For our evaluation, we aim to compare our Java-based analysis to the Prolog-based analysis. Our evaluation goals are to examine the accuracy and scalability of both analyses, to show that our reimplementation retains the core functionality of the Prolog-based analysis, while improving execution times and resource demand.

5.1 Evaluation Design

To examine and compare accuracy, we check whether both analyses are able to correctly identify violations, using various PCM models. To ensure a good base for comparison, we reuse the case study-based models that were originally used by Seifermann et al. [20] to evaluate the accuracy of the Prolog-based analysis. We selected the case studies using the default call return semantics of the current stable version of PCM. As a metric, we count correctly identified violations.

To examine and compare scalability, we check the full execution time of both analyses, when analyzing models with increasing size. To better distinguish the impact of different features of the models on the scalability, we generate individual minimal models with an increasing number of either node characteristics, characteristic label propagation, variable actions or SEFF parameters. We chose these elements, as they have the highest impact on either the length of Prolog code or Java loop iterations, depending on the analysis. For each run, we increase the model feature under consideration by the power of ten, starting at 10^0 and ending with 10^5. Each analysis is run with a constraint, which finds a violation at each node, forcing each node to be evaluated once. The constraint ensures a worst-case execution time for both analyses. We run each test 10 times and take the median execution time to exclude outliers or measurement anomalies. We executed the analyses on a dedicated VM. The VM has 4 AMD Opteron 8435 cores together with 97 GB RAM and runs Debian 11 with OpenJDK 11/17.

5.2 Evaluation Results

Regarding accuracy, both analyses were able to correctly identify the 42 violations that were present in the case study-based models and did not return any false positives. Table 1 shows the results of the accuracy evaluation and size of analyzed models. For a better overview, the results have been aggregated based on the underlying case study that has been analyzed. As both analyses performed the

same, we assume, that our reimplemented Java-based analysis is functionally equivalent to the Prolog-based analysis, when analyzing models using the call return semantics.

Case Study	Prolog-based	Java-based	Components	Labels
ContactSMS [12]	10 violations	10 violations	3	4
FlightControl [21]	0 violations	0 violations	6	6
FriendMap [26]	0 violations	0 violations	5	12
Hospital [26]	0 violations	0 violations	4	12
ImageSharing [21]	0 violations	0 violations	1	9
PrivateTaxi [12]	0 violations	0 violations	13	20
TravelPlanner [12]	32 violations	32 violations	7	8
WebRTC [26]	0 violations	0 violations	20	12

Table 1. Results of both analyses compared and size of case study-based models

Regarding scalability, we plotted the results of both analyses as line graphs for each examined model feature, shown in Figure 3. Each graph contains data points from both, the Prolog-based analysis, colored red, and the Java-based analysis, colored blue. Both axes have logarithmic scaling. The x-axis shows the increasing number of model elements and the y-axis the median of execution times in milliseconds. Our evaluation showed that the Prolog-based analysis is not able to complete a run for more than 1000, for node characteristics, or 100, for variable actions and SEFF parameters. As described in Section 3 the Prolog-based analysis has a high demand in system memory. In our tests, the analysis ran into *out of memory errors* or crashed, despite the 97 GB of memory.

When increasing the number of characteristic propagation, the execution time behavior of both analyses is similar. However, for the other evaluated cases, we can observe, that our reimplemented Java-based analysis retains a nearly constant execution time up to 10^3 elements, while the Prolog-based analysis shows an at least linear increase in execution times or fails to complete the analysis run.

The exponential increase in execution time of the Java-based analysis for larger models can be explained by some inefficiencies in sequence finding, overhead during characteristics propagation, and tradeoffs of our immutable approach to action sequences. Nonetheless, we reckon that the time required in all cases is still feasible for design-time analyses. Compared to the Prolog-based analysis, the feasible execution times and ability to even analyze large models make our reimplementation more usable for real-world systems. To overcome the lack of replication packages [13], we provide a data set [16].

6 Conclusion

In this paper, we showcase our Java-based reimplementation of a data flow analysis, based on the approach and tooling of Seifermann et al. [20]. Related approaches and tools are described in the previous publications [20, 19]. We show how to model confidentiality in software architecture and describe the abstract

