Detecting violations of access control and information flow policies in data flow diagrams

Stephan Seifermann, Robert Heinrich, Dominik Werle, Ralf Reussner

PII: S0164-1212(21)00235-1
DOI: https://doi.org/10.1016/j.jss.2021.111138
Reference: JSS 111138

To appear in: *The Journal of Systems & Software*

Received date : 9 January 2021
Revised date : 30 September 2021
Accepted date : 18 October 2021

Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams

Authors:

Seifermann, Stephan[a.*]

Heinrich, Robert[a]

Werle, Dominik[a]

Reussner, Ralf[a]

[a] KASTEL - Institute of Information Security and Dependability

Karlsruhe Institute of Technology

Am Fasanengarten 5

76131 Karlsruhe

Germany

Email: stephan.seifermann@kit.edu (Seifermann, Stephan), robert.heinrich@kit.edu (Heinrich, Robert), dominik.werle@kit.edu (Werle, Dominik), ralf.reussner@kit.edu (Reussner, Ralf)

* Corresponding author, phone: +49 721 608 45766, fax: +49 721 608 45990

Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams

Stephan Seifermann*, Robert Heinrich, Dominik Werle, Ralf Reussner

KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany

Abstract

The security of software-intensive systems is frequently attacked. High fines or loss in reputation are potential consequences of not maintaining confidentiality, which is an important security objective. Detecting confidentiality issues in early software designs enables cost-efficient fixes. A Data Flow Diagram (DFD) is a modeling notation, which focuses on essential, functional aspects of such early software designs. Existing confidentiality analyses on DFDs support either information flow control or access control, which are the most common confidentiality mechanisms. Combining both mechanisms can be beneficial but existing DFD analyses do not support this. This lack of expressiveness requires designers to switch modeling languages to consider both mechanisms, which can lead to inconsistencies. In this article, we present an extended DFD syntax that supports modeling both, information flow and access control, in the same language. This improves expressiveness compared to related work and avoids inconsistencies. We define the semantics of extended DFDs by clauses in first-order logic. A logic program made of these clauses enables the automated detection of confidentiality violations by querying it. We evaluate the expressiveness of the syntax in a case study. We attempt to model nine information flow cases and six access control cases. We successfully modeled fourteen out of these fifteen cases, which indicates good expressiveness. We evaluate the reusability of models when switching confidentiality mechanisms by comparing the cases that share the same system design, which are three pairs of cases. We successfully show improved reusability compared to the state of the art. We evaluated the accuracy of confidentiality analyses by executing them for the fourteen cases that we could model. We experienced good accuracy.

Keywords: data flow diagram, access control, information flow

*Corresponding author
Email addresses: `stephan.seifermann@kit.edu` (Stephan Seifermann), `robert.heinrich@kit.edu` (Robert Heinrich), `dominik.werle@kit.edu` (Dominik Werle), `ralf.reussner@kit.edu` (Ralf Reussner)

1. Introduction

In software-intensive systems, software contributes an essential influence on the design, construction, deployment, and evolution of the system as a whole [1]. Consequently, software-intensive systems certainly cover all software systems but also cover, for example, modern production systems, cyber-physical systems or the internet of things. Many attacks target software-intensive systems [2, 3]. Thus, establishing and maintaining security of software-intensive systems is necessary. There are various security objectives that shall be established. Confidentiality, which is one of these security objectives, ensures that "information is not made available or disclosed to unauthorized individuals, entities, or processes" [4]. Confidentiality is hard to achieve in software-intensive systems [5] but it is important to consider in order to avoid high penalties and loss of reputation. Strong data protection regulations such as the General Data Protection Regulation (GDPR) [6] of the European Union carry high financial penalties for failing to protect the data of users. For instance, British Airways is facing a penalty [7] of £20m and Marriott International is facing a £18.4m penalty [8] because of confidentiality breaches. Another threat to companies is loss of reputation after information disclosure. For instance, Facebook users lost trust [9], which also affected the market value, after the Cambridge Analytica scandal [10].

Considering confidentiality is not a small polishing step in the development process but has to be done right from the beginning on. Big software vendors like Microsoft already consider confidentiality in all development phases [11]. Considering confidentiality in the software design is especially crucial to avoid a significant increase in the overall development effort: Boehm et al. [12] reported that fixing an issue becomes more expensive, the later it is fixed. Therefore, issues should be fixed as early as possible in the development process. The same holds for security issues in the development process [13, 14, 15]. This is critical because design issues cause about 50 % of all security issues [15]. Ensuring proper software designs does not free developers from considering confidentiality in the remaining phases but builds a solid foundation for further phases by identifying and fixing fundamental issues that can barely be fixed later even when spending considerable effort.

Model-based confidentiality analyses are appropriate for identifying confidentiality violations caused by a confidentiality issue in software design, as Jürjens [16] demonstrated as part of a case study. A confidentiality violation is a detectable violation of a confidentiality requirement such as a system that receives data, to which it should not have access. A confidentiality issue is the reason why a confidentiality violation occurs. For instance, a system might acquire wrong data because of a wrong service call. Manual inspections of system designs can detect confidentiality violations but this task is complex and labor-intensive, which impedes fast and early detection of violations. A modeling language that is not capable of representing the important aspects for detecting confidentiality violations makes the detection process even harder. Automated model-based confidentiality analyses operating on appropriate models have the potential to speed up finding violations [17]. Especially, model-based confidentiality analyses operating on DFDs are promising because security problems tend to follow the data flow [18], i.e. to identify the cause of a violation, it is often necessary to follow the path that the data took. We already demonstrated that model-based confidentiality analyses based on software designs given as data flows can yield valuable results in Industry 4.0 settings in previous work [19]. DFDs are part of, among others, the curriculum of requirements engineering certifications ,such as the IREB certification [20], and textbooks on requirements engineering, such as [21, 22], which is why designers are usually familiar with DFDs and do not require a steep learning curve.

Confidentiality analyses must support access control and information flow control because both are important confidentiality mechanisms: Access control is the standard for protecting confidential data [23]. Therefore, it is commonly used in practice. For instance, a system might violate an access control requirement by providing a user with information of a certain type, which should be kept secret from that particular user. Information flow control can detect information leaks by data propagation that allow drawing conclusions without direct data flows [24]. For instance, a system might violate an information flow requirement by providing a user with information that has been derived from other information, which in turn should be kept secret from that particular user. Simple information flow control approaches such as taint analysis [25] are applied in practice but more powerful information flow control approaches such as fine-grained noninterference enforcements are not [26]. Access control and information flow control are valid options to use depending on the system and the development context. Even combinations of simple information flow control and access control are possible at implementation level [27, 28], which can improve the protection of information. If modeling and analysis approaches are not capable of representing information flow and access control, the chances are high that they are not applicable in a significant amount of cases in practice.

This article addresses the automatic detection of confidentiality violations in data-oriented software designs. Related work such as [29, 30, 31] (discussed in detail in Section 4) as well as our previous work [32] already suggested modeling languages and analysis semantics in order to realize automated confidentiality analyses of software designs. Nevertheless, we still see the need for further research because of the following challenges that neither related work nor our previous work addressed comprehensively so far: Ch1) A systematic consideration of all possible paths, which data can take in a system design, is necessary to find violations systematically. Ch2) Modeling and analyzing information flow and access control within separate artifacts introduces consistency issues, so a consistent modeling and analysis approach, which supports both confidentiality mechanisms, is necessary. Ch3) User-defined analyses are necessary to cope with specific analysis needs, which are hard or tedious to define in terms of established confidentiality mechanisms. We describe these challenges in more detail in Section 2. The following two contributions address these challenges:

C1) Extended DFD Syntax. We specify an extended DFD syntax by a metamodel that addresses the previously described challenges via syntactical extensions for representing confidentiality mechanisms. The metamodel introduces the concept of alternative data flows via pins to represent multiple data sources and destinations (Ch1). The metamodel distinguishes between system parts that depend on particular confidentiality mechanisms and system parts that do not. Everything related to specific confidentiality mechanisms is encapsulated in extensions that can be defined by users (Ch3). An extension consists of confidentiality properties and behavior descriptions, i.e. descriptions of how the system changes these properties during its execution. The metamodel can represent information flow and access control (Ch2) by such extensions.

C2) DFD Semantics for Confidentiality Analyses. We introduce analysis semantics based on label propagation that support various types of confidentiality analyses. Confidentiality properties are mapped to labels. Behavior descriptions are mapped to label propagation functions. An analysis is defined by a comparison of labels resulting from the label propagation with expected labels stemming from requirements. The comparison can cover information flow and access control analyses (Ch2) as well as user-defined analyses (Ch3). The semantics explicitly consider all possible data flows as well as their combinations, i.e. all data flow paths (Ch1).

We evaluate the presented modeling and analysis approach in a case study including fifteen cases. A case consists of a system, confidentiality requirements given in terms of a particular confidentiality mechanism as well as the properties and behaviors required to reason about confidentiality. We evaluate three aspects of the approach: the expressiveness in specifying systems and analyses, the reusability when replacing confidentiality mechanisms as well as the accuracy of analyses. We evaluate information flow analyses on nine cases and access control analyses on six cases. All cases used to evaluate information flow analyses and half of the cases used to evaluate access control analyses stem from related work. The results indicate good expressiveness and accuracy as well as improved reusability compared to the state of the art.

The remainder of this article is structured as follows. Section 2 describes the three challenges that we address. We describe the running example to illustrate our approach throughout the article in Section 3. Section 4 covers the discussion of the state of the art in DFD semantics as well as design time confidentiality analyses. An overview on how the approach works is given in Section 5. The core contributions are the syntax and the semantics, which we describe in Section 6 and Section 7, respectively. We show how to detect confidentiality violations using both contributions in Section 8. We briefly report on our tooling in Section 9. Section 10 presents the evaluation of the expressiveness and reusability of the syntax as well as the accuracy of defined analyses. Section 11 concludes the article.

2. Challenges

In this section, we describe the challenges in using the DFD syntax of DeMarco [33] for detecting violations of confidentiality requirements. DFDs as introduced by DeMarco [33] are graphs presenting a functional viewpoint on systems based on data processing. There are only four fundamental elements: *Data flows* are unidirectional edges that connect nodes to describe a data transmission between them. *Source and sink* nodes (also called *actors*) start or terminate a sequence of data flows. *Process* nodes transform incoming data to outgoing data. *File* nodes (also called *stores*) persist and emit data. DeMarco describes the semantics of DFDs in an intuitive but incomplete way, so there is no standard semantics.

The lack of full-fledged semantics and shortcomings of the simple syntax make automated analyses of DFDs challenging. Especially, we see the following three open challenges that have not been addressed sufficiently yet.

Ch1) Exploration of multiple data flow paths. A data flow path is a sequence of nodes, which a data item took to reach a particular node. Multiple paths providing the same type of data to the same node commonly occur in realistic applications. For instance, branches can change call destinations and thereby also the destination of sent data. Multiple calls arriving at a certain location imply multiple sources of data for the callee. Modeling approaches have to provide means for describing these multiple paths to represent realistic system designs. The corresponding analysis approaches have to consider all of these paths in a systematic way to detect possible violations. Often, not all combinations of data flows build a valid data flow path from a logical point of view. Therefore, modeling approaches should provide means to specify valid combinations. A common approach to treat multiple data flows is to require an explicit selection of one particular path before the analysis but this is problematic because it does not scale well: In theory, the cross product of all possible choices at every node in a DFD has to be considered if no specification of valid paths is available.

Ch2) Coverage of multiple confidentiality mechanisms. Usually, DFDs require extensions to capture the information required to conduct confidentiality analyses. Single purpose models and analyses cover phenomenons pretty well and provide accurate analyses. However, the downside of single purpose approaches is the lack of flexiblity, i.e. designers have to choose a particular confidentiality mechanism, e.g. information flow or access control, before they start modeling. Switching to another confidentiality mechanism implies remodeling large parts of the system in the new modeling language even if fundamental parts, such as the system structure, could be reused. Remodeling large parts may imply consistency problems: software designers have to ensure that the shared part of both models actually represents the same design. Creating (automated) mappings between two single purpose models is possible in general but such kind of consistency management is challenging if the languages diverge too much [34]. A feasible approach for addressing this consistency problem when switching confidentiality mechanisms is necessary.

Ch3) User-defined confidentiality analyses. Requirements to keep information confidential can be formulated in various ways. However, when designers are forced to use predefined confidentiality mechanisms, even simple requirements such as that a certain piece of information must not flow to one specific node can become complex: In Role-based Access Control (RBAC), a designer has to specify roles and assign these roles to data and nodes in a way that the simple policy can be checked by comparing roles. In information flow, a designer has to do roughly the same steps but for labels instead of roles. Defining custom analyses can be easier. To do so, designers need means for specifying custom analyses and according modeling concepts. As a side effect, this would also allow to integrate new confidentiality mechanisms. An underlying formalism supporting analyses of various confidentiality mechanisms as well as an appropriate modeling language is needed to provide such means.

3. Running Example

To illustrate the concepts described in this article as well as the limitations of the state of the art, we use the TravelPlanner case study [35] of iFlow as a running example. The case study consists of the four systems shown in Figure 1: The travel planner app queries flights and books them on behalf of the user. The credit card center app manages the credit card information of a user. An airline service provides flight information and allows booking flights. A travel agency service mediates between the travel planner and the airline. The scenario is that users query flights, load their credit card data (CCD), book the flight with the airline and the airline pays a commission for mediating to the travel agency.

With respect to confidentiality, there are three totally ordered security levels: The first level *User,Airline,Agency* contains information accessible to all parties. The travel agency, airline and user have clearance for this level. The travel planner and credit card center apps belong to the user. Both apps and the user always have the same clearance. The second level *User,Airline* dominates, i.e., it is bigger than or at least equal to ($\geqslant$), the first level and contains information regarding the flight booking. The airline and user have clearance for this level. The third level *User* dominates the previous levels and contains information only meant for the user. The user has clearance for this level. The critical part of the system is that credit card information from level three must not be disclosed to entities with lower clearance level. However, the airline needs the credit card information to process the booking. Therefore, a declassification of the credit card data explicitly lowers the security level to the second level. If this declassification is missing, there is a violation of the information flow requirements.

The corresponding DFD is shown in Figure 2. The level, behavior and user annotations are part of our extended DFD syntax. The remaining elements follow the notation of DeMarco [33]. Informally speaking, nodes annotated with level 1 belong to the travel agency, nodes annotated with level 2 belong to the airline and the remaining nodes belong to the user. A process with the user annotation (small actor symbol on the left side) is a step executed by the user instead of the system.

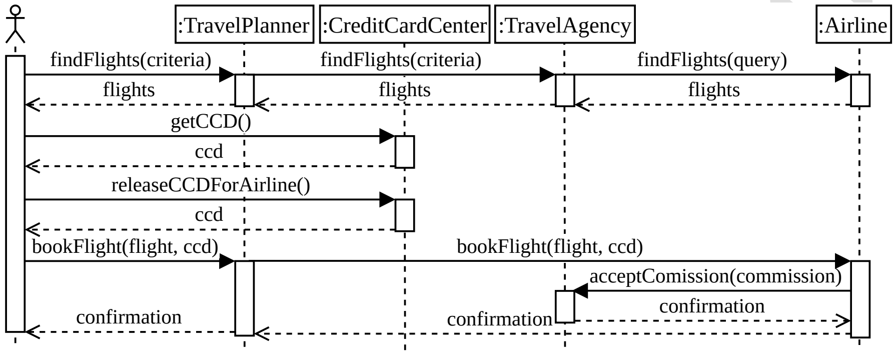


Figure 1: Interactions of components during the booking of a flight in the TravelPlanner running example.

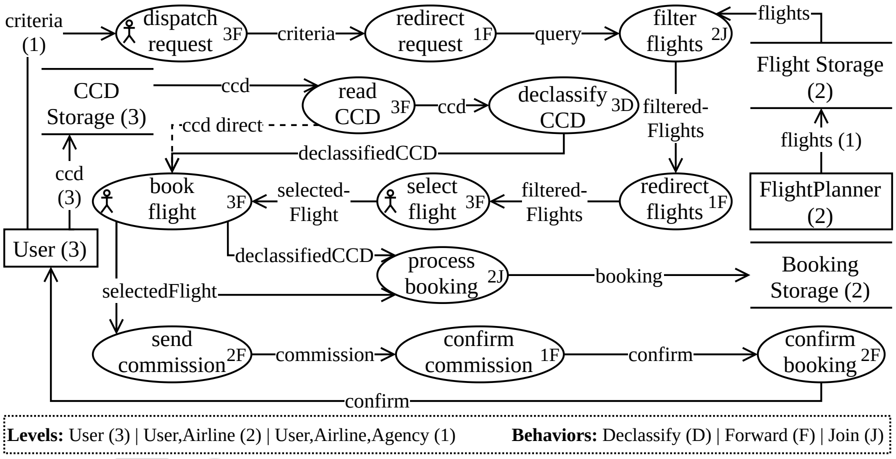


Figure 2: DFD of TravelPlanner example annotated with security levels (1–3) and behavior types (D,F,J). The dashed edge introduces a violation.

4. State of the Art

This article is about detecting violations of confidentiality requirements in software designs by analyzing DFDs. In order to analyze DFDs, we have to define the meaning of every element of a DFD. Various attempts (see Section 4.1), which do not focus on confidentiality, have been made to specify formal semantics of DFDs. Although these semantics are not usable to analyze confidentiality, they reveal shortcomings in the DFD semantics by DeMarco [33] that have to be addressed. Often, such shortcomings stem from ambiguities caused by imprecise or missing information in DFDs. Such ambiguities can be addressed the best by providing additional information in an extended syntax. We derive features that have to be considered by DFD semantics and the corresponding syntax based on the identified shortcomings and ambiguities. We show the significance of these features for modeling and analyzing confidentiality by discussing where these features are used in the running example. In general, the features are an enabler for addressing the challenges (Ch1, Ch2, Ch3) described in Section 2. However, the syntax and semantics cannot address these challenges completely on its own but require support from other parts of the approach such as the analyses.

