Data-Driven Software Architecture for Analyzing Confidentiality

Stephan Seifermann, Robert Heinrich, and Ralf Reussner
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{stephan.seifermann, robert.heinrich, ralf.reussner}@kit.edu

***Abstract*—Preservation of confidentiality has become a crucial quality property of software systems that software vendors have to consider in each development phase. Especially, neglecting confidentiality constraints in the software architecture leads to severe issues in later phases that often are hard to correct. In contrast to the implementation phase, there is no support for systematically considering confidentiality in architectural design phases by means of data processing descriptions. To fill this gap, we introduce data flows in an architectural description language to enable simple definition of confidentiality constraints. Afterwards, we transform the software architecture specification to a logic program to find violated confidentiality constraints. In a case study-based evaluation, we apply the analysis to sixteen scenarios to show the accuracy of the approach.**

***Keywords*-data flow, confidentiality, logic programming**

I. Introduction

Quality properties of software systems have significant impact on user acceptance. Besides traditional quality properties, such as performance and reliability, preservation of confidentiality becomes important to users and software vendors. Confidentiality ensures that "information is not made available or disclosed to unauthorized individuals, entities, or processes" [1]. In contrast to other quality properties, confidentiality issues do not only degrade user satisfaction but can have legal consequences. For instance, LinkedIn has been sued for failing to protect peoples' data and agreed on a $1.25 million settlement [2]. The Cambridge Analytica scandal [3] is another example of losing business value [4] by failing to protect data.

Software vendors must consider confidentiality in every development phase to ensure compliance. This is beneficial because the earlier vendors detect issues, the more cost efficient they can fix them. Architectural design is an early phase that helps cutting down costs for fixing issues significantly according to Boehm and Basili [5]. Software architects need means for expressing and analyzing confidentiality to ensure compliance. Because of the complexity of modern systems, detecting confidentiality issues manually is not feasible.

There is a wide range of approaches for analyzing confidentiality but most of them only target the implementation phase. The implementation is, however, not sufficient to introduce confidentiality in a cost-efficient way. There are approaches described in Section IV that cover the early design stage of developing a software architecture but none of them describes system behavior in a data-driven way. However, data-driven behavior descriptions are beneficial because stakeholders express confidentiality requirements in terms of data flow diagrams during stakeholder discussions. Using the same modeling paradigm helps avoiding inconsistencies. In addition, people usually talk about confidentiality in terms of data rather than in terms of processes.

In this paper, we coin the term Data-Driven Software Architecture (DDSA) that describes data and data processing on architectural level. The two contributions of our paper are a model to represent DDSA and an analysis of its compliance with confidentiality requirements. The metamodel for a DDSA introduces data and predefined data processing operators as first class entities. A chain of data processing operations that exchange data describes the system behavior to be analyzed for confidentiality issues. We realized the metamodel as an extension of the established Architectural Description Language (ADL) Palladio Component Model (PCM) [6]. Like Palladio, our approach targets business information systems. The confidentiality analysis detects access right mismatches by comparing access rights assigned to data with roles assigned to processing operations on this data. The analysis considers that data processing can change access rights, e.g. by a processing step declassifying data by an aggregation operation that removes confidential details. We realized the analysis as queries to a Prolog program that a transformation chain derives from the DDSA.

In a case study-based evaluation, we applied our approach to sixteen scenarios. The scenarios belong to four generally applicable equivalence classes for role-based access right analyses. Fourteen scenarios contain confidentiality violations, two scenarios do not. The scenarios stem from two case studies that cover access right violations and have already been used to evaluate the iFlow approach [7].

The remainder of this paper is structured as follows: Section II introduces our running example. In Section III, we explain Palladio and Prolog as foundations. We discuss the state of the art in Section IV. Section V presents an overview of our approach and the envisioned development process. The modeling concept for a DDSA is given in Section VI. Section VII covers the preparation of the confidentiality analysis. The analysis described in Section VIII detects confidentiality issues caused by access right mismatches. Section IX covers the evaluation of the confidentiality analysis. Section X concludes the paper.

978-1-7281-0528-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSA.2019.00009

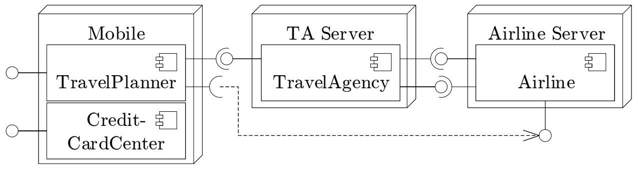

Figure 1. Flight booking system including payments.

II. Running Example

In this section, we introduce a running example to illustrate the modeling and analysis approach described afterwards. The running example stems from the secure information flow analysis iFlow [7]. Figure 1 illustrates a simplified version of a flight booking system in a combined UML composite structure and deployment diagram. We define three roles: *User*, *TravelAgency* and *Airline*. Every resource hosting component instances has one role assigned. The component instances *TravelPlanner* and *CreditCardCenter* have the role *User*. The component instance *TravelAgency* has the role *TravelAgency*. The component instance *Airline* has the role *Airline*.

Figure 2 visualizes the interaction between users and the system, as well as between the system assemblies during the booking of a flight. Users trigger all actions. First, they request flight offers from the travel planner app by giving the request information. The travel planner delegates the call to the travel agency, which in turn delegates it to the airline. The result is a list of flight offers, which the components pass back to the users. The users select a flight and decide to book. They retrieve the credit card information from the credit card center app. Afterwards, they declassify the credit card data via a call to the credit card center, i.e. they release the credit card data for the airline. Eventually, the users call the travel planner to book the given flight offer with the given credit card data. The travel planner forwards the request to the airline. The airline processes the request and pays a commission fee to the travel agency, which sends a confirmation. The airline confirms the booking to the travel planner app. The users receive the confirmation from the travel planner.

The confidentiality constraint to enforce is that the credit card data must only be available to the *User* role. If the credit card data shall be accessible to another role, users have to confirm this explicitly. In the example, this confirmation is given by the call to the *releaseCCDForAirline* operation.