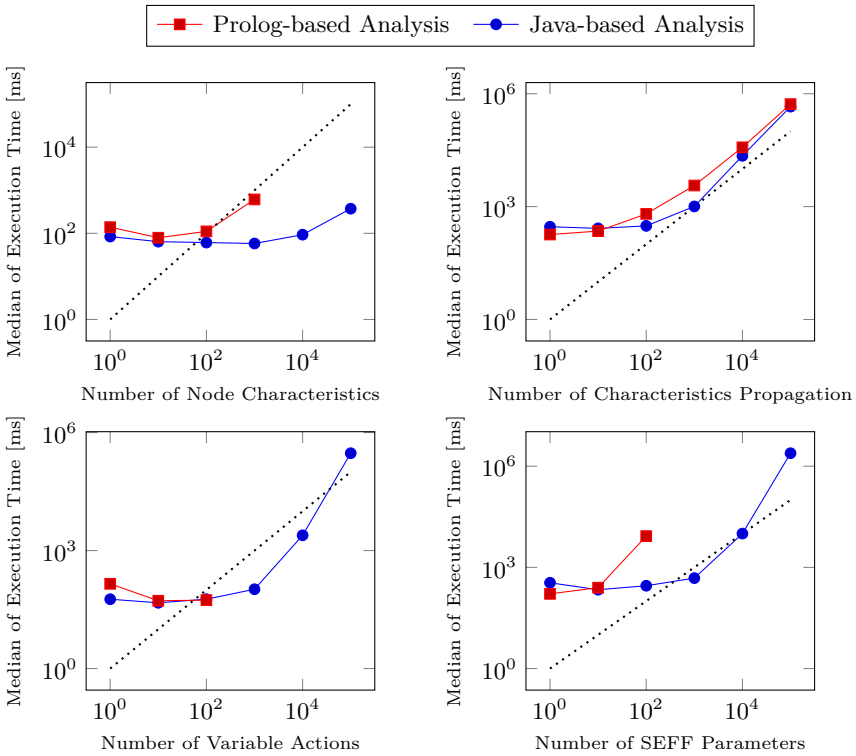

Fig. 3. Scalability Results Prolog-based Analysis and Java-based Analysis

architecture of the analysis. We highlight problems of the Prolog-based analysis of Seifermann et al. [20], including poor maintainability due to complexity and high demand in system memory, and describe the benefits of our Java-based analysis. Further, we show how to apply our new tooling and evaluate our Java-based analysis by comparing it to the existing Prolog-based analysis. In our evaluation, we show that our reimplemented Java-based analysis is functionally equivalent to the Prolog-based analysis and is able to analyze larger system models.

In future work, we aim to apply our tool to constraints regarding privacy as part of a framework for simplified collaboration in legal data protection assessments [4]. We are also currently working to allow explicitly modeled data flow diagram system representations as input.

Acknowledgements This publication is partially based on the research project SofDCar (19S21002), which is funded by the German Federal Ministry for Economic Affairs and Climate Action. This work was also supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs, the DFG (German Research Foundation) project number 432576552 (FluidTrust), the BMBF (German Federal Ministry of Education and Research) grant number 16KISA086 (ANYMOS) and "Kerninformatik am KIT (KiKIT)" funded by the Helmholtz Association (HGF).

References

1. Ahrendt, W., *et al.*: Deductive software verification–the key book. Springer (2016)
2. Beverley-Smith, H., Perowne, C.H., and Kelleher, F.: British Airways Faces Significantly Reduced £20M Fine for GDPR Breach. The National Law Review, (2020). https://www.natlawreview.com/article/british-airways-faces-significantly-reduced-20m-fine-gdpr-breach (visited on 06/07/2023)
3. Boltz, N., Walter, M., and Heinrich, R.: Context-Based Confidentiality Analysis for Industrial IoT. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 589–596. IEEE (2020). DOI: 10.1109/SEAA51224.2020.00096
4. Boltz, N., *et al.*: A Model-Based Framework for Simplified Collaboration of Legal and Software Experts in Data Protection Assessments. INFORMATIK 2022 (2022)
5. Boltz, N., *et al.*: Handling Environmental Uncertainty in Design Time Access Control Analysis. In: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 382–389. IEEE (2022). DOI: 10.1109/SEAA56994.2022.00067
6. Hahner, S.: Architectural Access Control Policy Refinement and Verification under Uncertainty. In: Companion Proceedings of the 15th European Conference on Software Architecture (ECSA-C), virtual (2021)
7. Hahner, S., Heinrich, R., and Reussner, R.: Architecture-based Uncertainty Impact Analysis to ensure Confidentiality. In: 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). IEEE/ACM (2023). Accepted, to appear.
8. Hahner, S., *et al.*: A Classification of Software-Architectural Uncertainty regarding Confidentiality. In: ICETE (2023). accepted, to appear
9. Hahner, S., *et al.*: Model-based Confidentiality Analysis under Uncertainty. In: ICSA-C, pp. 256–263. IEEE (2023). DOI: 10.1109/ICSA-C57050.2023.00062
10. Hahner, S., *et al.*: Modeling Data Flow Constraints for Design-Time Confidentiality Analyses. In: 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C), pp. 15–21 (2021). DOI: 10.1109/ICSA-C52384.2021.00009
11. ISO Central Secretary: Information technology — Security techniques — Information security management systems — Overview and vocabulary. en. Standard ISO/IEC 27000:2018, International Organization for Standardization, Geneva, CH (2018)
12. Katkalov, K.: Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme. doctoralthesis, Universität Augsburg (2017).
13. Konersmann, M., *et al.*: Evaluation Methods and Replicability of Software Architecture Research Objects. In: 2022 IEEE 19th International Conference on Software Architecture (ICSA), pp. 157–168 (2022). DOI: 10.1109/ICSA53651.2022.00023
14. Palladio-Addons-DataFlowConfidentiality-Analysis, https://github.com/PalladioSimulator/Palladio-Addons-DataFlowConfidentiality-Analysis (visited on 07/06/2023)
15. Reussner, R., *et al.*: Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press, Cambridge, MA (2016). ISBN: 9780262034760
16. Schwickerath, F., *et al.*: *Dataset: Tool-Supported Architecture-Based Data Flow Analysis for Confidentiality*. Zenodo, July 2023. DOI: 10.5281/zenodo.8119377. https://doi.org/10.5281/zenodo.8119377.
17. Seifermann, S., *et al.*: A unified model to detect information flow and access control violations in software architectures. In: Proceedings of the 18th International