Approaches for identifying violations of confidentiality requirements do not have to use DFDs but can operate on various artifacts. Because approaches operating on data flows are most closely related to our proposed approach, we separate the discussion of approaches for identifying violations of confidentiality requirements in Section 4.2 by the paradigm of the analyzed artifacts: Section 4.2.1 discusses approaches operating on control flow descriptions and Section 4.2.2 discusses approaches operating on data flow descriptions. We also discuss how the approaches realize the previously mentioned features and whether the approaches sufficiently address the challenges described in Section 2.

4.1. DFD Semantics

The publications about semantics of DFDs, which we describe in the following, frequently report on four shortcomings of the informal semantics introduced for DFDs by DeMarco [33]. They address these shortcomings by extensions. Consequently, we see these extensions as required features for DFD semantics as well as for the syntax if a syntax extension supports a semantical extension. The features are F1) properties of nodes, F2) defined meaning of multiple inputs, F3) behaviors of processes and F4) behaviors of actors. In the following, we explain the significance of these features for detecting confidentiality violations and how well solutions proposed by related semantics address these features.

F1 Node Properties. The properties of nodes are barely covered in related semantics but representing them is important: in our running example, it would not be possible to represent the clearance level of nodes, which is essential for comparing it with the classification level of data to identify violations. France et al. [36] and Petersohn et al. [37] define execution semantics for DFDs and cover node properties as part of the global execution state. This means, properties can change dynamically. While this is an interesting approach, dynamic annotations are more complex to specify compared to static annotations. Therefore, we are interested in exploring whether static annotations are sufficient to represent and analyze common confidentiality mechanisms.

F2 Multiple Inputs. The handling of multiple inputs is a commonly addressed feature. If a process has multiple inputs, they usually relate to each other but it is not clear how. In our running example, it would be unclear that the two credit card inputs are alternatives in *book flight* rather than two mandatory inputs. However, the choice of a particular input can change the analysis results. The simplest solution is to always require all inputs [38, 39, 37, 40] but this is often too restrictive: Requiring all inputs would not allow modeling the alternative input flows *ccd direct* and *declassifiedCCD* in our running example. Expecting all inputs, which roughly equals to expecting all possible incoming calls to be mandatory, is no realistic assumption. Building alternative groups of particular data flows is possible by defining preconditions to select flows [36, 41, 42, 43, 44, 45] or by building sets of data flows. However, an additional alternative flow implies changes in potentially multiple preconditions and sets which can lead to inconsistent specifications in case of many data flows and preconditions or sets. In our running example, adding another input providing credit card data to *book flight* would require adjusting the precondition or the sets. We see potential to further simplify adding an alternative data flow.

F3 Behavior of Nodes. A formal framework to specify the behavior of processes with respect to the effect on data is necessary. In our running example, it is important to specify that *declassify CCD* lowers the classification level of yielded data. This is not possible without means for specifying behavior. However, finding a reasonable level of abstraction for the specification of process behaviors is a challenging topic. Semantics focused on execution [46, 47, 37, 40] do not consider process behavior at all. Semantics using behaviors to specify trigger conditions (conditions for when a process can run) [42, 36, 38, 39] do not describe an effect on yielded data. Both approaches would not allow us to derive data properties from data processing by the system. This means manual and error-prone data classifications are necessary to still support powerful analyses. Specifications of algorithms to calculate outputs [41, 43, 44, 45] can represent wide ranges of effects by specifications given in general purpose languages. However, a generic specification language potentially is more complex to use than a tailored specification language.

F4 Behavior of Actors. The behavior of actors, i.e. the data processing done by actors instead of systems, is often neglected but can be important to consider. In our running example, it is crucial to know that the user does not pass the credit card information received from the simple getter call back into the system but the credit card information received from the declassification operation. Actor behaviors allow to specify that the data is received and passed back into the system. Without such descriptions, we could only guess the origin of data, which could lead to incorrect analysis results. About half of the identified semantics [46, 41, 47, 38] ignore actor behavior but about the other half [42, 36, 39, 44, 45] uses the same means as for specifying process behaviors. Representing actor behavior by the same means as for representing node behaviors is beneficial because this provides a uniform way of specifying behavior. This lowers the learning effort.

4.2. Confidentiality Modeling and Analysis Approaches

To cope with the large amount of confidentiality modeling and analysis approaches focusing on the design and development phases, we discuss categories of approaches and provide examples from these categories. The examples illustrate limitations with respect to the required features identified before as well as limitations in sufficiently addressing the challenges described in Section 2. The limitations apply to the whole category. In the following, we distinguish between approaches analyzing control flows (Section 4.2.1) and approaches analyzing data flows (Section 4.2.2). The latter approaches are closely related to the approach we present in this article.

4.2.1. Modeling and Analysis of Control Flows

Control flow modeling and analysis approaches describe actions to be executed and the order, in which these actions are executed. We distinguish between approaches working on abstractions of the system [48, 35, 49, 50, 51, 52], such as models specified in the Unified Modeling Language (UML), and approaches working with source code [25, 53, 54, 55]. Creating an abstraction of a system usually requires an upfront effort for modeling. However, once the model is created, it can be changed and analyzed for different design alternatives (cf. what-if-analyses) much easier compared to source code. This is because abstracting the system usually reduces dependencies that need to be considered.

Model-based Approaches. One of the most fundamental decision when creating a model-based approach is the level of abstraction of the model to be used. Therefore, we distinguish approaches by the level of detail required to model the behavior of nodes (F3). As illustrated in the overview on related model-based approaches operating on control flows in Table 1, we see three groups of approaches: i) approaches requiring detailed specifications (s) in the top section, ii) approaches using coarse-grained specifications such as predefined behaviors based on node types (nt) in the middle section and iii) approaches not describing the behavior at all (—) in the bottom section. In the following, we do not discuss the features F1, F2 as well as the challenge Ch1 individually because all approaches handle them the same: Properties of nodes (F1) are covered by annotations. The meaning of multiple incoming data flows (F2) is simple: because data flows only happen via calls, every individual call is an alternative data flow consisting of potentially many data items. Consequently, all approaches address the challenge of discovering all data flow paths (Ch1) but restrict themselves to data flows via calls, which cannot represent more complex data flow patterns of DFDs.

The three approaches [48, 35, 49] requiring detailed specifications (i) use the specifications to prove information flow properties of the system model. All three approaches cannot represent data processing by the behavior of actors

Table 1: Overview on model-based confidentiality analysis approaches exploiting control flows. Used abbreviations: a (annotations), c/r (call and return), s (specification), nt (node type), act (activities), IF (information flow), AC (access control), wf (well-formedness).

Approach	F1	F2	F3	F4	Ch1	Ch2	Ch3
Gerking et al. [48]	a	c/r	s	—	c/r	IF	—
iFlow [35]	a	c/r	s	—	c/r	IF	—
UMLSec [49]	a	c/r	s	—	c/r	IF/AC	wf
Hoisl et al. [50]	a	c/r	nt	act	c/r	IF	—
Almorsy et al. [51]	a	c/r	—	—	c/r	AC	wf
Abdellatif et al. [52]	a	c/r	—	—	c/r	IF	—

(F4) but limit behaviors to individual calls to the system. Therefore, they cannot provide full traceability of data that is processed by a user. Besides information flow, UMLSec [49] can analyze access control. However, UMLSec can only control access to actions but not access to data. The support for custom analyses (Ch3) is limited to simple well-formedness constraints. This means that custom analyses can compare annotations and report violations on a structural level. A custom data propagation analysis is not possible without intrusive changes in the UMLSec source code.

There are approaches operating on more abstract behavior descriptions: Hoisl et al. [50] use predefined behavior descriptions for processes (F3) and actors (F4), which they assign based on the type of various nodes. This is often simple to use for designers but also implies restrictions with respect to possible analyses and extensibility: The approach only supports taint analyses, which is a simple information flow mechanism (Ch2), and does not support custom analyses (Ch3).

Approaches not providing behavior specifications [51, 52] for processes (F3) or actors (F4) usually only have limited analysis capabilities. The approach of Almorsy et al. [51] supports a simple form of access control (Ch2) and means to define simple well-formedness analyses (Ch3). The approach of Abdellatif et al. [52] only supports information flow and no custom analyses. Both approaches do not analyze data propagation, so classifying data or other system elements is a manual task and the analyses are limited to pattern matching.

Source Code-based Approaches. There are three types of related approaches operating on source code: taint analyses such as FlowDroid [25], full-fledged information flow analyses such as JOANA [53] or IFcB [54], and verification approaches such as KeY [55]. The approaches either associate properties of nodes (F1) by the node type (e.g. a sensor of a certain type can always be manipulated by an attacker) or by the value of attributes (e.g. a class has an attribute holding its clearance level). The handling of multiple inputs (F2 and Ch1) is the same as for the model-based analyses operating on control flows. The behavior of nodes (F3) is given by the source code and the behavior of actors (F4) is usually not covered. Because approaches based on source

Table 2: Overview on model-based confidentiality analysis approaches exploiting data flows. Used abbreviations: a (annotations), st (structure only), opt (optional flows), es (explicit flow selection), tbl (table), pf (propagation function), s (specification), IF (information flow), AC (access control), m (manual), q (queries), pr (proof requests).

Approach	F1	F2	F3	F4	Ch1	Ch2	Ch3
Threat Modeling	a	st	—	—	—	IF/AC	m
Yampolskiy et al. [58]	a	opt	—	—	—	IF/AC	m
Abi-Antoun et al. [56]	a	st	a	—	—	IF/AC	m
Sion et al. [60]	a	st	a	—	—	IF/AC	m
Alghathbar et al. [61]	a	st	tbl	tbl	—	IF/AC	—
Tuma et al. [29]	a	es	pf	—	—	IF	—
Seifermann et al. [32]	a	es	pf	pf	—	AC	q
van den Berghe et al. [30]	a	es	s	s	—	IF	pr

code are often highly specific to certain application domains or scenarios, they only support one particular confidentiality mechanism (Ch2) and are barely extensible (Ch3). All approaches except for KeY only support information flow analyses. KeY does not prescribe a particular confidentiality mechanism but supports custom analyses (Ch3) via preconditions and postconditions. However, approaches based on source code are not applicable at design time as already motivated.

4.2.2. Modeling and Analysis of Data Flows

Design time approaches exploiting data flows are closely related to our work. Table 2 gives an overview on the approaches discussed in the following. The upper part of the table covers threat modeling approaches. The lower part covers data propagation analyses.

Threat modeling [56, 57, 58, 59, 60] is frequently researched. Because of the flexible nature of threat modeling, multiple confidentiality mechanisms (Ch2) and custom analyses (Ch3) are usually supported. All approaches support node properties (F1) by static annotations and do not consider actor behaviors (F4). All approaches allow multiple inputs but only Yampolskiy et al. [58] distinguish mandatory and optional data flows (F2). However, the selection process of their introduced optional flows is still not specified in [58], so systematically considering multiple flow paths is not possible (Ch1). The behavior of processes (F3) is often not represented: Only Abi-Antoun et al. [56] and Sion et al. [60] describe behaviors by annotations. These annotations are compared to patterns later. All analyses are limited to purely structural analyses that perform pattern matching and that do not derive properties of exchanged data based on its processing. Therefore, reasoning about information flow requires either manually classifying all exchanged data, which can be a complex task, or only yields results with the same granularity as simple taint analyses. Reasoning about multiple classification levels, like we do in the running example, is not possible.

Data propagation analyses reduce the complexity of the labeling task by not requiring all data to be labeled manually. Manual labeling is repetitive and sometimes challenging, so it is error prone. Instead, data propagation analyses require a limited set of initial labels that are propagated through the system. As a consequence, only few labels have to be assigned manually, which reduces the complexity compared to the category of approaches discussed before. FlowUML [61] derives DFDs from UML sequence diagrams, models them in a logic program and describes how to detect violations of information flow requirements as well as Discretionary Access Control (DAC) and Mandatory Access Control (MAC) requirements. Therefore, they support information flow and access control (Ch2). FlowUML uses specific node types to represent properties of nodes (F1), which is comparable to static annotations, and specifies behaviors of processes (F3) and actors (F4) by tables that relate data flows. The handling of multiple flows (F2) and also multiple data flow paths (Ch1) is not described in the FlowUML paper [61], so it is unclear how well realistic systems can be modeled and analyzed by the approach. Formulating custom analyses (Ch3) is not described. We could not find any publications reporting on an evaluation of FlowUML. Therefore, it is unclear whether the approach is applicable to realistic systems and whether it provides accurate results.

Tuma et al. [29] as well as our previous work [32] have been evaluated for realistic systems. Both approaches describe the system behavior (F3) as a sequence of label propagation functions and initial labels on data. Both approaches represent properties of nodes (F1) as static annotations. Tuma et al. only support information flow and do not consider the behavior of actors (F4). Our previous work [32] only supports access control and considers the behavior of actors (F4) by label propagation functions. Considering actor behaviors allows to specify, for instance, which particular credit card information is passed to the system in our running example, which in turn affects the analysis results. Both approaches only support exactly one type of confidentiality analysis (Ch2). Our previous work [32] additionally provides means for specifying custom analyses (Ch3) via queries. Both approaches do not provide means for systematically considering all possible data flow paths (Ch1) in presence of multiple valid selections of inputs but prescribe one particular input selection (F2). Prescribing one selection allows analyses in presence of ambiguities but does not guarantee to find violations produced by other possible selections.

van den Berghe et al. [62] describe systems by data flows between predefined processing operators to prove security properties including a simple form of information flow control but no access control (Ch2). They describe system behavior in the proof assistant Coq by stateful modeling in linear-time temporal logic. These behavior descriptions can be used to describe the behavior of processes (F3) and actors (F4). Properties of nodes (F1) can be defined freely and they can change dynamically. This also enables formulating custom analyses (Ch3). The behavior description of nodes includes the logic for selecting inputs (F2). However, the paper does not report on systematically considering all possible data flow paths (Ch1). Additionally, including the selection logic of inputs in behavior descriptions hinders reusability because the same behavior cannot be used for two nodes with different amounts of inputs. In our running example, we would have to specify a dedicated behavior for the *book flight* process instead of just reusing the *Forward* behavior because the behavior would have to be extended by the selection logic of the alternative incoming flows of credit card information.

5. Overview of the Approach

Before describing the contributions in detail, we give a high-level overview on how our modeling and analysis approach is applied and how it works. The goal of the approach is to detect violations of confidentiality requirements. This, especially, covers requirements given in terms of information flow or access control. To apply the approach, the three activities illustrated in Figure 3 are necessary: creating an analysis definition, modeling the system and running the analysis. The analysis definition introduces confidentiality-related model elements that are used while modeling the system. Often, it is sufficient to create the analysis definition once and use it for various systems. We explain all of these activities in the following.

Creating an Analysis Definition. An analysis definition is a collection of the following model elements: 1) properties of nodes (F1), 2) properties of data, 3) behavior description of nodes (F3 and F4) and 4) a comparison function. In our running example, the properties of nodes (1) are the clearance levels and the properties of data (2) are the classification levels. The behavior descriptions (3) define how nodes process data, i.e. what properties outgoing data will have based on properties of incoming data. In our running example, the behavior descriptions are *Forward*, *Join* and *Declassify*. The *Forward* behavior copies incoming data properties to outgoing data properties unchanged. The *Join* behavior looks for the highest classification level on all incoming data and applies that level to outgoing data. The *Declassify* behavior explicitly sets the classification to the second level. The comparison function (4) defines a pattern that indicates a violation by comparing data and node properties. In our running example, the comparison function looks for a node with a clearance level lower than the classification level of any data received by that node. A dedicated security expert creates the analysis definition because it requires security expertise to map a confidentiality analysis to the described four model elements. Alternatively, a

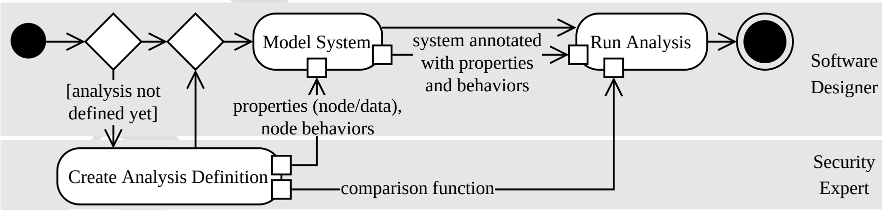


Figure 3: Process for creating and analyzing a system design visualized as UML activity diagram.

software designer can carry out these activities if he/she has security expertise. Analysis definitions (or at least parts of it) are often reusable. Therefore, defining an analysis is only required if it has not been defined before. Consequently, security experts do not have to take part in the design process of every system but only in the processes that require new analysis definitions. Decoupling the analysis-specific model elements, i.e. the analysis definition, from the remaining DFD elements is not only beneficial for assigning clear responsibilities: In previous work [63], we demonstrated that this separation also improves maintainability. In addition, the separation is beneficial for reusing models as we show in the evaluation in Section 10. In our running example, the whole analysis definition can be reused for other systems if the particular levels are renamed. Because the analysis definition is sufficient to represent the core elements of a confidentiality mechanism and creating the analysis definition does not require intrusive extensions of the overall approach via source code, the analysis definition addresses the challenge of defining custom analyses (Ch3). As we will show in the evaluation in Section 10, the analysis definition is expressive enough to represent information flow and access control mechanisms, so it also addresses the challenge about representing both confidentiality mechanisms (Ch2).

Modeling the System. First, a software designer models the structure of the system with the DFD elements, which DeMarco [33] introduced. Next, the designer integrates the confidentiality mechanism into the system by applying elements from the previously defined analysis definition to DFD elements. In our running example, the designer assigns each node a clearance level and a behavior description. Assigning data properties explicitly is not necessary: behavior descriptions can provide initial data properties of newly created data and the analysis will determine the remaining data properties later. In our running example, the behavior description of the *FlightPlanner* specifies that outgoing data always is classified by the first level. In contrast, the *Forwarding* behavior of the *dispatch request* process does not provide an initial data classification but will derive the classification during the analysis.