III. Foundations

Our approach relies on two foundations: a) the Palladio Component Model (PCM) serving as our ADL to describe software systems and b) logic programming using Prolog serving as our analysis technology.

The PCM [6] describes software architectures using five partial models. The repository model describes data types, components and the interfaces they provide and require. The component description contains abstractions of the component's behavior, which are called Service Effect Specification (SEFF). When using PCM, a Resource Demanding Service Effect Specification (RD-SEFF) describes basic control structures, resource demands and external calls. We see the resource demands and external calls of RD-SEFFs as abstractions of data processing operations that we have to specify in more detail. A system model describes how instances of the repository components are wired and which services the resulting system provides. A resource environment model describes the available resources including their characteristics. An allocation model describes the deployment of component instances to resources defined in the resource environment model. Usage models describe how users interact with the system. This includes a sequence of calls to system operations as well as the amount of users executing this sequence. We reuse the partial models of PCM and extend them by data processing. The following overview section describes the interaction between PCM and the data processing extension.

Logic programming [8, pp. 67] splits an algorithm into logic and control. Control defines the problem-solving strategy. Logic defines the knowledge to solve the problem. A solver component automatically determines and applies the problem solving strategy. A programmer merely specifies the knowledge. In Prolog, a programmer expresses logic by a set of clauses, which can be facts or rules. A fact starts with a predicate optionally followed by arguments. Rules define conditional facts, i.e. facts that only hold for a given condition. Queries describe the problem to solve. We use the form *predicate/arity* when referring to a *predicate* with *arity* arguments as used by the SWI Prolog manual [9]. We use logic programming to evaluate confidentiality constraints for our architecture description.

IV. State of the Art

There is a wide range of approaches addressing confidentiality. We focus on approaches that use structured system descriptions to carry out automated analyses as described in the survey of Nguyen et al. [10]. We favor automated analyses because they provide strong user guidance and usually require less experience to achieve satisfying results. Therefore, we even do not consider broadly applicable approaches like Threat Modeling [11] that do not provide strong user guidance.

Many design time approaches exploit control flows to reason about confidentiality. Almorsy et al. [12] present an approach for enriching architecture models with security to determine security metrics and analyze scenarios. Security is the umbrella term for various security goals including confidentiality. The approach aims for a generic framework and leaves it up to the users to specify concrete analyses, which leads to weak user guidance. UMLSec [13] defines a UML profile for specifying security properties of systems that can be verified by the CARiSMA [14] tool. However, UMLSec only considers confidentiality on an interface level, which leaves specifying the processing effect of a system service on confidentiality up to the user. Gerking and Schubert [15] describe how security policies defined on an interface level can be preserved during component decomposition but do not provide metamodels or analyses. Heyman et al. [16] describe architectures using

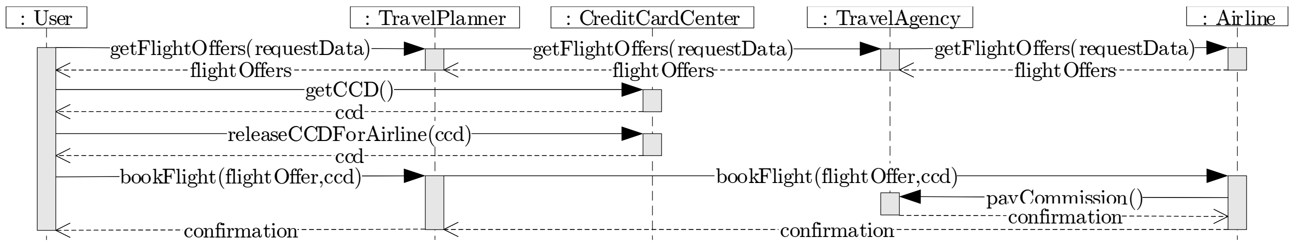

Figure 2. Interaction of system parts during flight booking.

the Alloy language with focus on applying and verifying security patterns. While Alloy is capable of expressing detailed behavior descriptions, the mapping provided by Heyman et al. focuses on security patterns only. In contrast to our work, all control flow oriented approaches do not consider data processing to reason about confidentiality.

There are also design time approaches exploiting data flows to reason about confidentiality. Berger et al. [17] describe an extension of conventional data flow diagrams to allow automated threat analyses. They do not focus on confidentiality in particular but on supporting threat modeling in general. Our approach shares an extended data description and a flexible annotation mechanism with the approach of Berger et al. Object Flows in SOA [18] compare confidentiality measures provided by activity diagram nodes with required confidentiality by object flows. The approach is comparable to a taint analysis. However, it is not capable of expressing access control. iFlow [7] defines a UML profile and the MODELFLOW language to describe systems, their behavior and security requirements. The information flow analysis considers security domains that can be mapped to access control. Modelers either have to explicitly model the transition of information between security domains or use an implementation-like description of the action. Reasoning about such transitions on the granularity of methods can, however, be challenging for users. The same holds true for implementation-like descriptions during design time. In contrast to our work, none of the data flow considering approaches provides means for describing system behavior as a concatenation of simple data processing operations that have an effect on confidentiality properties of the processed data.

In the implementation phase, approaches including information flow analysis [19], code verification [20] or taint analysis [21] are available. These approaches, however, require source code to reason about confidentiality. Therefore, they cannot avoid costly design errors.

During runtime, enforcement of policies such as described by MDSE@R [22] becomes important. Although runtime is not in the scope of this work, we can exploit our iObserve approach [23] in order to update design time models by observations during runtime to make use of the analysis proposed in this paper based on architectural runtime models.

V. Overview of Data-Driven Architecture

We define our data-driven software development process as an extension to the Component-based Software Engineering (CBSE) process. The CBSE process is well-suited for developing software systems based on reusable components. Analysis approaches such as Palladio [6] can analyze the architecture of those systems. However, the process is not sufficient for building data-driven architectures and analyzing their compliance with confidentiality constraints because it misses information about data, data processing and confidentiality properties. Our development process illustrated in Figure 3 bridges this gap. In this and all following figures, grey elements illustrate elements introduced as part of our contributions while white elements are already existing elements. This section provides a rough overview of our approach. For comprehensive descriptions, please refer to the following sections.