Conference on Security and Cryptography, SECRYPT 2021, Virtual, Online, 6 July 2021 - 8 July 2021, pp. 26–37. SciTePress (2021). DOI: 10.5220/0010515300260037
18. Seifermann, S.: Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements. PhD thesis, Karlsruher Institut für Technologie (KIT) (2022). DOI: 10.5445/IR/1000148748.
19. Seifermann, S., Heinrich, R., and Reussner, R.: Data-Driven Software Architecture for Analyzing Confidentiality. In: 2019 IEEE International Conference on Software Architecture (ICSA), pp. 1–10 (2019). DOI: 10.1109/ICSA.2019.00009
20. Seifermann, S., *et al.*: Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams. JSS (2021)
21. Seifermann, S., *et al.*: Detecting violations of access control and information flow policies in data flow diagrams. Journal of Systems and Software 184, 111138 (2022). DOI: https://doi.org/10.1016/j.jss.2021.111138
22. Seifermann, S., *et al.*: Identifying Confidentiality Violations in Architectural Design Using Palladio. In: ECSA-C '21 (2021)
23. Shull, F., *et al.*: What we have learned about fighting defects. In: Proceedings Eighth IEEE Symposium on Software Metrics, pp. 249–258 (2002). DOI: 10.1109/METRIC.2002.1011343ISSN: 1530-1435
24. Snelting, G., *et al.*: Checking probabilistic noninterference using JOANA. it - Information Technology 56(6), 280–287 (2014). DOI: 10.1515/itit-2014-1051
25. The Hamburg Commissioner for Data Protection and Freedom of Information: 35.3 Million Euro Fine for Data Protection Violations in H&M's Service Center, (2020). https://datenschutz-hamburg.de/assets/pdf/2020-10-01-press-release-h+m-fine.pdf (visited on 04/04/2023)
26. Tuma, K., Scandariato, R., and Balliu, M.: Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In: 2019 IEEE International Conference on Software Architecture (ICSA), pp. 191–200 (2019). DOI: 10.1109/ICSA.2019.00028
27. Walter, M., and Heinrich Robert Reussner, R.: Architecture-based Attack Path Analysis for Identifying Potential Security Incidents. In: ECSA 2023 (2023). Accepted, to appear
28. Walter, M., Heinrich, R., and Reussner, R.: Architectural Attack Propagation Analysis for Identifying Confidentiality Issues. In: 2022 IEEE 19th International Conference on Software Architecture (ICSA), pp. 1–12. IEEE, Honolulu, HI, USA (2022). DOI: 10.1109/ICSA53651.2022.00009
29. Walter, M., *et al.*: Architectural Optimization for Confidentiality Under Structural Uncertainty. In: Software Architecture : 15th European Conference, ECSA 2021 Tracks and Workshops; Växjö, Sweden, September 13–17, 2021 : Revised Selected Papers. Ed.: P. Scandurra. LNCS, vol. 13365, pp. 309–332. Springer, Heidelberg (2022). DOI: 10.1007/978-3-031-15116-3_1446.23.03; LK 01
30. Walter, M., *et al.*: Architecture-based attack propagation and variation analysis for identifying confidentiality issues in Industry 4.0. at - Automatisierungstechnik 71(6), 443–452 (2023). DOI: doi:10.1515/auto-2022-0135