Running the Analysis. The software designer starts the fully automated analysis. The result is a list of detected violations according to the comparison function. The fundamental idea of the analysis is to map the DFD, the properties and the behavior descriptions to a label propagation network. Properties become labels, nodes become label propagation functions according to their behavior description and data flows define the connections between the label propagation functions. The analysis propagates all labels through the network. After that propagation, the labels of all data at all nodes are known. In the last step, the comparison function compares the labels to identify a violation. To find information flow violations in our running example, we look for an edge with a higher classification label than the clearance level of the receiving node. The dashed edge in Figure 2 causes such a flow: The dashed flow circumvents the declassification process, which makes the credit card data arriving at *book flight* level 3 instead of level 2. The *process booking* process receives this level 3 data but its clearance is only valid up to level 2. This means that we found a violation. The dashed data flow as well as the solid data flow transport credit card data to the *process booking* process. As we will explain in Section 6, we introduced a notion to clearly state that both flows are alternative flows (F2), which means that exactly one of these flows has to be chosen. As we will explain in Section 7, the analysis systematically explores all possible combinations of data flows transporting labels, which addresses the corresponding challenge (Ch1).

In the presented running example, the violation is easy to spot but in more complex systems, finding all possible sources and properties of incoming data is challenging. Using the sketched analysis can help designers to identify issues in software designs and correct them before the implementation of the introduced issue starts. In the running example, a designer has to ensure that data, which has not been declassified, never arrives at the *book flight* process by removing the faulty data flow. Programmers later have to adhere to this specification and ensure that data always goes through a declassification operation.

6. Syntax of Extended Data Flow Diagram

In order to realize the identified missing features of DFD semantics described in Section 4.1, we have to extend the syntax and the semantics of DFDs. It is not sufficient to only extend the semantics because we need additional information to solve ambiguities such as the handling of multiple inputs and outputs. In this section, we introduce the syntactical DFD extensions that support the definition of semantics discussed in Section 7. An overview on the syntax is given by the metamodel in Figure 4. Grey elements are DFD elements as introduced by DeMarco [33]. Non-filled elements are the extending elements introduced in this article. As part of the following descriptions, we relate the syntax to the metamodel used in our previous work [32] as well as to closely related approaches [29, 30].

Node Characteristics (F1). To cover relevant properties of nodes, we introduce typed characteristics. Strong types are beneficial because identifying and matching properties becomes possible. Sets of discrete values can represent relevant properties such as roles or classification levels. We call such a discrete value `Label`. An `Enumeration` builds an ordered set of corresponding labels. Analyses can make use of the order, e.g. to determine dominance between labels. In our running example, the security levels are an enumeration of ordered labels. Labels with a higher index in such a list dominate labels with a lower index. A `CharacteristicType` is the type of a property with a value range given by an enumeration. In our running example, the clearance and classification are characteristic types referring to the enumeration of security levels. A `Characteristic` is an instance of a characteristic type selecting a subset of available labels, which means that these labels apply. Every node can hold multiple characteristics, which means that the selected labels apply to the node. In our running example, we use a number inside the node to visualize a node characteristic. The number indicates a particular label, i.e. clearance level, that has been selected from the characteristic type for clearance levels. In contrast

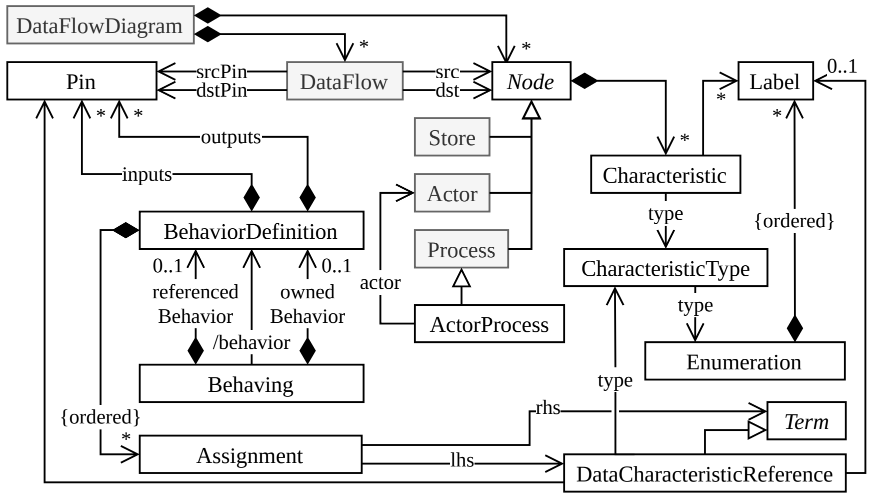


Figure 4: Metamodel of DFD (grey elements) with confidentiality extensions (non-filled elements).

to related work [29, 30] and to our previous work [32], labels can be ordered and that order can be used in analyses, which we describe later.

Pins (F2). We introduce the concept of a `Pin` to clearly specify required data. A pin describes either a required input data or output data. The set of all input and output pins describes the interface of the node. The pins are similar to pins in the UML [64, pp. 444], which also distinguishes inputs and outputs. In contrast to the UML, we use one fixed meaning of how data is transferred through pins to simplify the usage of pins. We will see this in the following and in the definition of the semantics for pins in Section 7. Multiple `DataFlow` edges to an input or output pin represent multiple sources or destinations for the same data, respectively. This concept lowers the complexity while modeling because connecting a new data flow has one clear meaning: An additional flow to an input pin is an alternative flow. An additional flow from an output pin is another forked flow. A new mandatory input or output requires a new dedicated pin, to which the new data flow connects to. In Figure 2, we visualize multiple flows for the same pin by overlapping edges. For instance, the *book flight* process receives credit card information from two sources when considering the dashed data flow. These two flows are alternatives, so they connect to the same pin. To foster this clear meaning of data flows, all data flows have to go through pins. Compared to related work [29, 30] and our previous work [32], pins simplify adding additional, alternative flows because the flow just has to be added instead of integrated into existing behavior specifications of the node. Without pins, it was necessary to duplicate the specification of processing effects for these additional flows and to define the order, in which these flows shall be considered.

Data Processing Behavior (F3). We describe the data processing behavior of nodes by BehaviorDefinitions. A behavior definition is meant to be reusable to reduce the specification effort to be done by a security expert. In our running example, the behaviors *Declassify*, *Forward* and *Join* are behavior definitions shared between the various processes. In Figure 2, the letters in the processes indicate the reused behavior definition. Such a definition consists of input and output pins as well as Assignments of labels to output pins. A Term specifies whether a label shall be assigned. It can refer to labels of input pins or nodes as well as to constants. The set of assignments specifies the label propagation function. Our previous work [32] neither provides means to specify types of behavior specifications or means to reuse them. Related work provides fixed types of behavior definitions [29] or means to specify types [30]. Not considering types complicates the interaction between designers and security experts because security experts have to inspect every node in the DFD instead of only providing a few behavior types.

Actor and Store Behavior (F4). To cover behavior of actors and stores, we apply BehaviorDefinitions to these node types as well. Stores act like forwarding processes, i.e. they redirect all labels from the input to the output. Because we do not represent time or state in the model and systematically consider all possible incoming flows into the store, the forwarding behavior fits the semantics of a store that saves data and emits unchanged data. In our running example, the *Flight Storage* emits the same flights as the flights entered by the *FlightPlanner*. Actors usually use behaviors specific to them that cannot be reused. Additionally, we add the ActorProcess to describe complex data processing done by actors. These processes act like regular processes and can reuse behavior definitions but act on behalf of the actor. Consequently, the node properties, i.e. characteristics, of the actor, also apply to these processes. In our running example, the *select flight* process is an actor process because the user manually selects a flight from a list. In contrast to related work [29, 30], we represent actors and stores with dedicated elements, which we already did in previous work [32]. Additionally, we clearly distinguish the behavior of the system from the behavior of actors. This is beneficial because developers can distinguish parts to develop from parts only describing usage.

7. Semantics of Extended Data Flow Diagram

In the previous section, we introduced extensions to the DFD syntax to ease defining unambiguous semantics, which we introduce in this section. We define the semantics of the extended DFD by mapping it to clauses in first-order logic. We chose to formalize the semantics in first-order logic using Prolog because Prolog provides comprehensive capabilities of exploring all possible data flow paths, which we will discuss later. We explain the semantics in three steps: In Section 7.1, we recap foundational knowledge about Prolog. Section 7.2 explains how to map DFD elements to clauses in first-order logic. Afterwards, Section 7.3 discusses the resulting semantics of the logic program.

Listing 1: Examples of clauses used in Prolog.

```
1  cat(jane).          % fact with constant jane
2  bird(john).         % fact with constant john
3  chases(X,Y) :-      % rule taking arguments X and Y
4    cat(X),           % term testing whether X is a cat
5    bird(Y).          % term testing whether Y is a bird
6  sum(0,[]).          % fact stating that an empty list has sum 0
```

7.1. Foundations on Prolog

Analyses presented in this article rely on the semantics given by a transformation from the DFD into a logic program given in Prolog [65]. Prolog is an established logic programming language that requires a programmer to specify the knowledge to solve a problem rather than the procedure. A Prolog program consists of clauses [65, pp. 13], which can be facts or rules. Facts such as the ones shown in lines 1 and 2 of Listing 1 are always true. A rule is only true if all of the terms of its body are true. In Listing 1, line 3 is the head of the rule and the lines 4 and 5 are the body terms. Terms are constants, variables, lists and compound terms. Compound terms consist of a name and arguments, which are also terms. Facts and the head of a rule are compound terms. By convention, variable names are always upper case while constants are lower case. Quoted strings and numbers are constants as well. Lists are denoted by square brackets. Empty brackets mean empty lists as shown in line 6. In Prolog, rules are given as Horn clauses, i.e. the conjunction of terms in the body imply the term in the head. From a procedural point of view, the rule in line 3 to 5 can be read as follows: In order to prove `chases(X,Y)`, prove `cat(X)` first and `bird(Y)` second. Queries ask the program to find answers to a question. A query is a list of goals that Prolog interpreters try to solve. Goals and terms in the body of rules can be connected with a logical conjunction `,` or logical disjunction `;`. Negation `\+` is also possible but does not have the exact same meaning as negation in boolean logic [66, pp. 17]. Terms in queries can contain variables, for which the interpreter finds values that make all goals true. Informally speaking, Prolog interpreters find all instantiations of variables that make all goals true, which means that they can be deduced based on facts and rules. Selection Rule Driven Linear Resolution for Definite Clauses (SLD) [67, pp. 447] is the most commonly used resolution process for finding variable bindings in Prolog but detailed knowledge about that process is not required for the remainder of this article.

7.2. Mapping to Logic Program

In this section, we describe the mapping from the extended DFD syntax to clauses in first-order logic formulated in Prolog. To keep things simple, we focus on the fundamental principles but omit implementation details such as the helper clauses that are always added as a preamble to the mapping result. Additionally, we use simple identifiers instead of unique identifiers that would

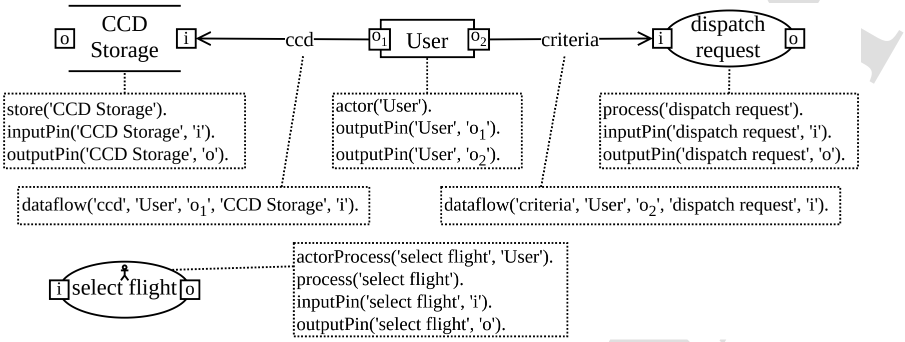

o
CCD Storage
i
ccd
o_1
User
o_2
criteria
i
dispatch request
o
store('CCD Storage').
inputPin('CCD Storage', 'i').
outputPin('CCD Storage', 'o').
actor('User').
outputPin('User', 'o_1').
outputPin('User', 'o_2').
process('dispatch request').
inputPin('dispatch request', 'i').
outputPin('dispatch request', 'o').
dataflow('ccd', 'User', 'o_1', 'CCD Storage', 'i').
dataflow('criteria', 'User', 'o_2', 'dispatch request', 'i').
i
select flight
o
actorProcess('select flight', 'User').
process('select flight').
inputPin('select flight', 'i').
outputPin('select flight', 'o').

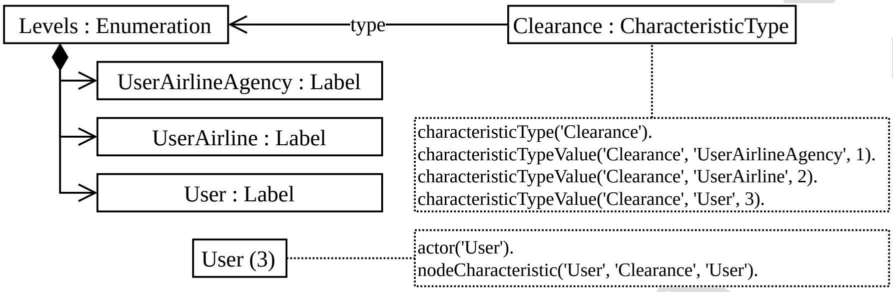
Figure 5: Mapping of structural DFD elements to clauses in first-order logic.

be hard to read in our examples. The full specification of the transformation is given by a model to model transformation in our data set [68].

DFD Nodes. First, we map the DFD nodes `Actor`, `Store` and `Process`, which DeMarco [33] introduced, to clauses. Figure 5 illustrates the mapping logic for these nodes and others that we describe later. The clauses only state that an element of the specified type exists with the given unique identifier. For instance, a store becomes a `store` clause with its identifier given as argument. Defining that elements exist is necessary to establish relations and specify further details such as behaviors as we will see later. For every node, we create one clause.

Actor Behavior (F4). An `ActorProcess` represents one activity, which an actor does. A set of such processes represents all activities done by an actor. The mapping of actor processes consists of two steps: First, we treat an actor process like a regular process, which means we generate a `process` clause as described before. By doing so, we can reuse all the logic for describing behaviors of nodes. Additionally, we do not need special logic for handling actor processes during label propagation. Second, we introduce an additional clause `actorProcess` stating that a process with given identifier belongs to an actor with a given identifier. This is necessary to find all activities of an actor. The clauses are visualized in Figure 5.

Multiple Inputs (F2). Our extended syntax supports multiple (alternative) inputs via `Pins` and `DataFlows` that refer to these pins. For every node, we create one clause for every pin specified in the `BehaviorDefinition` assigned to a node. We do not represent the `BehaviorDefinition` itself because its sole purpose is to make assignments and pins reusable. In Figure 5, input pins are visualized by squares containing the letter i at the border of the node. Output pins are visualized by squares containing the letter o at the border of the node. The pin clauses describe that there is an input or output pin with a given identifier on a node with a given identifier. For every data flow, we create one clause `dataflow` with a unique identifier as the first argument. The next two arguments describe the source node and the corresponding output pin. The last



```
characteristicType('Clearance').
characteristicTypeValue('Clearance', 'UserAirlineAgency', 1).
characteristicTypeValue('Clearance', 'UserAirline', 2).
characteristicTypeValue('Clearance', 'User', 3).
```

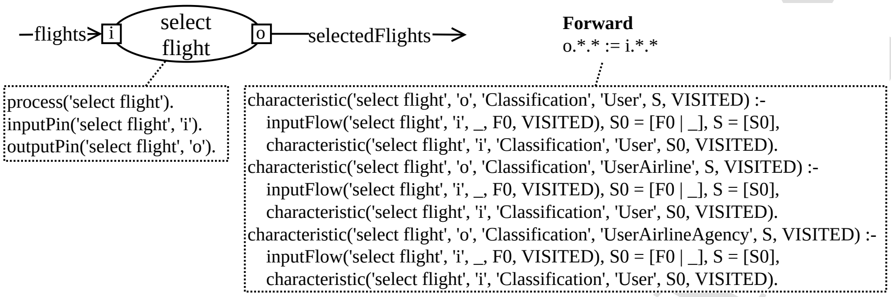
```
actor('User').
nodeCharacteristic('User', 'Clearance', 'User').
```

Figure 6: Mapping of characteristic types and characteristics to clauses in first-order logic.

two arguments describe the destination node and the corresponding input pin.

Node Characteristics (F1). Before we can map node characteristics, we first have to map the available types of characteristics. Characteristic types are also mapped to clauses stating their existence. As shown in Figure 6, we create one clause `characteristicType` for every characteristic type stating that there is a characteristic type with a given identifier. We do not represent enumerations because they only provide means for reusing labels while modeling. Instead, we create one clause `characteristicTypeValue` for every label transitively referenced by a characteristic type. The first argument specifies the characteristic type, the second argument specifies the label and the last argument specifies the index of the label in the enumeration. Naming the characteristic type and the label is necessary to establish a relation, i.e. to state that a certain label is a valid label for a certain characteristic type. A label is only unambiguous if it is used together with a characteristic type because a label can be reused in various characteristic types and can, therefore, have different meanings: In our running example, the meaning of the *User* level is different when used as classification or as clearance. Representing the index is beneficial because label comparison functions can refer to the order of the label via that index. Characteristics applied to a node are also represented by one clause for every label within a characteristic. In the example in Figure 6, the clearance level *User* is applied to the actor *User*. Thereto, we create one clause `nodeCharacteristic`, which states that the node *User* (given as first argument) has the label *User* (given as third argument) of the characteristic type *Clearance* (given as second argument) applied.