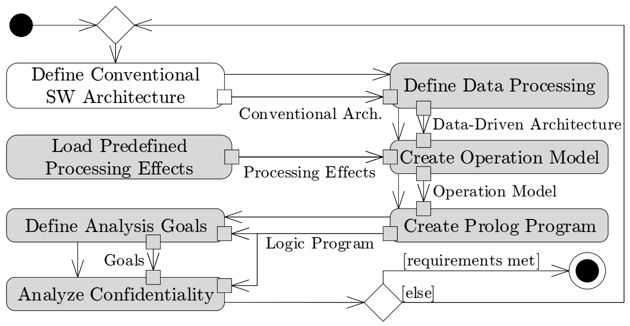

Figure 3. Development process of data-driven architectures.

The starting point of our approach is always a conventional software architecture model that is made of components and control flow oriented behavior descriptions that do not consider data and its processing explicitly. To analyze compliance with data-driven confidentiality constraints, a software architect has to add data flow oriented behavior descriptions, i.e. connected data processing operations and processed data. The result is a DDSA model. Once this DDSA model has been created, architects can easily modify it to reflect evolutionary changes and analyze their impact on confidentiality.

Two automated steps prepare the confidentiality analysis: First, a model transformation transforms the DDSA model and a set of processing effects into a simplified analysis model called *operation model*. By processing effects, we understand the effect a data processing operation has on the processed data. These processing effects are a property of the specific data processing operations. For instance, the processing effect of an aggregation operation might be defined as granting additional access rights to yielded data because it removes confidential details. The processing effect specification can be seen as a function that transforms characteristics of input data to characteristics of output data. These processing effects are predefined for a set of operations but are also extensible by software architects via model elements. The operation model only contains operations including their processing effects, as well as chains of operations that describe how a system handles requests of users. We favor the operation model over the DDSA model for automated analyses because it decouples the analysis from the ADL used to model the system, i.e. the modeling language and the analysis can be used independently. In the second preparation step, another model transformation creates a logic program based on the operation model. We assume that formulating analysis goals declaratively is simpler than specifying it in a procedural manner because the latter requires the logic and the control flow while the former only requires the logic [8, p. 67].

Before the analysis, software architects define analysis goals in terms of the logic program or select predefined goals. The goals use characteristics of data that the analysis derives. In this paper, the goal is to match access right characteristics of data and role characteristics of data processing operations.

In the last step, a logic programming environment tests the goals against the logic program to find confidentiality issues. The environment infers characteristics of processed data by applying processing effects specified for each processing operation to the yielded data based on the characteristics of the input data. The results allow the architect to rate compliance with confidentiality constraints and identify problematic system parts. If the architecture does not meet the requirements, architects start a new iteration to fix reported issues.

VI. Confidentiality-aware Modeling of Data Flows

To reason about confidentiality issues implied by data flows, the analysis needs a description of data and its processing. Our first contribution is the DDSA metamodel that provides this information for the analysis. We derived the metamodel from the concepts of conventional data flow graphs as described by Yourdon and Constantine [24, pp. 43–46]: A data flow graph consists of data *sources*, data *sinks*, *processing operations* and *data transmissions*. Data transmissions are annotated edges between sources, sinks or processing operations. Processing operations transform consumed data and emit new data. These graphs describe the required and provided data for every operation. A chain of data transmissions is a data flow. In addition, the confidentiality analysis requires information about the *characteristics* of processed data in order to derive confidentiality properties. The following paragraphs describe how the metamodel covers the elements of a data flow graph.

a) Characteristics: Characteristics describe properties of data or resources by named finite sets of values. We use characteristics to express access rights and assigned roles. Automated analyses require strongly typed finite sets instead of weakly typed strings, which could only serve documentation

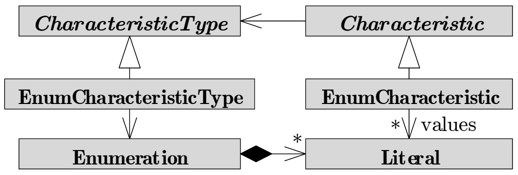

Figure 4. Metamodel excerpt of characteristic specification.

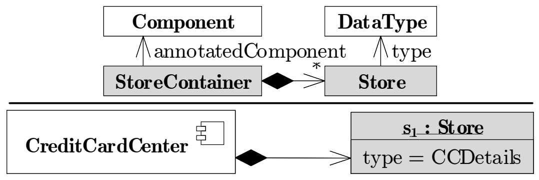

Figure 5. Metamodel excerpt of component stores including example.

purposes. Figure 4 illustrates the partial metamodel consisting of a *CharacteristicType* and a *Characteristic*. The latter describes a concrete characteristic holding values while the former defines the corresponding type. In our running example, we define characteristic types named *AccessRights* and *Roles* that refer to an enumeration. An *Enumeration* holds possible values, i.e. *Literal* elements, that multiple characteristic types can share. In our running example, we create an enumeration named *Roles* holding a literal for every available role.

b) Sources and Sinks: Sources are the starting point of a data flow. Sinks terminate a data flow. Users or stores often play the role of sources or sinks. For instance, a user that submits credentials to a system is the source of a data flow. If the system persists the credentials, the corresponding data store is the sink of the data flow. We define the *Store* concept in the DDSA metamodel as an element that persists data of a certain type. *StoreContainer* elements aggregate stores and attach them to a component. Figure 5 illustrates the partial metamodel of stores in the upper part and gives an excerpt from our running example describing the store of credit card information in the lower part. We do not want a store to be globally accessible because this would violate the requirement that components only communicate through defined interfaces. A global store would introduce a dependency to the context, which limits reusability. Default characteristics are applied to data elements after loading them from the corresponding store. We reuse PCM usage models to represent users.

c) Data: Data elements define typed information to be exchanged between data processing operators. Exchanged data holds characteristics such as access rights. Software architects can either attach characteristics explicitly to data or can let the analysis infer them. In our running example, the analysis infers the characteristics of the request data by just copying it from the previous data element every time the element is transmitted. The following section covers the description of all rules for characteristic inference in detail. The analysis later compares the access rights of the data element with the roles assigned to the data processing operations.

d) Processing Operations: A concatenation of data processing operations describe the data processing in the system. A data processing operation in the DDSA metamodel requires data and provides data like defined in data flow graphs. As explained previously, we describe the transformation of data known from data flow graphs by means of processing effects in order to automatically derive characteristics of provided data in the analysis. The result of applying processing effects are the characteristics that an operation attaches to the provided data elements. The following sections cover all rules for specifying these effects in detail. In our running example, we specify a declassification operation by processing effects that explicitly grant permission: This operation takes credit card data and adds the airline role to the access rights characteristics of the provided data.

We created five categories of data processing operations shown in Table I based on the basic elements in data flow graphs, relational algebra and explicit characteristic definition. We assume this to cover most data processing of information systems. This list is, however, not meant to be complete. Instead, software architects can extend the set of operations in a domain specific way or can specify the effects of data processing on characteristics explicitly with a fall-back operator described below. The operations describe data processing inside components but not between components. The system description of conventional architectures already provides wiring information of component instances. Source operations provide data but do not require data. Sink operations require data but do not provide data. Transmission operations transfer data between components and represent the data transmission concept of data flow graphs. The *PerformDataTransmission* operation can act as sink and source of data, while the *ReturnData* operation only acts as sink. Relational operators transform data using relational algebra. The *SelectData* operation requires a data element and provides a data element but can optionally take n parameters. Characteristic operations explicitly modify data characteristics as modeled by the component developer or software architect. In contrast to the operations mentioned before, such operations allow to model complex data processing. As described later, we assign default processing effects to the operations of the source, sink, transmission and relational categories shown in Table I. The effect often just copies the characteristics from the input data to the output data of the operation.

Figure 6 illustrates the data processing specification of the *getFlightOffers* service of the Airline component from our running example. New data processing operations in the gray box annotate the conventional non data-driven action. This relation between control flow and data flow supports consistency between both behavior descriptions. The first operation loads all flight offers from a given store. The second one selects a matching flight offer from the loaded data by using request data defined in the interface. The last action returns the selected data via return data defined in the interface. The analysis preparation determines the order of operations by resolving data dependencies.

Table I
Data processing operations extension of RD-SEFFs

Category	Operation	In	Out
Source	CreateData	0	1
	LoadData	0	1
	LoadAllData	0	1
Sink	StoreData	1	0
	DeleteData	1	0
	UserReadData	1	0
	SystemDiscardData	1	0
Transmission	PerformDataTransmission	n	m
	ReturnData	1	0
Relational	JoinData	n	1
	UnionData	n	1
	ProjectData	1	1
	SelectData	1(+n)	1
Characteristics	EffectSpecifyingTransformData	1	1

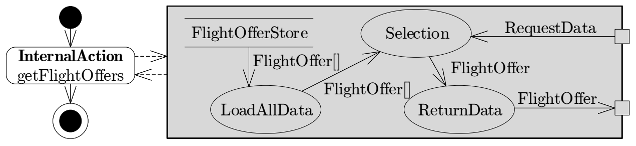

Figure 6. Example of data processing behavior specification.

e) Data Transmissions: Data transmissions exchange data between components. These operations only communicate via interfaces to preserve reusability of components. Signatures of interfaces, however, often miss data specifications. To bridge this gap, we define an *OperationSignatureRefinement* that holds data referring to parameters or signatures. *Result-BasedData* describes result data while *ParameterBasedData* describes parameter data. Figure 7 illustrates that signature refinement. Transmission operations refer to this data.

VII. Preparation of Confidentiality Analysis

The preparation steps described in this section bridge the gap between the DDSA described before and the confidentiality analysis based on a logic program. First, we transform the architecture into a condensed operation model. This operation model decouples the ADL from the analysis, which makes them usable independently. Second, we transform the operation model into a logic program. Both transformations are fully automated model transformations transparent to the software architect.

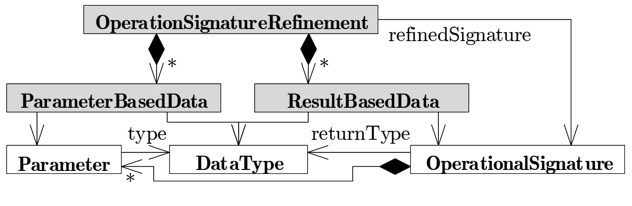

Figure 7. Metamodel excerpt of signature refinements.

A. Transformation to Operation Model

The operation model focuses on representing data processing and its effects. This intermediate model decouples the ADL and the analysis technique. A simplified version of the metamodel is given in Figure 8. It represents meta classes important for the analysis as described in Section VI.

The root node is the *System* class that transitively contains all other elements. *SystemUsage* elements are the starting point of a call sequence. A call sequence consists of operations and calls. An *OperationCall* always targets an *Operation*. An *Operation* can contain operation calls, as well as parameters, return variables, and state variables. The list of operation calls is ordered. If an operation is called, all calls specified in the called operation will happen before the original call returns. An operation holds *Variable* elements that represent parameters, return values and state values. An operation call specifies the characteristics of parameters and state variables via *VariableAssignment* elements. In the same way, an operation specifies the characteristics of return variables via such assignments. Assignments use logic formulas and references to variables of parameters, state variables, and return variables of issued calls to determine the value to assign. A set of boolean variables represents a characteristic. In our running example, we specified roles as the value set for the characteristics *AccessRights* and *Roles*. The operation model represents these characteristics by three boolean variables *User*, *TravelAgency*, and *Airline*. A data element accessible by the user and the travel agency would be represented by setting their variables to true and the airline variable to false. In addition, operations have *PropertyDefinition* elements that consist of characteristics. In our running example, an operation originally hosted in the credit card center component would have a property called *Roles* with the variable *User* set to true.