Node Behavior (F3). In the syntax, the node behavior is given by a sequence of assignments of truth values to boolean variables. The boolean variable on the left hand side defines whether one particular label, i.e. the tuple of characteristic type and label, is available at one particular output pin. The truth value on the right hand side can refer to labels on input pins, logic operations and constants. If no assignment specifies a truth value for a label, the default is that it is not available (false). The sequence of assignments represents the label propagation function. Representing labels as boolean variables is beneficial because first-order logic supports boolean variables and boolean expressions very well. In



```
process('select flight').
inputPin('select flight', 'i').
outputPin('select flight', 'o').
```

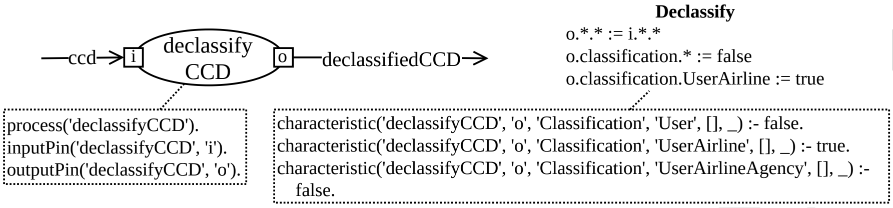
```
characteristic('select flight', 'o', 'Classification', 'User', S, VISITED) :-
   inputFlow('select flight', 'i', _, F0, VISITED), S0 = [F0 | _], S = [S0],
   characteristic('select flight', 'i', 'Classification', 'User', S0, VISITED).
characteristic('select flight', 'o', 'Classification', 'UserAirline', S, VISITED) :-
   inputFlow('select flight', 'i', _, F0, VISITED), S0 = [F0 | _], S = [S0],
   characteristic('select flight', 'i', 'Classification', 'User', S0, VISITED).
characteristic('select flight', 'o', 'Classification', 'UserAirlineAgency', S, VISITED) :-
   inputFlow('select flight', 'i', _, F0, VISITED), S0 = [F0 | _], S = [S0],
   characteristic('select flight', 'i', 'Classification', 'User', S0, VISITED).
```

Figure 7: Mapping of forward behavior of process to clauses in first-order logic.

the following, we explain how we represent boolean variables and how we map assignments.

We create one `characteristic` clause holding six arguments that represents a truth value for every label of a characteristic type on an output pin. Particular examples of these clauses are shown in Figure 7. The first two arguments identify the node and the output pin. The next two arguments identify the characteristic type and the label. The following two arguments are a flow tree `S` and a set of already visited flows `VISITED`. Roughly said, the flow tree contains the data flows connecting all transitive predecessors of a certain node. The leafs of the tree are always data flows from nodes without incoming data flows. There can be multiple trees for one node. If the `characteristic` clause evaluates to true, the label is available at the output pin for a particular flow tree and a particular set of visited flows. The flow tree is necessary to identify the data flows and nodes that lead to a violation. Without knowing this information, identifying the issue that lead to a violation would be hard. The set of visited flows prevents evaluation cycles in DFDs containing cycles. We explain both concepts (flow tree and visited flows) in more detail in Section 7.3.

Assignments describe when a label shall be available. The list of assignments contained in a `BehaviorDefinition` is ordered because an assignment that is defined later can override the effect of an assignment defined previously. `Terms`, which specify the right hand side of an assignment, cannot refer to labels on the output pins, which means they cannot refer to the boolean variables that the assignments change. Therefore, there is always only one assignment that determines the final truth value for a label on an output pin that does not depend on any previous assignment in the list of assignments. To simplify the mapping, we only consider that particular single assignment for building the body of the `characteristic` rule for the particular characteristic type and label. There is no point in representing other assignments than the so-called *last applicable* one because they do not affect the result of the label propagation. The mapping transforms the `Term` on the right hand side of the assignment to clauses in the rule body of the `characteristic` clause. Constants such as the ones shown in Figure 8 can be mapped to truth values. References to input labels are mapped to a `characteristic` clause referring to a label on an input


ccd
i
declassify CCD
o
declassifiedCCD
Declassify
o.*.* := i.*.*
o.classification.* := false
o.classification.UserAirline := true

```
process('declassifyCCD').
inputPin('declassifyCCD', 'i').
outputPin('declassifyCCD', 'o').
```

```
characteristic('declassifyCCD', 'o', 'Classification', 'User', [], _) :- false.
characteristic('declassifyCCD', 'o', 'Classification', 'UserAirline', [], _) :- true.
characteristic('declassifyCCD', 'o', 'Classification', 'UserAirlineAgency', [], _) :-
   false.
```

Figure 8: Mapping of declassify behavior of process to clauses in first-order logic.

pin as the mapping of the forwarding behavior in Figure 7 demonstrates: the label *User* shall be applied to the output pin if it is available on the input pin, which can be checked by the `characteristic` clause for the input pin (third line). To ensure traceability of the results, it is necessary to keep track of the data flows that have been considered while calculating the label, i.e. the flow tree. The `inputFlow` clause selects a data flow `F0` that shall be considered when determining the label for the input pin. `F0` will become the first flow in the flow tree. For the sake of brevity, we omit the implementation details of the `inputFlow` clause and the `characteristic` clause for input pins but refer to our data set [68] for the programs containing the full implementation.

7.3. Semantics of Logic Program

The goal of the clauses resulting from the previously defined mapping is to formalize data transmission and data processing by means of label propagation. Queries comparing propagated labels with expected labels prescribed by confidentiality requirements can identify violations as we show in Section 8. In the following, we explain the meaning of the previously introduced clauses in an informal way because a full formal discussion would require explaining all used helper clauses in detail, which is not possible within a reasonable amount of space. Instead, we just explain the effect of these helper clauses. The full specification is available in the logic programs in the dataset [68]. The underlying semantics for interpreting the logic programs are given by the SLD resolution algorithm [67, pp. 447] for first-order logic programs. Later, queries will also use the Prolog-specific all-solution predicates `findall` and `setof` [67, pp. 470]. The algorithm and the all-solution predicates have well-known and established semantics for first-order logic programs.

The majority of clauses have quite simple semantics: they state that an element of a certain type exists with a certain identifier. Additionally, some of the clauses described in the following establish relations between elements. The only clauses having complex semantics are the clauses covering node behaviors. We describe all clauses in the following.

DFD Nodes. The semantics of the clauses representing nodes is straight forward: the clauses for various node types simply mean that an element of the named type exists with a given identifier. For instance, the meaning of `process(N)` is that there exists a process with identifier `N`.

Actor Behavior (F4). The clauses representing actor processes only describe existence: The meaning of actorProcess(N, A) is that there exists an actor process with an identifier N that belongs to an actor with identifier A.

Multiple Inputs (F2). The clauses representing and involving pins only describe existence: The meaning of inputPin(N, PIN) is that an input pin PIN exists at the node N. The meaning of outputPin(N, PIN) is that an output pin PIN exists at the node N. The data flow clause states that a data flow from a source to a destination exists. dataflow(F, N_S, PIN_S, N_D, PIN_D) means that there exists a data flow F originating from pin PIN_S of node N_S and going to pin PIN_D of node N_D.

Node Characteristics (F1). The clauses covering characteristic types describe the existence of these types: The meaning of characteristicType(CT) is that a characteristic type CT exists. The meaning of characteristicTypeValue(CT, V, I) is that the characteristic type CT contains a label V at index I. The clauses representing node characteristics introduce a relation between a node and a label. nodeCharacteristic(N, CT, V) means that the label V belonging to a characteristic type CT applies to node N.

Node Behavior (F3). Node behaviors describe the label propagation functions of nodes. The previously described clauses define the structure of a DFD as directed graph of nodes and edges. Together, they build a label propagation network. The semantics of the label propagation are given by the characteristic clauses for input and output pins. We decided to realize the label propagation as label lookup to reduce the effort for considering multiple combinations of data flows. If we are only interested in the labels of one particular node, it is more efficient to follow data flows in reverse order. We can stop following data flows as soon as the label cannot be changed anymore. This is the case if an assignment only involves a constant because previous labels would be overridden by that constant assignment anyway. Therefore, we only consider nodes that actually change the labels. In contrast, a forward propagation would require us to evaluate all nodes and combinations of alternative data flows because we do not know yet whether the labels propagated by a node will eventually influence the labels of interest. This is costly in presence of alternative data flows. Besides the label lookup, we already identified further means for improving the performance of the analysis in a student's thesis [69]. These optimizations, however, increase the complexity of the mapping to the logic program as well as the logic program itself and are, therefore, subject to future research. Because we did not experience a performance issue in non-synthetic systems and, especially, not in the realistic systems of our evaluation, we did not include these optimization for the sake of comprehensibility.

The characteristic clause for input pins shown in Listing 2 is part of various helper clauses added as preamble to the mapping result of the previous section. The labels available at an input pin solely depend on the labels available at the output pin that a data flow connects to the input. Lines 2 and 3 find a data flow F that connects an output pin PIN_S to the input pin PIN. To avoid evaluation cycles, only data flows not already visited are considered in the next line. In the last line, the truth value of the label V of the output pin PIN_S is

Listing 2: Prolog rule for finding labels on input pins.

```
characteristic(N, PIN, CT, V, [F | S], VISITED) :-
  inputPin(N, PIN),
  dataflow(F, N_S, PIN_S, N, PIN),
  intersection([F], VISITED, []),
  characteristic(N_S, PIN_S, CT, V, S, [F | VISITED]).
```

just copied. In the same step, the set of visited flows `VISITED` is extended by the used data flow.

The major benefit of the SLD resolution algorithm used in Prolog is that it can find all possible variable bindings, i.e. all possible labels available via all possible data flow trees, by reevaluating the clause. This is important if there are multiple data flows connected to the same input pin and it also addresses the corresponding challenge Ch1 of systematically considering all possible data flow paths. A label is only available for a certain node *and* a certain data flow tree. In the running example, the *book flight* process has two alternative data flows providing credit card details. The reevaluation by Prolog automatically considers both data flows but only the direct flow of credit card details leads to a violation. A data flow tree, as introduced in the section before, can be seen as an acyclic subgraph of the DFD only representing nodes and data flows that potentially affect the labels available at a certain node. There are no alternative data flows contained in such a data flow tree but always exactly one choice for every alternative flow. Therefore, all data flow trees for the *book flight* process contain either the direct flow or the declassified flow but never both. This is important for identifying the underlying issue of a reported violation.

The `characteristic` clause for output pins has the same arguments as the clause for input pins but the body depends on the particular assignments as motivated in Section 7.2. The meaning of constant assignments is that the label is always available (true) or is never available (false) independent of the particular data flow tree or visited flows. The meaning of logical operators is equivalent to their intuitive meaning, e.g. the `And` operator translated to `,` means that both operands have to evaluate to true in order to become true. The meaning of references to node characteristics is that the particular label has to be available at the node, i.e. there has to be a `nodeCharacteristic` clause for the particular node and label. The meaning of references to characteristics of incoming data is that the particular label has to be available at the referenced input pin, i.e. the `characteristic` clause for the particular input pin, label and data flow tree has to evaluate to true. Again, Prolog considers all possible data flow trees when looking for labels. The data flow tree `S` initially consists of one particular data flow for every input pin. The resolution of further clauses extends this data flow tree until it contains all relevant data flows.

Listing 3: Prolog API to specify comparison functions.

```
1 actor(N), store(N), process(N), actorProcess(N,A),
2 inputPin(N,PIN), outputPin(N,PIN),
3 characteristicType(CT),
4 characteristicTypeValue(CT, CV, I),
5 flowTree(N, PIN, S),
6 traversedNode(S, N),
7 nodeCharacteristic(N, CT, CV),
8 characteristic(N, PIN, CT, CV, S).
```

8. Definition and Execution of Label Comparison Function

Extended DFDs as described before can be analyzed for violations of access control and information flow requirements. To do that, the automated model transformation described in the section before translates the DFD into a logic program given in Prolog. The label comparison function is a query to the Prolog program. Queries compare labels of received data with labels of other data or nodes. Prolog automatically considers all data paths via backtracking (Ch1), which means that all possible sets of labels that can be found via all possible data flow paths are considered in the comparison. The label comparison function is part of the analysis definition introduced in the approach overview in Section 5. We focus on the label comparison function in this section because we already motivated and explained the other elements of the analysis definition, namely the node properties, the property types used for data and the behavior descriptions. In the following, we recapture the Prolog clauses that security experts can use to define label comparisons. Afterwards, we define the query for our running example. For the complete logic program of the running example, please refer to our data set [68]. We create and discuss further queries as part of our evaluation in Section 10.3.

Security experts define queries using the clauses in Listing 3. Line 1 gives clauses to find identifier `N` representing actors, stores or (actor) processes. Line 2 gives clauses for finding identifier `PIN` of input or output pins. Line 3 gives a clause to find identifier `CT` of a characteristic type. Line 4 gives a clause to find label identifier `CV` of characteristic type `CT` with order number `I`. Line 5 gives a clause to find a flow tree `S` consisting of all data flows that potentially contributed labels to pin `PIN` of node `N`. The tree chooses exactly one data flow at any pin having multiple, alternative data flows. Line 6 gives a clause to check whether node `N` is visited when following flow tree `S`. Line 7 gives a clause to find label `CV` of characteristic type `CT` that is active on node `N`. Line 8 gives a clause to find label `CV` of characteristic type `CT` that is present on pin `PIN` of node `N` when choosing flow tree `S`. Please note that the clause given in line 8 is a shorthand for the characteristic clause introduced in the previous section that uses an initial empty list of already visited data flows. This is reasonable because no flows have been visited yet when a label lookup starts at one particular node.

Queries are tailored to policy types such as RBAC policies or non-interference policies. A policy is a set of confidentiality requirements. A policy type pre-

Listing 4: Information flow analysis for totally ordered labels.

```
1 ?- nodeCharacteristic(P, 'clearance', V_CLEAR),
2 characteristicTypeValue('clearance', V_CLEAR, N_CLEAR),
3 inputPin(P, PIN),
4 characteristic(P, PIN, 'class', V_LEVEL, S),
5 characteristicTypeValue('class', V_LEVEL, N_LEVEL),
6 N_CLEAR < N_LEVEL.
```

Listing 5: Reported information flow violations for the running example.

```
1 P = 'Booking Storage', V_CLEAR = 'User,Airline', N_CLEAR = 2, PIN = 'input',
  V_LEVEL = 'User', N_LEVEL = 3, S = ['booking', ['selectedFlight'],
  ['declassifiedCCD', 'selectedFlight'], ['ccd direct', ['ccd', ['ccd']]]]];
2 P = 'process booking', V_CLEAR = 'User,Airline', N_CLEAR = 2, PIN = 'input',
  V_LEVEL = 'User', N_LEVEL = 3, S = ['declassifiedCCD', ['selectedFlight'],
  ['ccd direct', ['ccd', ['ccd']]]] .
```

scribes the structure of confidentiality requirements that may be used in a policy. Therefore, the first step is to define a violation in the context of the policy type. An information flow policy with totally ordered levels is violated if someone with clearance l_{clear} accesses data with classification l_{class} bigger than the clearance $l_{clear} < l_{class}$. In terms of our semantics, we have to find and compare the clearance label of a node with the classification label of its input pins. In the second step, we encode this detection rule by the query shown in Listing 4. In line 1, we determine the clearance level V_CLEAR of a node P. Line 2 determines the position N_CLEAR of the clearance level V_CLEAR in the enumeration. We defined the levels in ascending order, so the level with a lower index is semantically lower than a level on a higher index. Line 3 finds an input pin PIN for node P. The classification level V_LEVEL is determined in line 4. The order number N_LEVEL of the classification level V_LEVEL is determined in line 5. Line 6 tests for $l_{clear} < l_{class}$.

The DFD as modeled in Figure 2 does not contain an information flow violation when not considering the dashed edge. Considering the dashed edge, we detect the two violations shown in Listing 5. The first result in line 1 detects that the *Booking Storage* receives data on its input that is classified higher than its clearance. The same violation is detected for the *process booking* process in line 2. In both cases, we can find the cause of the violation in the data flow tree S, which contains *ccd direct*. This data flow directly transfers the credit card data without declassification, which causes the violation. Therefore, we can trace back both violations to the introduced issue given by the *ccd direct* data flow.

Specifying queries requires security expertise. However, designers do not need this competence. They can reuse defined queries from an existing analysis definition that also contains characteristic types, characteristics of nodes and behavior definitions. Security experts can create these reusable elements and put them in a catalogue structured by the particular policy types. After that, designers can select the elements and make use of them without the need for security expertise. For instance, the query presented in this section does not depend on particular levels. Therefore, it is applicable to information flow policies consisting of arbitrary totally ordered levels. How many elements of analysis definitions for such policy types are reusable depends on how tailored they are to the use case. For instance, the clearance and classification levels defined for our running example are tailored to the example, so they are reusable but require renaming to fit another system.

9. Tool Support

We realized the previously presented concepts to show that an implementation is feasible (Böhme and Reussner [70] call this a level 0 validation) and to support the evaluation described in Section 10. This article is not meant to be a technical report, so we only briefly report on our tooling. Our data set [68] gives more details about the tooling. The full implementation is available in various projects on GitHub, which we describe in the following.

First of all, we realized all metamodels[1]described in this article in the Eclipse Modeling Framework (EMF) [71] and defined appropriate invariants to specify the well-formedness of DFDs in more detail. For instance, invariants ensure that a data flow always originates from an output pin and leads to an input pin, which both must not belong to the same node. Using EMF automatically provides us with ready to use editors. The metamodel projects on Github also contain an enhanced graphical editor that adopts the classic DFD syntax. Designers can reuse elements defined for particular policy types by referencing catalogue models.

To automate the detection of violations, we realized the mapping to the Prolog program as model-to-model transformation in Xtend [72] and implemented an adapter[2] to run Prolog interpreters. The transformation has about 480 LLOC in total, which includes about 130 LLOC for adding the static preamble to the logic program. LLOC covers all lines containing a statement. We also created a metamodel[3] for Prolog programs as well as a model printer to serialize the program and a model parser to parse results based on Xtext [72]. The query is executed in the commonly used SWI Prolog interpreter [73] that we connected to our prototype via the implemented adapter. Therefore, users do not have to interact with the interpreter directly.

We decided to specify the analysis directly in Prolog because the resulting specification is self-contained and executable. It is easy to find the concepts introduced as part of the semantics definition in Section 7 within the analysis program, so there is no gap in abstraction. We could also have used existing

[1] https://github.com/FluidTrust/Palladio-Supporting-DataFlowDiagram
https://github.com/FluidTrust/Palladio-Supporting-DataFlowDiagramConfidentiality

[2] https://github.com/FluidTrust/Palladio-Supporting-Prolog4J

[3] https://github.com/FluidTrust/Palladio-Supporting-Prolog

model checking approaches [74] but this would not free us from a model transformation into a particular formalism or at least a special encoding of the logic to discover multiple data flow paths (Ch1).

To ease writing Prolog queries, we developed a Domain-specific Language (DSL) [75] that is capable of formulating common queries without the need to adhere to the Prolog syntax or even be aware of Prolog. When formulating the query with the DSL, it is also possible to process the interpreter result directly and report the detected violations in terms of the DFD, which is know to the designer. The prototype of the DSL is still under development and not ready to use yet, so we did not use or evaluate it as part of this article.