The mapping from the architecture model to the operation model reduces the complexity of the model to be analyzed while maintaining relevant information for the confidentiality analysis. The described mapping is fully automated by a Xtend-based model transformation that is available in a data set [25]. For the sake of brevity, we do not describe the complete mapping but describe an exemplary execution for Figure 6. In addition, we describe the mapping for *VariableAssignment* elements that specify the effect of data processing. To avoid name clashes, we add the prefix *om::* to elements of the operation model in the description whenever necessary.

Figure 6 illustrates a SEFF of a component. We create one *om::Operation* for the SEFF that holds one state variable for the input data *RequestData* and one return variable for the output data *ReturnData*. In the next step, we create one *om::Operation* for every data processing operation in the SEFF. Every *om::Operation* holds a return variable typed with the data it yields. For instance, the *om::Operation* element *LoadAllData* holds a return variable typed by *FlightOffer[]*. Afterwards, we create one *om::OperationCall* element for every data dependency that an *om::Operation* has. For instance, the *Selection* operation has a

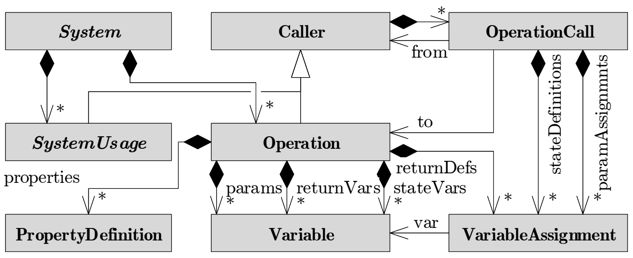

Figure 8. Simplified meta-model of operation model.

dependency to flight offers of the *LoadAllData* operation. To satisfy the dependency, we create an *om::OperationCall* to the *om::Operation* representing the *LoadAllData* operation. No operation satisfies the dependency to request data. Therefore, we do not create an *om::OperationCall*. Now, we create *om::VariableAssignment* elements to connect the data gathered by operation calls to the yielded data for every *om::Operation*: The *om::Operation* representing *LoadAllData* sets the boolean variables of the roles user, travel agency and airline to true because of the default access rights of the *FlightOfferStore*. The *om::Operation* elements representing *Selection* and *ReturnData* create *om::VariableAssignment* elements that refer to the return variables of the called operations, i.e. they specify to copy the boolean variables of the called operation. Finally, we create an *om::OperationCall* from the *om::Operation* representing the SEFF to the *om::Operation* representing *ReturnData*. The corresponding *om::VariableAssignment* elements again just refer to the return variable of the yielded data of the called *om::Operation*.

The general logic for creating *om::VariableAssignment* elements is as follows: For *om::Operation* elements representing *PerformDataTransmissionOperation*, *SEFF* or usage scenario elements, the assignments delegate received characteristics. For all remaining data processing operation types, we apply an extensible set of assignment generators to describe processing effects depending on the operation and the characteristic type.

The default assignment generators use the following logic: *CreateData*, *LoadData* and *LoadAllData* set all characteristic values to false, which equals to having no characteristic activated. Afterwards, initial characteristics based on store characteristics or default characteristics for data creation are applied. Sink data operations have no assignments. *ReturnData* copies the characteristics from its inputs to the return data. *ProjectData* and *SelectData* copy the characteristics from its input to the output without considering parameter data. The *EffectSpecifyingTransformData* operation applies the characteristics that have been specified by the architect. As soon as the effect of a data processing operation depends on the concrete data type being processed, architects can use *EffectSpecifyingTransformData* to cover that effect in the analysis. For all remaining operators, no default assignments apply.

Besides the default assignment generators, domain-specific generators are available. Our confidentiality specific generator applies the following rule in addition to the defaults: *JoinData* and *UnionData* take multiple data inputs and produce one output. The resulting access right characteristic contains the intersection of the access right characteristics of all inputs.

B. Transformation to Logic Program

The operation model contains all information relevant for our confidentiality analysis in a condensed way. We see two benefits of using logic programming to implement the actual analysis: First, we assume that writing confidentiality queries in a declarative way using Prolog to be easier than specifying a full procedure to analyze the model. Providing software architects with predefined goals for common analyses reduces the effort even more. Second, we can use constraint logic programming to find solutions or optimize our architecture.

For the sake of brevity, we omit the description of the transformation of the operation model to a logic program. The transformation is straight forward because no further simplifications or analyses during the transformation are necessary. It is fully automated by a model to text transformation using the Xtend language. We describe parts required to formulate the confidentiality queries in the following section and provide the transformation code in a data set [25].

VIII. Confidentiality Analysis using Logic Programming

The confidentiality analysis is our second contribution. The previous modeling and preparation steps produce a logic program representing data processing and its effect on confidentiality. The analysis queries the logic program to detect confidentiality issues in the represented architecture.

As already defined, confidentiality ensures that data is not accessible to unauthorized subjects. We translate this definition to a constraint that says that the intersection of the access rights of a data object and the roles of a processing operation must not be empty if the operation receives this data object.

To formulate the constraint in Prolog, we need some helper rules: Listing 1 defines a rule to determine all access rights *R* of an operation *OP*. The *findall/3* predicate is built in and collects all solutions for the goal given as the second argument by varying the binding of a variable *X*. The goal uses the fact *operationProperty* generated by the Prolog transformation mentioned before. The existence of a fact *operationProperty(OP, P, V)* means that for an operation *OP* the value *V* of the property *P* is true. In our use case, the property is the roles property and the values are the available roles.