10. Evaluation

In this section, we evaluate our aforementioned contributions. We present our evaluation goals and metrics in Section 10.1. The evaluation design is described in Section 10.2. In Section 10.3, 10.4 and 10.5, we discuss results. We discuss threats to validity in Section 10.6 and limitations in Section 10.7. We report on the availability of evaluation data in Section 10.8.

10.1. Evaluation Goals and Metrics

We structure our evaluation according to the *Goal-Question-Metric* methodology [76, 77]. We formulate three evaluation goals.

G1) Evaluate the expressiveness of our syntax and semantics to represent and analyze systems using information flow and access control.

G2) Evaluate the reusability of DFDs when switching confidentiality mechanisms.

G3) Evaluate the accuracy of confidentiality analyses realized with our semantics.

We evaluate expressiveness, reusability and accuracy. Expressiveness describes what confidentiality mechanisms our approach can express. We want to evaluate expressiveness to see whether the approach supports information flow and access control (Ch2). The evaluation of expressiveness also shows that we addressed the challenge of enabling custom analyses (Ch3) because we do not limit ourselves to predefined confidentiality mechanisms and analyses but use extensions to cover confidentiality mechanisms. We evaluate reusability of DFD parts when switching confidentiality mechanisms to show that our approach reduces the amount of elements, which have to be recreated. This was one motivation for developing the extended DFD syntax for covering information flow and access control within one modeling language (Ch2). We evaluate accuracy because expressiveness and reusability are only useful if resulting analyses have satisfying accuracy, which means designers can identify violations. To provide accurate analyses, it is necessary to systematically consider all possible data flow paths, i.e. combinations of these data flows. Otherwise, violations might not be discovered. Because the evaluated system designs contain multiple data flow paths, evaluating the accuracy of the analyses is appropriate to show that we addressed the challenge of considering all data flow paths (Ch1).

To evaluate G1, we formulate the following evaluation questions:

Q1.1) Is the proposed syntax capable of representing systems as well as their properties and their behaviors relevant for identifying access control violations?

Q1.2) Is the proposed syntax capable of representing systems as well as their properties and their behaviors relevant for identifying information flow violations?

Q1.3) Are the proposed semantics capable of defining analysis queries for identifying access control violations?

Q1.4) Are the proposed semantics capable of defining analysis queries for identifying information flow violations?

To answer the questions for G1, we use the *syntactic quality* metric $s = {}^{|R \cap E|}/_{|R|}$ as defined by Boyd et al. [78] for rating the quality of constrained natural languages. The metric is also usable for rating a DSL [79], which fits to the DFD metamodel presented in this paper. In our context, a *language requirement* $r \in R$ is a DFD or analysis query that we would like to express. The set of expressions E contains every possible DFD or analysis query that can possibly be constructed using our artifacts. The metric value ranges from zero (no DFD or analysis query could be expressed) to one (all DFDs or analysis queries could be expressed).

To evaluate G2, we formulate the following evaluation questions:

Q2.1) How much DFD elements can be reused when switching between confidentiality mechanisms?

To answer Q2.1, we calculate the similarity coefficient according to Jaccard $j = {}^{|M \cap N|}/_{|M \cup N|}$ [80] between the models (M and N) of the cases that represent the same system but use different confidentiality mechanisms. A model is defined as set of model elements, i.e. instances of meta-classes. A model element $m \in M$ is equal to a model element $n \in N$ if the type and all properties of the model elements are equal. We determine this equality of model elements by applying EMFCompare [81]: First, we match model elements by their identifiers. Afterwards, we compare their properties. A reference to another model element is such a property. References are considered equal if they refer to equal model elements. The coefficient is simple but is a good measure of the amount of unchanged model elements and consequently also on the amount of elements, which have to be changed when switching the used confidentiality mechanism in a system design. The metric value ranges from zero (every element is different and has to be recreated) to one (the models are equal and nothing has to be recreated). The coefficient of Jaccard is appropriate to rate the similarity of software design models as we have shown in previous work [82, 83].

To evaluate G3, we formulate the following evaluation question:

Q3.1) What is the accuracy of the analyses?

To answer Q3.1, we apply the commonly used metrics precision $p = {}^{t_p}/_{(t_p+f_p)}$ and recall $r = {}^{t_p}/_{(t_p+f_n)}$ with the number of true positives t_p, false positives f_p and false negatives f_n. We describe the classification of results as t_p, f_p or f_n in the evaluation design.

We intentionally do not evaluate usability or correctness of the modeling and analysis approach. Usability is usually evaluated in user studies that evaluate the tool support and the concrete syntax used for modeling. We neither aim for evaluating our implementation nor for a particular concrete syntax because both are no contributions of this paper. We do not verify correctness because this would not provide insights into the application of our approach and how well the approach addresses the challenges (Ch1, Ch2 and Ch3). Instead, a case study provides such insights in the context of realistic systems, which is the objective of this paper.

10.2. Evaluation Design

Evaluations based on case studies are the second most common evaluation approach for security notations after just illustrating how to use notations and analyses as van den Berghe et al. [84] point out. Especially with respect to expressiveness and reusability, a detailed discussion of established cases provides more insight than a generic discussion about hypothetical systems. Therefore, we evaluate the proposed syntax and semantics based on a case study. We select cases from related work [29, 85] and from one of our previous publications [32] or define new cases if there are no appropriate cases available. A case is a pair of a system design and confidentiality requirements. In the following, we discuss the evaluation design per evaluation goal before we discuss cases and their selection.

Expressiveness. For evaluating expressiveness, we model the system design as DFD and the corresponding analysis query using our semantics for each case. The procedure described in the following is the same for both access control and information flow control. 1) We identify relevant data and node properties, i.e. the labels and the corresponding characteristic types. 2) We identify relevant behavior descriptions including the label propagation rules. 3) We model the system design as DFD by using the behaviors defined before. 4) We define the analysis query for identifying violations. After step 3, we finished modeling the system design, so we can calculate the syntactic quality metric and answer Q1.1 and Q1.2. A requirement as specified by the metric is a thing that shall be expressed by a modeling language. In our evaluation, one case is one requirement, i.e. a thing to be expressed according to the definition of syntactic quality by Boyd et al. [78]. This means, the DFD metamodel has to be capable of representing the whole case or the whole case will be counted as not expressible. After step 4, we finished the analysis definition, so we can calculate the syntactic quality metric and answer Q1.3 and Q1.4.

We build weighted sums while calculating the syntactic quality metric. The weighted sums normalize the influence of cases that use different system designs but share the same analysis type. Without such a normalization, a single case using a not supported analysis type can be hidden by a group of cases sharing the same but well-supported analysis type. For instance, the information flow cases *TravelPlanner*, *DistanceTracker* and *ContactSMS* from related work [85] share the same analysis definition representing noninterference with declassification using totally ordered security levels. If our approach supports this analysis type well but does not support another analysis type, that is only used by one case, the value of the syntactic quality metric would be ${}^{3}/{}_{4}$. However, we are, especially, interested in the support of confidentiality mechanisms. Therefore, the metric value using a weighted sum ${}^{\frac{1+1+1}{3}}/{}_{\frac{1+1+1}{3}+1} = 0.5$ would be more appropriate. As illustrated, we group cases by their type of analysis definition. We sum up the amount of fully modeled cases and divide this sum by the amount of cases in the corresponding group. Eventually, we sum up all of these weighted sums and divide it by the number of different types of analysis definitions.

Reusability. To answer Q2.1, we identify cases that are about the same systems but that use different confidentiality mechanisms. This applies to the cases using the previously mentioned systems *TravelPlanner*, *DistanceTracker* and *ContactSMS*: For each system, there exists one case using RBAC and another case using noninterference with totally ordered levels. For every such pair of cases, we calculate the Jaccard Coefficient by comparing the model elements. We use EMFCompare [81] to compare the model elements in order to identify equal and unequal model elements. The comparison approach of EMFCompare provides the necessary steps to decide whether two model elements are equal: In a first step, matching elements are identified by comparing their identifiers. This is reasonable because we copied and adjusted the models to switch the confidentiality mechanism. This is also the approach designers would most likely do. In a second step, differences are calculated, which covers all properties of the model elements. As a result, we receive a list of differences. We walk through that list and add all model elements that have been matched and that have no changed property to the set of equal elements $M \cap N$. The metric indicates a benefit compared to the state of the art if the value is above 0.

Accuracy. The accuracy evaluation reuses the previously created DFDs and analysis queries. The procedure described in the following is the same for access control and information flow. 1) We identify a way to introduce an issue into the DFD that leads to violations with respect to the defined analysis. We derive the issue from related work or by defining a new issue if no issue is reported in related work. We describe how we did that for every case in the description of the case selection below. 2) We inject the issue into the DFD of the case. The issue is usually introduced by an additional data flow. Therefore, the analysis has to consider multiple data flow paths. 3) We execute the analysis and classify the results.

To calculate the accuracy metrics, we classify the violations, which our analysis reports. A reported violation is valid if it traces back to the injected issue. A reported violation is invalid if it does not trace back to the injected issue. Because the DFD does not contain an issue before we inject an issue, it is reasonable to trace back violations to exactly the one known issue. A violation traces back to an issue if the injected data flow is in the flow tree of the violation. We classify the set of reported violations per case to avoid that large cases with many reported violations for one analysis type hide the violations of smaller cases for another analysis type in the metric. If all reported violations are valid, the case is counted as a true positive t_p. If at least one violation is invalid, the result is a false positive f_p. Not reporting any violations is a false negative f_n.

The reason for classifying all violations together is that analyses do not only report one but multiple violations. This is no flaw in our analysis but the logical consequence of propagating data through the system: if data must not be used in one node, the chances are high that it must not be used in following nodes as well. In our running example, the analysis reports two violations: one violation at *process booking* and one violation at the store, into which the process writes the data. Related work [35, 29] often only discusses why a violation occurs, i.e. the root cause of a violation, but does not discuss individual occurring violations. In contrast, our approach reports violations but no root cause. Again, this is no limitation of our approach because a root cause is a design decision that has to be changed in order to meet confidentiality requirements. Neither our nor other approaches can free software designers from choosing a solution because this is a creative process. Doing this automatically is barely possible. Therefore, we have to bridge the gap between the set of individual violations that our approach yields and the root causes that related approaches report in their publications. We do this by ensuring that every reported violation traces back to the issue we introduced. We already demonstrated how to trace back issues in Section 8.

Case Selection for Information Flow. There are various security models based on information flow but noninterference is one of the most commonly used models [86], which can be extended by declassification to increase its applicability. Related approaches [29, 85] also use this security model and provide cases including points to insert issues. These cases support our evaluation because they provide data-oriented system descriptions, define information flow requirements based on data and provide reference results, issues or critical points to inject issues for rating the accuracy of analysis results. All cases consider declassification and are based on real systems. We select all cases presented in the mentioned publications. Katkalov [85] provides five cases with flow requirements: *TravelPlanner*, *DistanceTracker* and *ContactSMSManager* cover noninterference with declassification using totally ordered security levels (OL). The information flow analysis ensures that no data arrives at a node that has a clearance level lower than the data classification. *PrivateTaxi* covers fine-grained noninterference rules between nodes and selected data types (LG). *BankingApp* covers noninterference between tenants of a banking system. All aforementioned cases of Katkalov do not provide reference results in form of a set of violations or cases containing issues. However, they describe the critical point, i.e. a declassification function, in the design that prevents violations. Therefore, we introduce an issue by circumventing these declassifications. Tuma et al. [29] provide the four cases *FriendMap*, *Hospital*, *JPmail* and *WebRTC* that cover noninterference analyses with two security levels (2L). The

Table 3: Characteristics of information flow cases (top) and access control cases (bottom) realized in our DFD syntax.

Case	Analysis	Nodes	Edges	Behaviors	Characteristic Types	Labels
TravelPlanner	OL	17	19	7	2	3
DistanceTracker	OL	8	9	5	2	3
ContactSMS	OL	9	12	7	2	2
PrivateTaxi	LG	35	54	11	6	6
BankingApp	–	–	–	–	–	–
FriendMap	2L	17	18	7	3	4
Hospital	2L	14	14	7	3	4
JPMail	2L	15	17	9	3	4
WebRTC	2L	50	56	10	3	4
TravelPlanner	RBAC	17	19	7	2	3
DistanceTracker	RBAC	8	9	5	2	3
ContactSMS	RBAC	9	12	7	2	3
DAC	DAC	7	7	6	4	4
MAC	MAC	15	22	7	2	3
ABAC	ABAC	13	18	6	4	6

information flow analysis ensures that no data classified high arrives at a node observable by an attacker. Tuma et al. provide a variant with and a variant without issue for the cases *FriendMap* and *Hospital*. We use both variants, so we do not have to introduce an issue by ourselves. For the remaining cases, the critical points in the design, i.e. the declassifications, are available. We introduce an issue into every case by circumventing the declassification. The upper part of Table 3 gives an overview of the size of the cases after realizing them with our syntax. The publications describe the cases in more detail than we can provide in this article, so we refer to the respective publications and our data set [68] for detailed descriptions.

Case Selection for Access Control. Access control and corresponding analyses are a wide field. Unfortunately, finding cases that are neither about correctly implementing access control systems nor designing appropriate requirements is challenging. The related approach FlowUML [61] does not provide an evaluation and therefore no cases. In previous work [32], we provide three RBAC cases derived from the already known cases *TravelPlanner*, *DistanceTracker* and *ContactSMS* by mapping the security levels to roles. The RBAC analysis ensures that every node holds at least one role that a data item requires to grant access. The cases support our evaluation by covering various system designs and providing an analysis covering the core of RBAC. We introduce the same issues in the access control cases that we already introduced in the information flow cases for the same reasons. There are further three common access control models [87, pp. 61], for which we could not identify appropriate cases in literature: DAC, MAC, and Attribute-based Access Control (ABAC). Therefore, we create one case for each access control model on our own. We use a textbook [87] that describes the foundational concepts of these models. The cases support our evaluation because they are designed to cover the remaining, most common access control models, which we have to consider to reason about expressiveness. The lower part of Table 3 gives an overview of the size of the cases after realizing them with our syntax. In the following, we describe the cases created by us. Our data set [68] contains additional details about the cases.

DAC Case. Discretionary Access Control (DAC) [87, pp. 61] directly assigns access privileges on objects to the accessing subjects. The case covers these aspects: The DFD describes a system consisting of a storage of family pictures and a system function to read the pictures as illustrated by Figure A.10. The DFD reflects common usage scenarios of DAC in operating systems or file sharing systems. There are four users: *Mother*, *Dad*, *Aunt* and *Indexing Bot*. The mother is the owner of the pictures. She grants read access to all but the bot. Consequently, the index bot must not access the storage. The introduced issue is that the index bot accesses the pictures.

MAC Case. Mandatory Access Control (MAC) [87, pp. 64] defines mandatory, global rules that aim for avoiding unwanted explicit information flows. The military access control model is one of the most prominent examples for MAC. Therefore, we assume that this particular model is a representative example for MAC. Military information systems often use MAC requirements prohibiting access to information classified higher than the user's clearance. The case is about such a system: The DFD describes a system for monitoring the airspace using the military access control model [87, pp. 65] as illustrated by Figure A.11. There are three user types: *Clerks* have the clearance *Unclassified*. They create and store weather reports. *Flight Controllers* have the clearance *Classified*. They register civil planes, look them up in a database and determine new routes for them by considering weather reports. *Military Flight Controllers* have the clearance *Secret*. They do the same as the civil flight controller but for military planes by also considering positions of civil planes. Information about weather is *Unclassified*, information about civil planes is *Classified* and information about military planes is *Secret*. The levels have the total order *Unclassified*, *Classified* and *Secret*. The introduced issue is that the civil flight controller reads military plane information.

ABAC Case. Attribute-based Access Control (ABAC) [87, pp. 74] describes subjects and objects by attribute descriptors rather than roles or identity. Access control permissions are defined between subject descriptors and object descriptors. The case covers these aspects: The DFD describes a system design for managing customers of a bank with branches in the USA and Asia as illustrated by Figure A.12. There are *Clerks* that register customers, look them up and determine credit lines for them. A clerk has the attributes *Role* and *Location*. *Managers* have the same abilities and properties as a clerk but can also register celebrity customers and move customers between branches. Processed information has the attributes *Customer Status* and *Customer Location*. The access permissions are defined as follows. Users with a certain location can access information about customers that are in the same location and that are not celebrities. Users that have the role manager can access all information. Any other access is forbidden. The introduced issue is that a manager registers a celebrity as regular customer.

10.3. Evaluation Results and Discussion of Expressiveness

We could successfully model all system designs including properties and behaviors relevant for confidentiality for all cases mentioned in Table 3 except for the *BankingApp* case. We explain why we could not model the *BankingApp* case as part of the discussion below. The syntactical quality of the access control cases (Q1.1) is $s = {}^{\frac{3}{3}+1+1+1}/_{\frac{3}{3}+1+1+1} = 1.0$. The syntactical quality of the information flow cases (Q1.2) is $s = {}^{(\frac{3}{3}+1+0+\frac{4}{4})}/_{(\frac{3}{3}+1+1+\frac{4}{4})} = 0.75$. As explained as part of the evaluation design, we normalized the influence of multiple cases using the same analysis type by building a weighted sum. We could fully represent the analysis definitions for access control (Q1.3), which implies a syntactical quality $s = {}^{(\frac{3}{3}+1+1+1)}/_{(\frac{3}{3}+1+1+1)} = 1.0$. We could fully represent the analysis definitions for information flow (Q1.4) except for the definition of the *BankingApp* case. We explain why we could not represent the analysis definition of the *BankingApp* case as part of the discussion below. The syntactical quality is $s = {}^{(\frac{3}{3}+1+0+\frac{4}{4})}/_{(\frac{3}{3}+1+1+\frac{4}{4})} = 0.75$. In the following, we discuss the modeling results and examine the reason for reduced syntactical quality. We do not present the resulting DFDs but focus on the used characteristic types, behavior descriptions and the analysis queries because they are the crucial parts that have potential to limit expressiveness. As introduced in Section 5, we refer to the combination of these three things as *analysis definition*. The full DFDs are available in our data set [68].