Listing 1
Rule for determining all access rights of an operation.

```
1  accessRights(OP, R) :-
2     findall(X, operationProperty(OP, 'Roles', X),
          R).
```

Listing 2 shows rules to find mismatches between a given set of roles and access rights available for a return value. The first rule in line 1 is true if the list of roles is empty because this implies no access rights will match. The second rule in lines 2–4 takes the first entry *Role* of the set of roles given as first argument and compares the role with the access rights assigned to the return value *RETVAL* in presence of call stack *S*. The clause in line 3 is true if Prolog cannot prove that the return value has the value *Role* for the access right characteristic set to true for a given call stack *S*. The clause in line 4 uses recursion to check the remaining roles in the list tail *R*. The clauses in line 3 and 4 have to be true both in order to yield true for the whole rule. Describing how the predicate *returnValue/4* is defined to determine the characteristics of variables is not possible because of space limitations. However, we provide the full Prolog code in a data set [25].

Listing 2
Rule matching roles with access rights of return value.

```
1  noRoleAuthorizedRetVal([], _, _).
2  noRoleAuthorizedRetVal([Role|R], S, RETVAL) :-
3     \+ returnValue(S, RETVAL, 'AccessRights',
          Role),
4     noRoleAuthorizedRetVal(R, S, RETVAL).
```

We define our query for finding unauthorized access to return values in Listing 3. All clauses in lines 1–6 must be true for a valid solution to our query. In line 1, we define the structure of the call stack *S* to have access to the violating operations and calls in the results. In line 2, we bind the variables of the call stack in a way to ensure that an operation call *CALL* from the operation *OP* to the operation *CALLEE* exists. In line 3, we bind the variable *RT* to the data type of the return value *RETVAL* of the called operation *CALLEE*. In line 4, we ensure that the data type *RT* has an access rights attribute, which equals to having the access rights characteristic. In line 5, we use a helper predicate to collect all access rights *R* of the calling operation *OP*. In line 6, we use the previously defined helper predicate to determine if non of the roles *R* of the calling operation is authorized to access the return value *RETVAL* in presence of the call stack *S*.

Listing 3
Query for finding unauthorized access to return values.

```
1  ?- S=[CALLEE, CALL, OP|_],
2  operationCall(OP, CALLEE, CALL),
3  operationReturnValueType(CALLEE, RETVAL, RT),
4  dataTypeAttribute(RT, 'AccessRights'),
5  accessRights(OP, R),
6  noRoleAuthorizedRetVal(R, S, RETVAL).
```

To find all access violations, we have to consider state variables as well, which means that we added queries analogous to Listing 2 and Listing 3 that test state variables instead of return variables. We do not have to test parameters because the transformation from the DDSA model to the operation model relies on state and return variables only.

IX. Evaluation

This section describes the case study-based evaluation of our approach. First, we give an overview on our goals and metrics. Afterwards, we present our evaluation design and discuss results. Finally, we discuss threats to validity, as well as assumptions and limitations.

A. Evaluation Goal and Metrics

The goal of the study is to evaluate the accuracy of the data flow based confidentiality analysis. Following the guidelines of Basili and Weiss [26], we decompose our goal into a question and corresponding metrics to answer the question. Our question is whether the confidentiality analysis accurately identifies confidentiality issues in a given data processing scenario. We use two metrics suggested by Metz [27] to rate accuracy: The true positive fraction $TPF = \frac{t_p}{P}$ metric M1 calculates the ratio of correctly identified scenarios having confidentiality issues t_p and the scenarios P actually having confidentiality issues. The true negative fraction $TNF = \frac{t_n}{N}$ metric M2 calculates the ratio of scenarios that were identified as not having a confidentiality issue t_n and the scenarios without confidentiality issues N. The benefit of using these two metrics is that we can rate how good the analysis performs regarding scenarios with issues and without issues separately. We do not calculate the false positives fraction $FPF = 1 - TNF$ and false negatives fraction $FNF = 1 - TPF$ because we cannot get additional insights in the accuracy by using them.

B. Evaluation Design

We derive scenarios from existing case studies. Case studies are often used for evaluating confidentiality analyses such as iFlow [7] and UMLSec [14] for instance. The derived scenarios cover all equivalence classes of applicable confidentiality issues that we describe later. Using equivalence classes is necessary because there is an unlimited amount of possible scenarios that can lead to the same type of reported confidentiality issue. We model every scenario using the DDSA metamodel, execute the analysis preparation and execute the queries for the resulting logic program. We classify a result as positive if at least one query results in a solution found by the logic programming environment. We classify a result as negative if no query results in a solution. A scenario can only be classified positive or negative. In the last step, we calculate metric M1 for all equivalence classes that contain confidentiality issues and metric M2 for all equivalence classes that do not contain confidentiality issues. The following paragraphs describe the equivalence classes and the scenarios in more detail.

The confidentiality analysis detects an issue if a data operation having a set of roles R_{op} accesses data with access rights R_{data} such that $R_{op} \cap R_{data} = \{\}$. This analysis specification is not specific for our approach but general applicable. An equivalence class is a set of scenarios that shares the same cause of a confidentiality issue. There are three possible causes of issues: 1) immediate mismatches caused by empty access rights or roles, as well as access right mismatches on data creation, 2) mismatches after data transfer, and 3) access rights changing data processing. This leads to four equivalence classes including the situations without confidentiality issues. These classes are applicable to all types of confidentiality analyses that compare access rights of data and operations.

In order to derive scenarios belonging to the equivalence classes, we picked the case studies *DistanceTracker* and *ContactSMSApp* know from the evaluation of the iFlow approach [28] as a foundation. These case studies are suitable because they are defined on a design level and they basically compare access rights and assigned roles in their analysis part. The distance tracker case study consists of an app that tracks the user, aggregates coordinates and sends this information to a tracking service, which shall not receive raw coordinates but only the distance. We refer to this case study as the *distance tracker* case. The combination of contact and SMS app consists of an app to manage contacts and an app to send SMS messages that shall not receive the name of a contact but only its number. We refer to this case study as the *SMS* case. We modeled both case studies using the DDSA modeling language. To derive a scenario, we inject a confidentiality issue matching the equivalence class into one base model. We designed the scenarios to cover all cases that lead to confidentiality issues. Humans can easily detect such issues in small systems but not in systems with medium or high complexity, for which the approach is meant to be used. The following paragraph describes all scenarios in short. For a detailed description, please refer to our evaluation data set [25]. One scenario always contains exactly one confidentiality issue, which means that we do not have to distinguish original confidentiality issues from follow-up issues in order to identify an issue. We favor introducing issues by removing elements because this requires less changes than adding new elements.

In the *No Issues* class, we use the unchanged data processing of the distance tracker case as scenario *S1* and the unchanged data processing of the SMS case as scenario *S2*.

We defined eight scenarios in the *Immediate* class: *S3* removes the roles assigned to the user in the SMS case. *S4* removes the roles assigned to the contact component in the SMS case. *S5* removes the default access rights of created contact data in the SMS case. *S6* removes the user access rights of created contact data in the SMS case. *S7* extends *S5* by adding the contact role to the access rights during data creation explicitly in the SMS case. *S8* removes the default user access right from the user identifier store in the distance tracker case. *S9* removes the default contact access right from the contacts store in the SMS case. *S10* removes all default access rights from the user identifier store in the distance tracker case. *S11* removes all default access rights from the contacts store in the SMS case.

We defined two scenarios in the *Transfer* class: *S12* removes the default user access right of created confirmation data in the distance tracker case. *S13* removes the default contacts access right of created contact data in the SMS case.

We defined three scenarios in the *Characteristics* class: *S14* removes the tracking service role from the declassification effect in the distance tracker case. *S15* removes the SMS role from the declassification effect in the SMS case. *S16* extends *S5* by returning a union of loaded and one newly created contact data item.

C. Evaluation Results and Discussion

The results of modeling the scenarios and analyzing them for confidentiality issues are given in Table II. *Issues Detected*

Table II
Evaluation results structured by equivalence classes

Class	Issues Detected	No Issues Detected	TPF	TNF
No Issue	–	S1, S2	–	1.0
Immediate	S3, S4, S5, S6, S7, S8, S9, S10, S11	–	1.0	–
Transfer	S12, S13	–	1.0	–
Characteristics	S14, S15, S16	–	1.0	–

means that the analysis detected a confidentiality issue in the mentioned scenarios. *No Issues Detected* means that the analysis did not detect any issues.

In the *No Issue* equivalence class, the correct analysis result is no identified issue. The analysis correctly reported this for both scenarios. We can only calculate the TNF because there are no true positives in this equivalence class. The value of 1.0 indicates the best possible results for scenarios without issues, i.e. the analysis rated both scenarios in this class correctly.

The remaining equivalence classes only contain scenarios with confidentiality issues. Therefore, we cannot calculate TNF. In all equivalence classes, the analysis achieved a TPF of 1.0. This means that the analysis correctly identified all scenarios as having a confidentiality issue.

The analysis shows adequate results for both TPF and TNF. Thus, the quality of the results is good for cases with and without issues. The values are high because we created scenarios that have exactly one confidentiality issue, which simplifies decisions to true or false for every scenario.

D. Threats to Validity

We structure the threats to validity according to the guidelines for case study research of Runeson and Höst [29] into four categories as described in the following.

Internal validity ensures that causal relations are valid, i.e. the factor that is expected to have an influence is the only influencing factor. In our study, we evaluate the accuracy of the confidentiality analysis. We expect the concept and the realization of the analysis to be the influence factors. Additional influencing factors might be the selection of scenarios and the creation of models. We mitigated the model creation bias by using case studies defined by the iFlow approach instead of creating our own case studies. We mitigated the biased selection of scenarios by not deriving them from of equivalence classes tailored to our approach. Instead, we used equivalence classes based on a general definition of role comparing confidentiality analyses as described in the evaluation design. Thus, we expect the selection to be sound.

External validity ensures that the findings can be generalized and the results are valuable for others than the researcher that conducted the study. According to Runeson and Höst [29], case study based research honors deep understanding of a phenomenon more than representativeness. Instead, they see the benefit in providing insights for cases with similar characteristics. We believe that the results of our evaluation hold for cases that allow inferring confidentiality properties on a type level using data processing steps that change characteristics, i.e. cases that do not require inspecting the interplay between two concrete data instances.

Construct validity ensures that the metrics taken are appropriate for answering the research questions. The chosen metrics TPF and TNF are appropriate to rate accuracy in our study design because our analysis yields a binary result that the analysis derives in an objective way. If rating confidentiality was more fuzzy, we had to use other metrics as discussed by Metz [27]. For instance, we did not inject multiple confidentiality issues in one scenario, which would require the analysis to filter follow-up issues. We omitted this in evaluation because this is a known limitation of the analysis for now that we will consider in future work. Another threat is that only one metric is applicable for every equivalence class. However, this does not affect the validity of the results because not calculating an inapplicable metric does not hide inaccuracy of the analysis.

Reliability means that the results shall not depend on the researchers conducting the evaluation. The model creation heavily depends on the conducting researcher because successful modeling requires experience with the modeling language to be used. We avoid this dependency by providing the used models in a data set instead of requiring their creation for conducting the study. In general, we ensured reproducibility by providing the models for all scenarios, the code for the preparation steps, the queries to be used with the logic program, and the classification criteria for the results. Statistical metrics exclude subjective interpretation of the results. Conducting the study requires no creativity but sequential execution of steps that we described in detail. Therefore, we do not expect the results to depend on the conducting researcher but to the easily reproducible.

E. Assumptions and Limitations

The fundamental assumption of our approach is that software architects can describe system behavior using data flows. Even if this is not common yet, we assume this to be possible because data flow diagrams have been used in structured design [24] for years. In our evaluation, we demonstrated that data flows can describe systems in a way that we can identify confidentiality issues. At least in PCM, behavior descriptions are designed to be replaced by other types of descriptions than control flow. Most of the newly introduced metamodel elements are about the behavior of components and users. Therefore, we consider the learning effort for describing an architecture using our modeling approach to be low.