The cases *TravelPlanner*, *DistanceTracker* and *ContactSMS* share the same analysis: non-interference using a totally ordered lattice (OL). The characteristic types are the *classification* of information and the *clearance* of nodes. Both use totally ordered security levels. The behavior descriptions are as follows. A *Forwarder* copies the classifications from input to output. A *Store* acts like the forwarding behavior. A *Joiner* merges two inputs into one and classifies the output by the highest of all incoming levels. A *Syncer* acts like the forwarding behavior but waits for an additional input without considering its classification. A *Declassifier* explicitly sets the classification of the output. The analysis query is the same as already presented in Listing 4. We could successfully represent all three cases, which includes the system designs and analyses. The presented analysis definition is applicable to all noninterference analyses including declassification that have totally ordered security levels.

The cases *FriendMap*, *Hospital*, *JPMail* and *WebRTC* share the same analysis: non-interference using high/low levels (2L). The characteristic types are the *classification* of information, the *classification of encrypted content* and the *zone* of nodes. The classification characteristic type uses the values *high* and *low*. The zone characteristic type uses the values *attack* and *trusted*. The behavior descriptions are as follows: *Store*, *Forwarder* and *Joiner* share the semantics

Listing 6: Information flow analysis query equivalent to Tuma et al. [29].

```
?- inputPin(P, PIN),
nodeCharacteristic(P, 'zone', 'attack'),
characteristic(P, PIN, 'classification', 'high', S).
```

already described for the previous cases. The *Encryptor* always sets the classification of the output to low but attaches the old classification in the *classification of encrypted content* characteristic. The *Decryptor* sets the classification of the output to the classification stored in the *classification of encrypted content* characteristic. The analysis query shown in Listing 6 searches for data with a high classification that arrives on a node `P` in the attack zone. We could successfully represent all four cases, which includes the system designs and analyses. The presented analysis definition is applicable to all noninterference analyses including declassification by encryption that use two classification levels and only distinguish regular and attacking system nodes or users.

The *PrivateTaxi* case is complex and covers non-interference using lattice groups (LG). It requires a decent amount of characteristic types and behaviors. The characteristic type *PublicKeyOf* and *PrivateKeyOf* describe that the information is a public key or a private key of an entity. *DecryptableBy* describes the entities that can decrypt the encrypted information. *Entity* describes that a node belongs to an entity. All of these characteristic types use a list of entities as values. The characteristic type *CriticalData* describes that a data type requiring protection is contained in the information. *EncryptedContent* describes the content of encrypted information. Both characteristic types use a list of data types as values. The *Store*, *Forwarder* and *Syncer* behavior are as explained previously. The *Joiner* determines the output characteristics by building the union of received labels for each characteristic type except for the decryptable characteristic type, which requires the intersection of labels. The *Encryptor* stores the critical data type in the characteristic for encrypted content, removes the critical data type characteristic, and sets the decryptable characteristic to the owner of a received public key. The *Decryptor* inverts the effect of the *Encryptor* if the decryptable characteristic matches the owner of a received private key. There are two behaviors that declassify data: The *Proximity* behavior acts like the forwarding behavior but removes the critical data type label for routes because the route cannot be reconstructed from a single valued metric. The *RouteCreator* behavior creates routes from a location and a destination. It acts like the joining behavior but explicitly sets the critical data type characteristic to *route*. The analysis query shown in Listing 7 tests whether either the service for calculating distances has access to contact information or the private taxi service has access to the route. We could successfully represent the case, i.e. the system design and the corresponding analysis. The behaviors to handle encryption are reusable but the characteristic types and the analysis goal are tailored to the case. The reason for this is the explicit reference to nodes in the analysis goal as defined by Katkalov [85, p. 211].

Listing 7: Information flow analysis query for PrivateTaxi case.

```
?- (E = 'CalcDistanceService', D = 'ContactData';
E = 'PrivateTaxi', D = 'Route'),
inputPin(N,PIN),
nodeCharacteristic(N, 'entity', E),
characteristic(N, PIN, 'criticalData', D, S).
```

Listing 8: RBAC analysis query for iFlow cases.

```
?- inputPin(P, PIN), flowTree(P, PIN, S),
setof(R, nodeCharacteristic(P, 'Roles', R), ROLES),
setof(A, characteristic(P, PIN, 'AccessRights', A, S), REQ),
intersection(REQ, ROLES, []).
```

As mentioned before, we could not fully express the *BankingApp* case. The information flow requirements to be considered in this case are about ensuring that tenants/users of a banking app including the banking backend system do not interfere with each other. For instance, a user must not have access to the balance of another user. While we could represent the system structure consisting of processes, the actor and stores, we could not represent the remaining system aspects such as multiple users of the same type. Consequently, we could also not represent the analysis query. We cannot represent multiple users because the DFD model and the semantics operate on a type-level. However, representing multiple users of the same type requires models and semantics operating on instance-level. We discuss this aspect as part of the limitations in Section 10.7.

The access control versions of the cases *TravelPlanner*, *DistanceTracker* and *ContactSMS* share the same analysis: Core RBAC. The characteristic types are *AccessRights* of data and *Roles* of nodes. Both use three available roles as values. The behavior types are the same as described for the corresponding information flow cases. The *Joiner* applies the intersection of access rights of incoming data to the output. The *Declassifier* copies the access rights including a defined additional access right to the output. The remaining behaviors remain the same. The analysis query illustrated in Listing 8 collects all access rights `REQ` of a data item, collects all roles `ROLES` of a processing node, and reports a violation if the intersection between access rights and roles is empty. We could successfully represent all cases, i.e. the system design and the corresponding analysis. The analysis definition can be reused to represent access control scenarios covering static Core RBAC [87, pp. 71].

The *DAC* case covers DAC without delegation of rights. The used characteristic types are the *Identity* of actors as well as the *ReadAccess* and *Owner* of stores. All characteristic types use a set of identities as values. We reuse the *Store* and *Forwarder* behavior descriptions that we described previously. The analysis query in Listing 9 detects data received by actors, which comes from a store that has not granted read access to that actor. It uses the flow tree `S`, Listing 9: DAC analysis query.

```
?- store(STORE), actor(A), inputPin(A,PIN), flowTree(A, PIN, S),
traversedNode(S, STORE), nodeCharacteristic(A, ’Identity’, AID),
\+ nodeCharacteristic(STORE, ’ReadAccess’, AID).
```

Listing 10: Extension of analysis query for non-interference using totally ordered labels.

```
?- actor(A), (actorProcess(P, A); A=P), ...
```

as well as the helper clause `traversedNode` that tests whether the given store `STORE` is in the flow tree `S`. We could successfully represent the system design and the corresponding analysis. The involved characteristic types and behavior descriptions are reusable for other DAC cases.

The *MAC* case covers MAC with the military access control model. We use the characteristic types *Classification* of data and the *Clearance* of nodes. Both characteristic types use an ordered set of security levels. We reuse the previously described behavior descriptions *Store* and *Forwarder*. A *Joiner* propagates the highest classification value of all incoming data items. The analysis query is the same query as already presented in Listing 4 but we restrict the nodes to be checked to nodes directly associated to an actor as shown in Listing 10. We could successfully represent the MAC case, i.e. the system design and the analysis.

In the *ABAC* case, we use the characteristic types *CustomerLocation* and *CustomerStatus* to describe attributes of data as well as *EmployeeLocation* and *EmployeeRole* to describe attributes of actors. We reuse the previously defined behavior descriptions *Store* and *Forwarder*. The *Joiner* applies the union of all incoming data characteristics to the outgoing data. The *LocationChanger* acts like the forwarding behavior but sets the location to *Asia*. The analysis query in Listing 11 encodes the specific requirements of the case. A violation is detected if (i) the location of the actor and the data is not the same and the actor is not a manager or (ii) the data is about a celebrity and the actor is not a manager. We could successfully represent the case, i.e. the system design and the analysis. All behaviors except the location changing behavior are reusable. The analysis query is specific for the ABAC rules and not reusable. However, the flexibility of Prolog allows to represent even complex attribute descriptors and relations.

Listing 11: Analysis query for ABAC case.

```
?- actor(A), inputPin(A, PIN),
nodeCharacteristic(A, ’EmployeeLocation’, SUBJ_LOC),
nodeCharacteristic(A, ’EmployeeRole’, SUBJ_ROLE),
characteristic(A, PIN, ’CustomerLocation’, OBJ_LOC,S),
characteristic(A, PIN, ’CustomerStatus’, OBJ_STAT,S),
(SUBJ_LOC \= OBJ_LOC, SUBJ_ROLE \= ’Manager’;
OBJ_STAT = ’Celebrity’, SUBJ_ROLE \= ’Manager’).
```

As the values of the syntactic quality metric and the corresponding discussion demonstrated, we can represent multiple types of information flow and access control mechanism (Ch2) in system designs. We integrated the confidentiality mechanisms via extensions rather than predefined behavior descriptions or characteristic types. Because further, custom analyses would be integrated via the same extensions, the evaluation also demonstrated that custom analysis definitions (Ch3) can be integrated without invasive source code extensions.

10.4. Evaluation Results and Discussion of Reusability

We calculated the Jaccard Coefficient for the cases covering the *TravelPlanner*, *DistanceTracker* and *ContactSMS* to answer Q2.1. For the *TravelPlanner* system design, the coefficient is $j = {}^{89}/_{219} = 0.41$. For the *DistanceTracker* system design, the coefficient is $j = {}^{47}/_{98} = 0.48$. For the *ContactSMS* system design, the coefficient is $j = {}^{66}/_{123} = 0.54$.

The Jaccard Coefficients that we calculated for the three cases *TravelPlanner*, *DistanceTracker* and *ContactSMS* range between 0.41 and 0.54. The bigger the value is, the more similar the DFDs are. A value of 0.5 means that the shared amount of model elements is as big as the sum of the individual model elements of both involved models. This is a significant improvement compared to a value of 0, which would be the result of using two dedicated modeling languages of the state of the art for representing two versions of a system. An in-depth look at the individual model elements, i.e. the model elements that are different when using different confidentiality mechanisms, confirms that the structural elements, i.e. the nodes and data flows, are not affected by the switch to another confidentiality mechanism. This means, the DFD structure is equal, which is the expected effect of separating the system structure from the confidentiality mechanism in the metamodel. The good metric values show that the chosen modeling approach supports considerable reuse of existing models when switching confidentiality mechanisms.

To give an idea what these results mean, we would like to explain how we switched the mechanism in the case study. Figure 9 presents the Distance Tracker case. The upper part shows the DFD extended by properties and behavior descriptions. The lower part shows the particular properties and behaviors. In order to switch the confidentiality mechanism from information flow to RBAC, we adjusted the properties and behaviors of information flow (shown in dark grey) in a way that they look like the RBAC properties and behaviors shown in light grey. This means, we neither had to adjust the DFD structure nor the annotations of the DFD (shown as upper case letters). It is also possible to first strip all annotated information, i.e. properties and behaviors, from the DFD, import an existing analysis definition and add new annotations to the DFD but this implies additional effort for recreating the annotations. Either way, the DFD structure will always remain the same, which means designers can save effort by not recreating the model from scratch.

As the results and the previous discussions show, the proposed extended DFD is capable of representing access control and information flow control mechanisms. Because we did not have to change the modeling language to

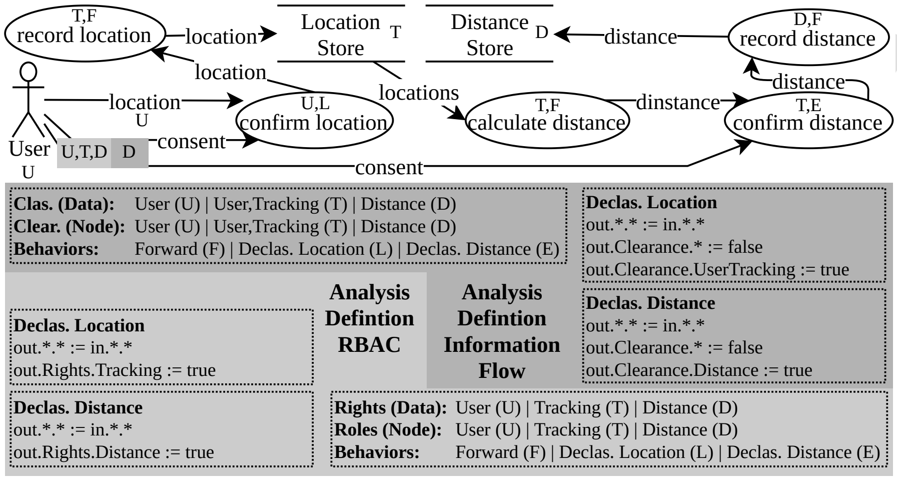

T,F
record location
location
location
location
U
U,L
confirm location
consent
User
U
U,T,D
D
Location
Store
T
Distance
Store
D
locations
T,F
calculate distance
distance
D,F
record distance
distance
dinstance
T,E
confirm distance
consent
Clas. (Data): User (U) | User,Tracking (T) | Distance (D)
Clear. (Node): User (U) | User,Tracking (T) | Distance (D)
Behaviors: Forward (F) | Declas. Location (L) | Declas. Distance (E)
Declas. Location
out.*.* := in.*.*
out.Clearance.* := false
out.Clearance.UserTracking := true
Analysis Defintion RBAC
Analysis Defintion Information Flow
Declas. Distance
out.*.* := in.*.*
out.Clearance.* := false
out.Clearance.Distance := true
Declas. Location
out.*.* := in.*.*
out.Rights.Tracking := true
Declas. Distance
out.*.* := in.*.*
out.Rights.Distance := true
Rights (Data): User (U) | Tracking (T) | Distance (D)
Roles (Node): User (U) | Tracking (T) | Distance (D)
Behaviors: Forward (F) | Declas. Location (L) | Declas. Distance (E)

Figure 9: Differences in Distance Tracker cases for information flow (dark grey) and access control (light grey).

represent both mechanisms, we successfully addressed challenge Ch2. As the Jaccard Coefficient illustrated, we did not only achieve this by merging two distinct modeling languages but by using a commonly shared modeling core (the DFD core elements) and extending it by analysis-specific modeling constructs. We represented all confidentiality mechanisms by extensions, which means that these extensions are the foundation of confidentiality analyses that users can define. Therefore, we addressed the modeling aspect of Ch3.

10.5. Evaluation Results and Discussion of Accuracy

We executed the previously defined analyses for every case that we could express and classified the results. We found violations in 14 cases and all reported violations trace back to the specific issue. This means all results are classified as true positives ($t_p = 14$) and there are no false positives ($f_p = 0$). Because all cases contain an issue and violations have been reported for all cases, there are no false negatives $f_n = 0$. This brings us to a precision of $p = {}^{14}/_{(14+0)} = 1$ and a recall of $r = {}^{14}/_{(14+0)} = 1$. Thus, our analyses achieved perfect accuracy. We could reproduce the analysis results of related publications that initially defined the cases. We represented and analyzed information flow and access control cases, while related approaches can only represent subsets as discussed in Section 4.2.

As part of the result classification, we checked every reported violation. Reporting on every violation as part of this article would require a considerable amount of space and also knowledge about the particular DFD. Therefore, we do not report on the details of this classification in this article but refer to our data set [68], in which we give enough details on the DFD to understand the classification for each violation that is also part of the data set.

The values of the precision and recall metrics demonstrated that we cannot only represent systems and confidentiality mechanisms as well as analyses but that we can also derive accurate results via the defined analyses. We always introduced errors by adding an additional, alternative data flow to a DFD without an issue. If we did not systematically explore all possible data flow paths, we could not have received such accurate results. Therefore, we addresses the challenge about considering multiple data flow paths (Ch1). The results also support our claim to support information flow and access control analyses (Ch2) as well as custom analysis definitions (Ch3) because we cannot only model them (see Section 10.3) but also execute them.

10.6. Threats to Validity

We structure the discussion of threats to validity by the four categories of Runeson and Höst [88] for evaluations based on case studies.

Internal validity assures that no unknown factor influences the investigated factor in order to draw valid causal relations. The investigated factors in this evaluation are the expressiveness, reusability and accuracy of our syntax and semantics. The expected influencing factors are our syntax and semantics. However, further factors can influence the expressiveness: Limited experience with the modeling language can influence the expressiveness negatively. We can exclude this factor because the authors of this article are the designers of the modeling language. Too simple scenarios can make the expressiveness look more positive than it actually is because they omit relevant aspects. We selected all information flow cases and half of the access control cases based on related publications [29, 85, 32], so we do not expect them to be tailored or too simple in this field of research. In addition, we selected all cases from the mentioned, related publications to avoid a tailored or bias selection. We used weighted sums to avoid an increased influence of cases sharing the same analysis definition. Without this, it would be possible to hide a lack of expressiveness regarding one type of analysis definition by adding many cases using a well supported analysis definition. We created three access control cases on our own but included fundamental concepts mentioned in a corresponding textbook [87]. We report on aspects of the particular access control mechanisms that we did not cover in the limitations in Section 10.7 to not claim more expressiveness than the case study could show. Overly simplifying analyses can positively influence the expressiveness by hiding important details. We stick as closely as possible to the analyses presented in related work [29, 85, 32] or the corresponding textbook [87] to mitigate simplification. We report on aspects of the particular information flow control mechanisms that we did not cover in the limitations in Section 10.7. There are also factors that can influence the accuracy: Even if we did not insert issues in initially created DFDs, there still might be issues that lead to a violation. We cannot rule this out but the evaluation showed that we can successfully detect all injected violations and trace them back at least. Therefore, we can only claim that the analyses at least provide results as good as the results of related approaches. Incorrect analysis queries or DFDs can yield always the same result, which might be a detected violation or not.

We addressed this issue by always tracing back violations, which is unlikely to be successful if the analysis query does not properly describe the violation to be expected. Overfitting analysis queries, such as by encoding the violation to be reported directly in the query, can make the accuracy look more positive than it actually is. We evaluated three analysis types (OL, 2L, RBAC according to Table 3) with more than one DFD and achieved accurate results. This is unlikely to succeed for analysis types, which use queries that are overfitted to a particular issue or DFD. In *PrivateTaxi* (LG), the query is system-specific as requested by the original case description. For the remaining access control queries (DAC, MAC, RBAC), we discussed their generalizability, which would also reveal overfitted queries.