The most restricting limitation of our approach is that it only supports type based confidentiality analyses. This means that we do not consider individual data instances and their concrete values but infer confidentiality properties from classes of data. For instance, the request data exchanged between a user and the airline in our running example is one class of data because it always has the same characteristics even if the concrete values might change. The downside is that we do not support confidentiality that requires specific properties of multi-tenant systems. One example of such an unsupported system is the Bank case study of the iFlow approach [28]. In this system, the iFlow approach ensures that no customer can access the account information of another customer. In contrast, our approach can only ensure that only customers can access account information. For such kinds of systems, control flows have to be taken into account or more complex data processing effect descriptions are required.

Another limitation is that we did not implement measures to handle follow-up issues yet. This means that an issue propagates itself along the data flow until the characteristics change in way that fix the issue or a sink is reached. This is an important part of our future work.

X. Conclusions

In this paper, we proposed our approach for Data-Driven Software Architecture (DDSA) to ensure confidentiality properties in the software design phase. We introduced a metamodel for expressing data and data processing as first class entity in software architectures. The metamodel extends the commonly used ADL PCM. The core idea of the metamodel is to introduce data processing operations that are connected via exchanged data elements. Each operation has an effect on the characteristics of exchanged data. The presented confidentiality analysis uses the access rights characteristic of data and compares it with the roles assigned to an operation. A confidentiality violation appears if the intersection of access rights and roles is empty. The analysis is realized as queries to a logic program. A preparation step generates the logic program based on the ADL artifact. The evaluation examines the accuracy of the confidentiality analysis by applying it to sixteen scenarios. The analysis correctly detected all confidentiality issues while not reporting any false positives.

The benefit of our approach is that software architects can conduct confidentiality analyses in the design phase by means of data processing. We expect this to be more straight forward to specify than control flow oriented descriptions and to be possible in an earlier stage because the required information is already available from the requirements engineering phase. In addition, the model serves as documentation that can be used for communication with other stakeholders.

In future work, we aim for filtering follow-up issues in the reported analysis results. We plan to do this as part of improving the usability of our approach by providing a mapping from the Prolog results back to architectural elements. We want to expand the evaluation of our approach by expressiveness of the metamodel and scalability.

Acknowledgments

The German Federal Ministry of Education and Research funded this work under grant 01IS17106A (Trust 4.0).

References

[1] ISO, "ISO/IEC 27000:2018(E)", Standard, 2018.

[2] J. Fontana, *LinkedIn will pay $1.25 million to settle suit over password breach*, 2015. [Online]. Available: https://zd.net/2rnvggE (visited on 12/05/2018).

[3] J. Isaak and M. J. Hanna, "User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection", *Computer*, vol. 51, no. 8, pp. 56–59, 2018.

[4] H. Weisbaum, *Trust in Facebook has dropped by 66 percent since the Cambridge Analytica scandal*, 2018. [Online]. Available: https : / / www . nbcnews . com / business / consumer / trust - facebook - has - dropped - 51 - percent - cambridge - analytica - scandal - n867011 (visited on 11/21/2018).

[5] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List", *Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[6] R. H. Reussner *et al.*, *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.

[7] K. Katkalov, K. Stenzel, M. Borek, and W. Reif, "Model-Driven Development of Information Flow-Secure Systems with IFlow", in *International Conference on Social Computing*, 2013, pp. 51–56.

[8] W. Ertel and N. Black, *Introduction to artificial intelligence*. Springer, 2011.

[9] SWI Prolog, *Reference Manual*, 2018. [Online]. Available: http://www.swi-prolog.org/pldoc/doc_for?object=manual (visited on 12/08/2018).

[10] P. H. Nguyen, M. Kramer, J. Klein, and Y. L. Traon, "An extensive systematic review on the Model-Driven Development of secure systems", *Information and Software Technology*, vol. 68, pp. 62–81, 2015.

[11] F. Swiderski and W. Snyder, *Threat Modeling*. Microsoft Press, 2004.

[12] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures", in *International Conference on Software Engineering*, 2013, pp. 662–671.

[13] J. Jürjens, *Secure Systems Development with UML*. Springer-Verlag, 2005.

[14] A. S. Ahmadian, D. Strüber, V. Riediger, and J. Jürjens, "Model-Based Privacy Analysis in Industrial Ecosystems", in *Modelling Foundations and Applications*, vol. 10376, 2017, pp. 215–231.

[15] C. Gerking and D. Schubert, "Towards Preserving Information Flow Security on Architectural Composition of Cyber-Physical Systems", in *Software Architecture*, vol. 11048, 2018, pp. 147–155.

[16] T. Heyman, R. Scandariato, and W. Joosen, "Reusable Formal Models for Secure Software Architectures", in *WICSA/ECSA*, 2012, pp. 41–50.

[17] B. J. Berger, K. Sohr, and R. Koschke, "Automatically Extracting Threats from Extended Data Flow Diagrams", in *Engineering Secure Software and Systems*, vol. 9639, 2016, pp. 56–71.

[18] B. Hoisl, S. Sobernig, and M. Strembeck, "Modeling and enforcing secure object flows in process-driven SOAs: An integrated model-driven approach", *SoSyM*, vol. 13, no. 2, pp. 513–548, 2014.

[19] G. Snelting *et al.*, "Checking probabilistic noninterference using JOANA", *it-Information Technology*, vol. 56, no. 6, pp. 280–287, 2014.

[20] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification – The KeY Book*. Springer International Publishing, 2016.

[21] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android", in *International Conference on Dependable Systems and Networks*, 2016, pp. 514–525.

[22] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Mdse@ r: Model-driven security engineering at runtime", in *Cyberspace Safety and Security*, 2012, pp. 279–295.

[23] R. Heinrich, "Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications", *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 4, pp. 13–22, 2016.

[24] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st. Prentice-Hall, Inc., 1979.

[25] S. Seifermann, R. Heinrich, and R. Reussner, *Evaluation Data Set Data-Driven Software Architecture ICSA 2019*, 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2574146.

[26] V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, 1984.

[27] C. E. Metz, "Basic principles of ROC analysis", *Seminars in Nuclear Medicine*, vol. 8, no. 4, pp. 283–298, 1978.

[28] K. Katkalov, "Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme", PhD Thesis, University of Augsburg, 2017.

[29] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.