External validity assures that researchers only generalize findings if it is valid to do so. According to Runeson and Höst [88], case study research does not focus on representativeness but on specific aspects of the case under study to get a better understanding of the phenomena. Therefore, insights cannot be generalized to arbitrary other cases unreservedly. However, generalizing insights to cases with comparable characteristics is possible. Therefore, we discussed the characteristics of the case and how it can be generalized for each analysis type in the discussion of expressiveness in Section 10.3. We consider the cases derived from related work representative for the application area. In addition, we evaluated 15 cases, which we consider a reasonable amount, especially when comparing the amount to related work [29, 85, 32], which usually only considers 5 cases with similar analysis definitions at most. The remaining cases at least comply with common definitions.

Construct validity assures that the used metrics are capable of answering the evaluation question. We chose the syntactical quality metric to rate expressiveness. It is barely possible to summarize expressiveness by metrics because variations and limitations of the studied cases have to be discussed, so we extensively discussed the results and provided the metric values for the sake of a quick overview. Syntactic quality is an appropriate metric for this as it has already been used to rate the expressiveness of a DSL [79]. We use the Jaccard Coefficient to reason about reusability when switching confidentiality mechanisms. The Jaccard Coefficient is an established metric for rating similarity of sets in various fields [80]. The coefficient requires a definition of an element and a definition of equality between two elements in order to rate similarity. We defined both in the evaluation design in Section 10.2. The definitions cover models elements and their properties. Because the whole model only consists of model elements and properties, the definition covers the whole model. Therefore, the coefficient is applicable to rate the similarity of our models. In addition, we demonstrated the applicability of the Jaccard Coefficient for comparing models in previous work [82, 83]. Using the comparison approach of EMFCompare [81] to determine equal model elements is reasonable as we explained in the evaluation design. We described the steps that EMFCompare takes in the evaluation design in Section 10.2. The steps are intuitive, established and could also be carried out manually. The precision and recall metrics used to rate the accuracy of the analyses are commonly applied metrics for rating the accuracy of various information flow analyses [25, 89]. The selection of cases is appropriate for answering the evaluation questions as discussed before.

Reliability assures that the conducted study, i.e. the data collection and data analysis, does not depend on the particular researcher but other researchers come to the same results. As discussed before, the model quality depends on the experience of the modeler with the syntax and semantics. We cannot completely mitigate this issue. However, we provide all material required to replicate the evaluation starting from the models as stated in Section 10.8. Additionally, all metric values can be calculated in an objective way: we provide clear instructions on how to collect input data for calculating the metrics without the need for subjective interpretations. Therefore, the process and results are traceable and other researchers can decide whether the study has been carried out correctly.

10.7. Limitations

We distinguish between limitations of the proposed syntax and semantics on the one hand, as well as limitations of the evaluation on the other hand.

One limitation of the syntax and semantics has been demonstrated in the evaluation: there are no means to represent individual data or users but only classes of data or users. A class of data describes a group of data that is treated the same. A class of users describes a group of users acting the same. This limitation implies limited support for some specific aspects of confidentiality mechanisms: The RBAC extension providing means to specify constraints on individual subjects, which hold roles, cannot be represented. Therefore, we cannot represent that two clerks have to approve something but they must not be the same person, for instance. The delegation of rights in DAC cannot be represented, so we cannot distinguish between valid and invalid access to data that involves delegated access rights, for instance. Also, it is not possible to ensure non-interference between users of the same type, so we cannot ensure, for example, that a bank customer cannot access the balance of another customer. All of these aspects would require detailed information about individual users and data as well as a mechanism to express time and dependencies between system states in different times. We intentionally excluded this because such detailed information required to model individuals might not be available during design time. Additionally, the amount of elements to specify will certainly be increased when more detailed models and even considering time are necessary. We demonstrated that the proposed syntax and semantics can provide valuable results and insights and suggest to cover the remaining aspects in later development phases when more detailed information or even source code is available. This lowers the overhead for analyzing these aspects significantly. Other approaches building on DFDs such as SecDFD [29] or FlowUML [61] share the same restrictions.

Our evaluation focused on the expressiveness and accuracy of our syntax and semantics as well as on reusability. We did not evaluate usability. As already said before, we intentionally did not evaluate usability because this would only evaluate our implementation of tool support rather than our concepts. We do not see an open research question in whether usable tooling for modeling and analyzing DFDs can be created because the users in a recent study of Tuma et al. [17] could successfully use their DFD modeling and analysis approach. We also did not verify the correctness of the mapping and the resulting logic program. As already motivated in the evaluation goals, verifying correctness does not allow us to answer whether we sufficiently addressed the challenges mentioned in the introduction. However, we plan to report on the correctness in future publications on different aspects of our approach.

10.8. Data Availability

We provide all data used in the evaluation in our data set [68]. This includes metamodels, source code, data flow diagrams, logic programs, analysis queries and results. In addition, we provide a manual to replicate all steps of our evaluation as part of the dataset.

11. Conclusions

In this article, we proposed an extended DFD syntax and analysis semantics that allow expressing analyses to detect violations of access control and information flow requirements with good accuracy. The DFD syntax is based on DFD elements as introduced by DeMarco [33] but extends these elements with means for representing behavior relevant for confidentiality analyses. The semantics describe this behavior in terms of label propagation rules formulated in a logic program. An automated mapping translates the extended DFD into an executable logic program that yields detected violations. Thereby, we address three open challenges of software design modeling and analysis approaches aiming to find confidentiality violations. In our evaluation, we demonstrated the expressiveness with respect to information flow and access control, demonstrated effective reuse of existing models when switching between information flow and access control as well as evaluated the accuracy in a case study considering fifteen cases.

Practitioners as well as researchers can benefit from our contributions. Our syntax and semantics provide means for systematically considering confidentiality properties in an early design stage. This allows identifying fundamental design issues early and fixing them in a cost-efficient way. Because our syntax is close to the commonly known concepts and syntax of DFDs, we assume a flat learning curve for designers. Researchers can use our provided analyses as a foundation for defining their own confidentiality analyses based on data property propagation. This allows focusing on the application area and analysis concepts rather than on generic issues like data propagation or data dependency resolution. Additionally, the cases published as part of this article [68] can serve as a benchmark for existing analyses.

We see five major points as part of future work. First, we plan to investigate how our DFD-based modeling language and analyses can be integrated with existing early design modeling and analysis approaches. Many modeling languages have counterparts for the modeling elements we presented in this paper.

It might be possible to cover all aspects of our extended DFD modeling language by some lightweight modifications and a mapping to our modeling language. We already created a preliminary concept for Architectural Description Language (ADL) integration [90] that needs to be refined and evaluated in future. Second, we plan to investigate whether the presented syntax and semantics are capable of supporting further security objectives such as integrity. Evaluating the support for integrity is reasonable because information flow requirements often ensure confidentiality and integrity. Third, we plan to investigate how we can build catalogues of reusable model elements. Reusing model elements has the potential to lower the modeling effort. An important question to answer is how designers could use these catalogues and how to design analysis definitions as reusable as possible. Fourth, we would like to know whether parts of our analyses could be executed in real-time while modeling to guide designers and provide fast feedback. Challenges in doing so include the handling of incomplete models, incomplete analysis results and how to identify and present useful analysis results while editing. Fifth, we plan a publication on the verification of correctness with respect to the mapping and the logic program. We plan to do the verification of the mapping based on the properties to verify for correctness collected by Rahim and Whittle [91]. The verification of the correctness of the logic program will consider completeness and correctness as suggested by Drabent [92].

Acknowledgements

This work was supported by the German Research Foundation (DFG) under project number 432576552, HE8596/1-1 (FluidTrust), as well as by funding from the topic 46.23.03 Engineering Security for Mobility Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs.

References

[1] Institute of Electrical and Electronics Engineers, IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description for Software-Intensive Systems, Standard, IEEE (Oct. 2000).

[2] J. Deogirikar, A. Vidhate, Security attacks in IoT: A survey, in: 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2017, pp. 32–37. doi:10.1109/I-SMAC.2017.8058363.

[3] A. Sadeghi, C. Wachsmann, M. Waidner, Security and privacy challenges in industrial Internet of Things, in: 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2015, pp. 1–6. doi:10.1145/2744769.2747942.

[4] International Organization for Standardization, ISO/IEC 27000:2018(E) Information technology – Security techniques – Information security management systems – Overview and vocabulary, Standard, ISO, Geneva, CH (Feb. 2018).

[5] R. Alguliyev, Y. Imamverdiyev, L. Sukhostat, Cyber-physical systems and their security issues, Computers in Industry 100 (2018) 212–223. doi:10.1016/j.compind.2018.04.017.

[6] European Union, Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), Official Journal of the European Union 59 (2016) 1–88.
URL https://eur-lex.europa.eu/eli/reg/2016/679/oj

[7] E. Denham, Penatly Notice, Penatly Notice COM0783542, Information Commissioner's Office, United Kingdom, accessed 2021-08-02 (Oct. 2020).
URL https://web.archive.org/web/20210620130131/https://edpb.europa.eu/sites/default/files/article-60-final-decisions/uk_2010-10_data_breach_security_of_processing_decisionpublic_final.pdf

[8] E. Denham, Penatly Notice, Penatly Notice COM0804337, Information Commissioner's Office, United Kingdom, accessed 2021-08-02 (Oct. 2020).
URL https://web.archive.org/web/20210802034347/https://edpb.europa.eu/sites/default/files/article-60-final-decisions/uk_2020-10_personal_data_breach_decisionpublic_final.pdf

[9] H. Weisbaum, Trust in Facebook has dropped by 66 percent since the Cambridge Analytica scandal, accessed 2021-08-20 (Apr. 2018).
URL https://web.archive.org/web/20210820004535/https://www.nbcnews.com/business/consumer/trust-facebook-has-dropped-51-percent-cambridge-analytica-scandal-n867011

[10] J. Isaak, M. J. Hanna, User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection, Computer 51 (8) (2018) 56–59. doi:10.1109/MC.2018.3191268.

[11] Microsoft Corporation, Microsoft Security Development Lifecycle (SDL) (2020).
URL https://web.archive.org/web/20210924183639/https://www.microsoft.com/en-us/securityengineering/sdl/

[12] B. W. Boehm, R. K. Mcclean, D. E. Urfrig, Some experience with automated aids to the design of large-scale reliable software, IEEE Transactions on Software Engineering SE-1 (1) (1975) 125–133. doi:10.1109/TSE.1975.6312826.

[13] Microsoft Corporation, iSEC Partners, Inc., Microsoft SDL: Return-on-Investment, accessed 2021-09-25 (2009).
URL https://web.archive.org/web/20210925085942/https://www.nccgroup.com/globalassets/our-research/us/whitepapers/isec-partners---microsoft-sdl-return-on-investment.pdf

[14] K. S. Hoo, A. W. Sudbury, A. R. Jaquith, Tangible ROI through Secure Software Engineering, Secure Business Quarterly 1 (2) (2001) 1–3, accessed 2020-09-22.
URL https://web.archive.org/web/20060614122530/http://www.sbq.com/sbq/rosi/sbq_rosi_software_engineering.pdf

[15] G. McGraw, Software Security - Building Security In, Addison-Wesley Professional, 2006.

[16] J. Jürjens, Sound methods and effective tools for model-based security engineering with UML, in: Proceedings of International Conference on Software Engineering, ICSE'05, 2005, pp. 322–331. doi:10.1109/ICSE.2005.1553575.

[17] K. Tuma, L. Sion, R. Scandariato, K. Yskout, Automating the early detection of security design flaws, in: E. Syriani, H. A. Sahraoui, J. d. Lara, S. Abrahão (Eds.), Proceedings of ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS'20, ACM, 2020, pp. 332–342. doi:10.1145/3365438.3410954.

[18] A. Shostack, Threat modeling: designing for security, Wiley, Indianapolis, IN, 2014.

[19] R. Al-Ali, P. Hnetynka, J. Havlik, V. Krivka, R. Heinrich, S. Seifermann, M. Walter, A. Juan-Verdejo, Dynamic security rules for legacy systems, in: Proceedings of European Conference on Software Architecture - Volume 2, ECSA '19, ACM, 2019, pp. 277–284. doi:10.1145/3344948.3344974.

[20] K. Pohl, C. Rupp, Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam, foundation level, IREB compliant, 2nd Edition, Rocky Nook, Santa Barbara, CA, 2015.

[21] J. Dick, E. Hull, K. Jackson, Requirements Engineering, Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-61073-3.

[22] K. E. Wiegers, More About Software Requirements: Thorny Issues and Practical Advice, Microsoft Press, USA, 2005.

[23] A. Sabelfeld, A. Myers, Language-based information-flow security, IEEE Journal on Selected Areas in Communications 21 (1) (2003) 5–19. doi:10.1109/JSAC.2002.806121.

[24] D. Hedin, A. Sjösten, F. Piessens, A. Sabelfeld, A Principled Approach to Tracking Information Flow in the Presence of Libraries, in: M. Maffei, M. Ryan (Eds.), Principles of Security and Trust, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2017, pp. 49–70. doi:10.1007/978-3-662-54455-6_3.

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, ACM, 2014, pp. 259–269. doi:10.1145/2594291.2594299.

[26] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, A. Sabelfeld, An Empirical Study of Information Flows in Real-World JavaScript, in: Proceedings of ACM SIGSAC Workshop on Programming Languages and Analysis for Security, PLAS'19, ACM, 2019, pp. 45–59. doi:10.1145/3338504.3357339.

[27] W. Xu, S. Bhatkar, R. Sekar, Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks, in: A. D. Keromytis (Ed.), Proceedings of USENIX Security Symposium, USENIX Association, 2006, pp. 121–136.
URL https://web.archive.org/web/20210822002152/http://usenix.org/events/sec06/tech/full_papers/xu/xu.pdf

[28] L. Wang, C. Fang, B. Mao, L. Xie, TMAC: Taint-Based Memory Protection via Access Control, in: Proceedings of International Conference on Dependability, DEPEND'09, 2009, pp. 19–27. doi:10.1109/DEPEND.2009.33.

[29] K. Tuma, R. Scandariato, M. Balliu, Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis, in: Proceedings of IEEE International Conference on Software Architecture, ICSA'19, IEEE, 2019, pp. 191–200. doi:10.1109/ICSA.2019.00028.

[30] A. van den Berghe, K. Yskout, R. Scandariato, W. Joosen, A Lingua Franca for Security by Design, in: Proceedings of IEEE Cybersecurity Development, SecDev'18, IEEE, Cambridge, MA, 2018, pp. 69–76. doi:10.1109/SecDev.2018.00017.

[31] K. Alghathbar, D. Wijesekera, authUML: a three-phased framework to analyze access control specifications in use cases, in: Proceedings of ACM Workshop on Formal Methods in Security Engineering, FMSE '03, ACM, 2003, pp. 77–86. doi:10.1145/1035429.1035438.

[32] S. Seifermann, R. Heinrich, R. Reussner, Data-Driven Software Architecture for Analyzing Confidentiality, in: Proceedings of International Conference on Software Architecture, ICSA, IEEE, 2019, pp. 1–10. doi:10.1109/ICSA.2019.00009.

[33] T. DeMarco, Structured analysis and system specification, Prentice-Hall, Englewood Cliffs, N.J., 1979.

[34] W. Torres, M. G. J. van den Brand, A. Serebrenik, A systematic literature review of cross-domain model consistency checking by model management tools, Software and Systems Modelingdoi:10.1007/s10270-020-00834-1.

[35] K. Katkalov, K. Stenzel, M. Borek, W. Reif, Model-Driven Development of Information Flow-Secure Systems with IFlow, in: Proceedings of International Conference on Social Computing, SocialCom'13, 2013, pp. 51–56. `doi:10.1109/SocialCom.2013.14`.

[36] R. France, Semantically extended dataflow diagrams: a formal specification tool, IEEE Transactions on Software Engineering 18 (4) (1992) 329–346. `doi:10.1109/32.129221`.

[37] C. Petersohn, W.-P. de Roever, C. Huizing, J. Peleska, Formal Semantics for Ward & Mellor's Transformation Schemas, in: C. J. van Rijsbergen, D. Till (Eds.), Proceedings of Refinement Workshop, Springer, London, 1994, pp. 14–41. `doi:10.1007/978-1-4471-3240-0_2`.

[38] D. Fensel, J. Angele, D. Landes, R. Studer, Giving Structured Analysis Techniques a Formal and Operational Semantics with KARL, in: H. Züllighoven, W. Altmann, E.-E. Doberkat (Eds.), Requirements Engineering '93: Prototyping, Vieweg+Teubner Verlag, 1993, pp. 267–285. `doi:10.1007/978-3-322-94703-1_18`.

[39] P. G. Larsen, N. Plat, H. Toetenel, A Formal Semantics of Data Flow Diagrams, Formal Aspects of Computing 6 (6) (1994) 586–606. `doi:10.1007/BF03259387`.

[40] H. Xiong, H. Zhang, X. Dong, L. Meng, W. Zhao, DFDVis: A Visual Analytics System for Understanding the Semantics of Data Flow Diagram, in: B. Zou, M. Li, H. Wang, X. Song, W. Xie, Z. Lu (Eds.), Proceedings of International Conference of Pioneering Computer Scientists, Engineers and Educators, ICPCSEE'17, Springer, 2017, pp. 660–673. `doi:10.1007/978-981-10-6385-5_55`.

[41] T. Liu, C. S. Tang, Semantic specification and verification of data flow diagrams, Journal of Computer Science and Technology 6 (1) (1991) 21–31. `doi:10.1007/BF02943404`.

[42] N. Plat, J. Katwijk, K. Pronk, A case for structured analysis/formal design, in: G. Goos, J. Hartmanis, S. Prehn, W. J. Toetenel (Eds.), VDM'91 Formal Software Development Methods, Vol. 551, Springer, 1991, pp. 81–105. `doi:10.1007/3-540-54834-3_8`.

[43] T. Wahls, A. L. Baker, G. T. Leavens, An Executable Semantics for a Formalized Data Flow Diagram Specification Language, Technical Report TR93-27, Iowa State University (1993).
URL `https://web.archive.org/web/20200601123325/https://lib.dr.iastate.edu/cs_techreports/160/`

[44] G. T. Leavens, T. Wahls, A. L. Baker, K. Lyle, An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams, Technical Report TR93-28c, Iowa State University (1996).

URL https://web.archive.org/web/20200725235500/https://lib.dr.iastate.edu/cs_techreports/101/

[45] G. T. Leavens, T. Wahls, A. L. Baker, Formal semantics for SA style data flow diagram specification languages, in: Proceedings of ACM Symposium on Applied Computing, SAC'99, ACM, 1999, pp. 526–532. doi:10.1145/298151.298433.

[46] Kavi, Buckles, Bhat, A Formal Definition of Data Flow Graph Models, IEEE Transactions on Computers C-35 (11) (1986) 940–948. doi:10.1109/TC.1986.1676696.

[47] P. Brunza, T. van der Weide, The Semantics of Data Flow Diagrams, in: N. Prakash (Ed.), Proceedings of International Conference on Management of Data, CISMOD'89, McGraw-Hill, Hyderabad, India, 1989, pp. 66–78. URL https://web.archive.org/web/20181008133048/http://cs.ru.nl/Th.P.vanderWeide/docs/1989-Bruza-DataFlowSem.pdf

[48] C. Gerking, D. Schubert, E. Bodden, Model Checking the Information Flow Security of Real-Time Systems, in: M. Payer, A. Rashid, J. M. Such (Eds.), Proceedings of International Symposium on Engineering Secure Software and Systems, ESSoS'18, Springer, 2018, pp. 27–43. doi:10.1007/978-3-319-94496-8_3.

[49] J. Jürjens, Secure Systems Development with UML, Springer-Verlag, Berlin Heidelberg, 2005. doi:10.1007/b137706.

[50] B. Hoisl, S. Sobernig, M. Strembeck, Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach, Software & Systems Modeling 13 (2) (2014) 513–548. doi:10.1007/s10270-012-0263-y.

[51] M. Almorsy, J. Grundy, A. S. Ibrahim, Automated software architecture security risk analysis using formalized signatures, in: Proceedings of International Conference on Software Engineering, ICSE'13, IEEE, San Francisco, CA, USA, 2013, pp. 662–671. doi:10.1109/ICSE.2013.6606612.

[52] T. Abdellatif, L. Sfaxi, R. Robbana, Y. Lakhnech, Automating information flow control in component-based distributed systems, in: Proceedings of International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE'11, ACM Press, 2011, pp. 73–82. doi:10.1145/2000229.2000241.

[53] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, D. Wasserrab, Checking probabilistic noninterference using JOANA, it-Information Technology 56 (6) (2014) 280–287. doi:10.1515/itit-2014-1051.

[54] T. Runge, A. Knüppel, T. Thüm, I. Schaefer, Lattice-Based Information Flow Control-by-Construction for Security-by-Design, in: Proceedings of International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE'20, ACM, 2020, pp. 44–54. doi:10.1145/3372020.3391565.

[55] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, M. Ulbrich (Eds.), Deductive Software Verification – The KeY Book, Springer International Publishing, 2016. doi:10.1007/978-3-319-49812-6.

[56] M. Abi-Antoun, D. Wang, P. Torr, Checking threat modeling data flow diagrams for implementation conformance and security, in: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, 2007, pp. 393–396. doi:10.1145/1321631.1321692.

[57] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, W. Joosen, A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements, Requirements Engineering 16 (1) (2011) 3–32. doi:10.1007/s00766-010-0115-7.

[58] M. Yampolskiy, P. Horvath, X. D. Koutsoukos, Y. Xue, J. Sztipanovits, Systematic analysis of cyber-attacks on CPS-evaluating applicability of DFD-based approach, in: Proceedings of International Symposium on Resilient Control Systems, ISRCS'12, 2012, pp. 55–62. doi:10.1109/ISRCS.2012.6309293.

[59] B. J. Berger, K. Sohr, R. Koschke, Automatically Extracting Threats from Extended Data Flow Diagrams, in: J. Caballero, E. Bodden, E. Athanasopoulos (Eds.), Proceedings of International Symposium on Engineering Secure Software and Systems, ESSoS'16, Springer International Publishing, 2016, pp. 56–71. doi:10.1007/978-3-319-30806-7_4.

[60] L. Sion, K. Yskout, D. Van Landuyt, W. Joosen, Solution-aware data flow diagrams for security threat modeling, in: Proceedings of the ACM Symposium on Applied Computing, SAC '18, ACM Press, 2018, pp. 1425–1432. doi:10.1145/3167132.3167285.

[61] K. Alghathbar, C. Farkas, D. Wijesekera, Securing UML Information Flow using FlowUML, Journal of Research and Practice in Information Technology 38 (1) (2006) 111–120.
URL https://50years.acs.org.au/content/dam/acs/50-years/journals/jrpit/JRPIT38.1.111.pdf

[62] A. van den Berghe, K. Yskout, R. Scandariato, W. Joosen, A Model for Provably Secure Software Design, in: Proceedings of IEEE/ACM International FME Workshop on Formal Methods in Software Engineering, FormaliSE'17, IEEE, 2017, pp. 3–9. doi:10.1109/FormaliSE.2017.6.

[63] R. Heinrich, M. Strittmatter, R. Reussner, A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis, IEEE

Transactions on Software Engineering 47 (4) (2021) 775–800. doi:10.1109/TSE.2019.2903797.

[64] Object Management Group (OMG), Unified Modeling Language 2.5.1, Specification formal/17-12-05, accessed 2021-09-21 (Dec. 2017).
URL https://web.archive.org/web/20210225061032/https://www.omg.org/spec/UML/2.5.1/PDF

[65] M. Bramer, Logic programming with Prolog, 2nd Edition, Springer, London, 2013. doi:10.1007/978-1-4471-5487-7.

[66] M. Kifer, Y. A. Liu (Eds.), Declarative Logic Programming: Theory, Systems, and Applications, Vol. 20, ACM and Morgan & Claypool, 2018. doi:10.1145/3191315.

[67] P. M. Nugues, An Introduction to Prolog, in: An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German, Cognitive Technologies, Springer, Berlin, Heidelberg, 2006, pp. 433–486. doi:10.1007/3-540-34336-9_16.

[68] S. Seifermann, R. Heinrich, D. Werle, R. Reussner, Data Set of Publication on Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams (Sep. 2021). doi:10.5281/zenodo.5535598.

[69] J. Kunz, Efficient Data Flow Constraint Analysis, Master's thesis, Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany (2018). doi:10.5445/IR/1000122485.

[70] R. Böhme, R. Reussner, Validation of Predictions with Measurements, in: I. Eusgeld, F. C. Freiling, R. Reussner (Eds.), Dependability Metrics, Vol. 4909, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 14–18. doi:10.1007/978-3-540-68947-8_3.

[71] D. Steinberg (Ed.), EMF: Eclipse Modeling Framework, 2nd Edition, The eclipse series, Addison-Wesley, 2009.

[72] L. Bettini, Implementing Domain Specific Languages with Xtext and Xtend - Second Edition, 2nd Edition, Packt Publishing, 2016.
URL https://web.archive.org/web/20210127045423/https://www.packtpub.com/product/implementing-domain-specific-languages-with-xtext-and-xtend-second-edition/9781786464965

[73] J. Wielemaker, SWI Prolog Reference Manual 7.6.0, Tech. rep., VU University Amsterdam, accessed 2017-11-16 (2017).
URL https://web.archive.org/web/20171116090534/https://www.swi-prolog.org/download/stable/doc/SWI-Prolog-7.6.0.pdf

[74] C. A. González, J. Cabot, Formal verification of static software models in MDE: A systematic review, Information and Software Technology 56 (8) (2014) 821–838. doi:10.1016/j.infsof.2014.03.003.

[75] S. Hahner, S. Seifermann, R. Heinrich, M. Walter, T. Bures, P. Hnetynka, Modeling Data Flow Constraints for Design-Time Confidentiality Analyses, in: Proceedings of International Conference on Software Architecture Companion, ICSA-C'21, IEEE, 2021, pp. 15–21. doi:10.1109/ICSA-C52384.2021.00009.

[76] V. R. Basili, D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering SE-10 (6) (1984) 728–738. doi:10.1109/TSE.1984.5010301.

[77] V. R. Basili, G. Caldiera, H. D. Rombach, The Goal Question Metric Approach, in: Encyclopedia of Software Engineering - 2 Volume Set, John Wiley & Sons, 1994, pp. 528–532.

[78] S. Boyd, D. Zowghi, A. Farroukh, Measuring the expressiveness of a constrained natural language: an empirical study, in: Proceedings of IEEE International Conference on Requirements Engineering, RE'05, 2005, pp. 339–349, iSSN: 2332-6441. doi:10.1109/RE.2005.39.

[79] J. Munnelly, S. Clarke, A Domain-Specific Language for Ubiquitous Healthcare, in: Proceedings of International Conference on Pervasive Computing and Applications, Vol. 2 of ICPCA'08, 2008, pp. 757–762. doi:10.1109/ICPCA.2008.4783710.

[80] M. Levandowsky, D. Winter, Distance between Sets, Nature 234 (5323) (1971) 34–35. doi:10.1038/234034a0.

[81] C. Brun, A. Pierantonio, Model Differences in the Eclipse Modeling Framework, UPGRADE: The European Journal for the Informatics Professional 9 (2) (2008) 29–34.
URL https://web.archive.org/web/20090205174152/http://upgrade-cepis.org/issues/2008/2/upg9-2Brun.pdf

[82] R. Heinrich, Architectural runtime models for integrating runtime observations and component-based models, Journal of Systems and Software 169 (2020) 110722. doi:10.1016/j.jss.2020.110722.

[83] D. Monschein, M. Mazkatli, R. Heinrich, A. Koziolek, Enabling consistency between software artefacts for software adaption and evolution, in: Proceedings of IEEE International Conference on Software Architecture, ICSA'21, IEEE, 2021, pp. 1–12. doi:10.1109/ICSA51549.2021.00009.

[84] A. van den Berghe, R. Scandariato, K. Yskout, W. Joosen, Design notations for secure software: a systematic literature review, Software & Systems Modeling 16 (3) (2017) 809–831. doi:10.1007/s10270-015-0486-9.

[85] K. Katkalov, Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme, PhD Thesis, University of Augsburg, Augsburg, german (2017).
URL https://web.archive.org/web/20210926154149/https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/4339

[86] A. Sabelfeld, D. Sands, Declassification: Dimensions and principles, Journal of Computer Security 17 (5) (2009) 517–548. doi:10.3233/JCS-2009-0352.

[87] S. Furnell (Ed.), Securing information and communications systems: principles, technologies, and applications, Artech House computer security series, Artech House, Boston, 2008.

[88] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164. doi:10.1007/s10664-008-9102-8.

[89] F. Wei, S. Roy, X. Ou, Robby, Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps, in: Proceedings of ACM SIGSAC Conference on Computer and Communications Security, CCS '14, ACM, 2014, pp. 1329–1341. doi:10.1145/2660267.2660357.

[90] S. Seifermann, R. Heinrich, D. Werle, R. Reussner, A Unified Model to Detect Information Flow and Access Control Violations in Software Architectures, in: Proceedings of International Conference on Security and Cryptography, SECRYPT'21, SCITEPRESS, 2021, pp. 26–37. doi:10.5220/0010515300260037.

[91] L. A. Rahim, J. Whittle, A survey of approaches for verifying model transformations, Software & Systems Modeling 14 (2) (2015) 1003–1028. doi:10.1007/s10270-013-0358-0.

[92] W. Drabent, Correctness and Completeness of Logic Programs, ACM Transactions on Computational Logic 17 (3) (2016) 18:1–18:32. doi:10.1145/2898434.

Appendix A. Data Flow Diagrams of Selected Evaluation Cases

We used existing cases as well as self-defined cases in the evaluation. This appendix contains the DFDs of the cases defined by ourselves. Figure A.10 illustrates the image sharing system used in the DAC evaluation. Figure A.11 illustrates the flight monitoring system used in the MAC evaluation. Figure A.12 illustrates the banking system used in the ABAC evaluation.

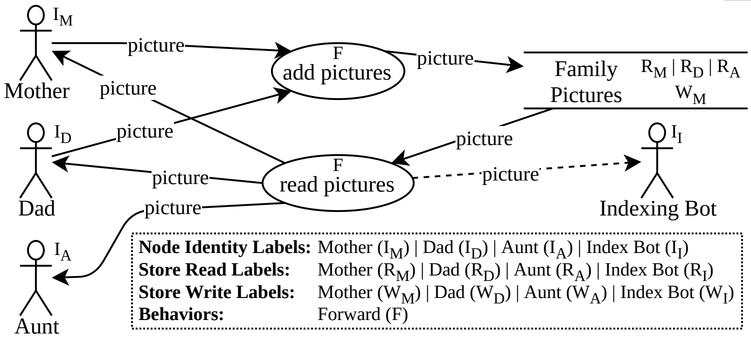

I_M
Mother
picture
picture
F
add pictures
picture
Family Pictures
R_M | R_D | R_A
W_M
I_D
Dad
picture
picture
picture
F
read pictures
picture
I_I
Indexing Bot
I_A
Aunt
picture
Node Identity Labels: Mother (I_M) | Dad (I_D) | Aunt (I_A) | Index Bot (I_I)
Store Read Labels: Mother (R_M) | Dad (R_D) | Aunt (R_A) | Index Bot (R_I)
Store Write Labels: Mother (W_M) | Dad (W_D) | Aunt (W_A) | Index Bot (W_I)
Behaviors: Forward (F)

Figure A.10: Image sharing system used in the DAC evaluation (dashed data flow introduces a violation).

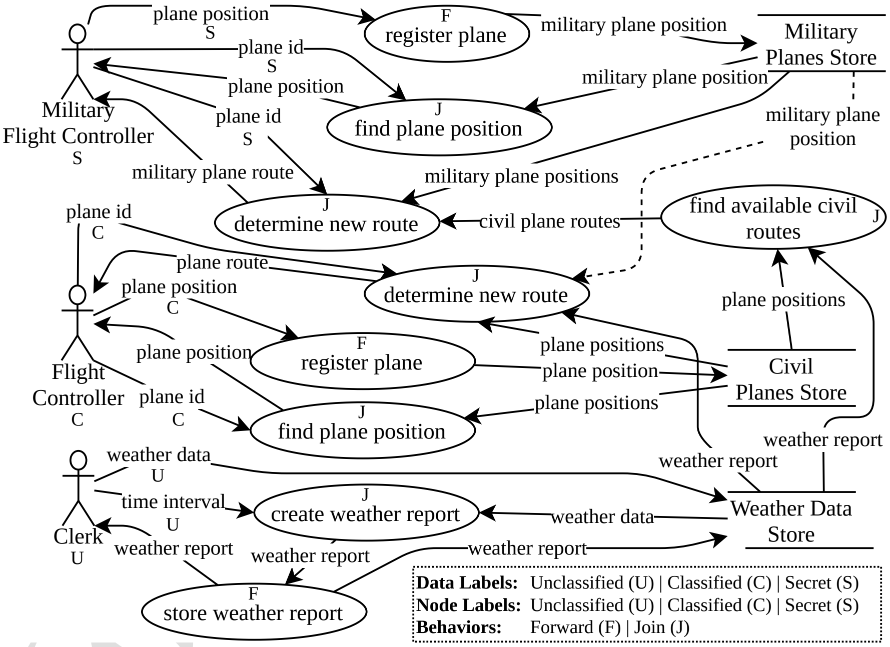


Figure A.11: Flight monitoring system used in the MAC evaluation (dashed data flow introduces a violation).

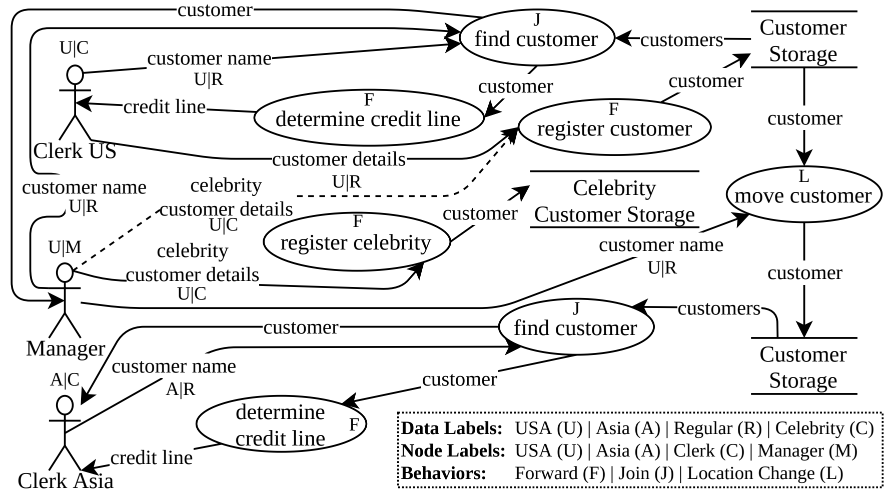


Figure A.12: Banking system used in the ABAC evaluation (dashed data flow introduces a violation).

Author Biographies

Stephan Seifermann

Stephan Seifermann is a researcher at the Karlsruhe Institute of Technology (KIT) since 2018. After graduating with M.Sc. in Informatics from KIT in 2015, he worked as a researcher at the Research Center of Information Technology (FZI) in research and consulting projects. Stephan's research is mainly concerned with architectural confidentiality analyses, model-driven software engineering and architectural quality predictions. In particular, he researches how data-oriented modeling can be used to predict confidentiality violations in early software design phases.

Robert Heinrich

Robert Heinrich is head of the Quality-driven System Evolution research group at Karlsruhe Institute of Technology (KIT). He holds a doctoral degree from Heidelberg University and a degree in Computer Science from University of Applied Sciences Kaiserslautern. His research interests include modularization and composition of model-based analysis for several quality properties, such as performance, confidentiality and maintainability, as well as for several domains, such as information systems, business processes and automated production systems. He is involved in the organization committees of several international conferences, established and organized various workshops, is reviewer for international premium journals.

Dominik Werle

Dominik Werle is a researcher at Karlsruhe Institute of Technology (KIT) since 2016. His research interests include model-based quality prediction for software systems with a focus on the performance of data-intensive applications. Dominik's research is also concerned with using data-flow models to better understand design decisions and architectures for these applications.

Ralf Reussner

Ralf Reussner is a computer science professor at Karlsruhe Institute of Technology (KIT). He holds the chair for Software Design and Quality (SDQ) since 2006 and heads the Institute for Program Structures and Data Organization. His research group works in the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: