# A Model-Driven Approach to Noninterference*

Kurt Stenzel†, Kuzman Katkalov, Marian Borek, and Wolfgang Reif
*Institute for Software & Systems Engineering*
*University of Augsburg, Germany*

**Abstract**

Systems consisting of mobile apps and web services continue to grow in popularity. Guaranteeing that private or sensitive data is treated confidentially in such systems is non-trivial and poses several challenges due to their distributed and platform-specific nature. Information flow control is a formal technique that is used to guarantee the privacy of such data, but is difficult to utilize in practice. We present a model-driven approach which allows to develop such systems with secure information flow using intuitive modeling guidelines. From an abstract system model, partial Java code as well as a formal model is generated automatically and used to verify information flow properties. This paper explains the automatic generation of the formal model and presents several advantages of a model-driven approach for the practical application of information flow control.

**keywords:** noninterference, model-driven development, information flow control, formal methods

## 1 Introduction

As mobile devices are becoming more ubiquitous, mobile apps, designed to solve any task imaginable, are continuing to grow in both number and complexity. Application systems spanning across numerous mobile apps and web services are common, and such systems often need to manage or access their users' private data in order to implement the promised functionality. However, keeping sensitive data confidential w.r.t. information leaks in a complex, distributed system is a difficult task and guaranteeing that such system is secure is not easy. Reports about mobile applications leaking their users' data to third parties due to vulnerabilities or poor system design [1] keep appearing regularly.

Current security mechanisms fail in most cases to address the issue of information flow (IF) leakage. Popular mobile platforms like iOS, Windows Phone and Android implement access control, application sandboxing and permission systems to protect their users' data. However, those are not nearly fine-grained or flexible enough to guarantee anything but the most basic IF properties. Access control and application sandboxing prevent apps from freely accessing each other's memory, but once those apps start interacting using proper interfaces, no further statement can be made about the confidentiality of the exchanged data. A developer can voluntarily restrict her app's access to a predefined set of data sources and sinks like the phone's GPS sensor or the Internet using the platform's permission system. The user is then presented with a list of requested permissions such as "allow Internet access" which she has to agree to prior to installing the app. She is unable to deny certain permissions or further restrict them, and, most importantly, she is unable to tell which information is going to flow to or from those data locations. Thus, an application requesting both the permission to access GPS data and the Internet might only use the Internet connection to display ads, or to leak the user's current position to a third party without her

*Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, volume: 5, number: 3, pp. 30-43

*This work is part of the IFlow project and sponsored by the Priority Programme 1496 "Reliably Secure Software Systems - RS$^3$" of the Deutsche Forschungsgemeinschaft (DFG).

†Corresponding author: Institute for Software- and Systems Engineering Universitaetsstr. 6a, 86159 Augsburg, Germany, Tel: +49-8215982123, Email: stenzel@informatik.uni-augsburg.de, Web: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/staff/stenzel/`

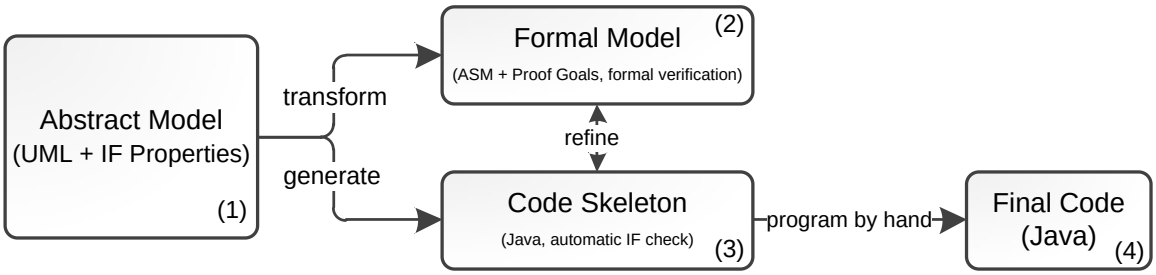


Figure 1: The IFlow approach

knowledge [2]. Furthermore, neither of those security mechanisms is able to prevent the collaboration of several apps with different permissions to exfiltrate sensitive information from the user's phone [3, 4]; e.g., an app that only accesses GPS sensor data could forward it to an app with Internet access and thus leak it to an external web service.

We develop an approach called IFlow which enables the model-driven development of real world systems consisting of several mobile apps and web services with provable information flow properties. The approach describes modeling guidelines for creating an application model in UML, which serves as a basis for the automatic transformation to a formal model as well as distributed, platform-specific Java code for Android devices and Java web services. The overall approach has been described in [5, 6]. IFlow uses information flow control (IFC) to prove modeled information flow properties as it is more powerful and precise than access control or even data flow analysis, since it restricts the propagation of information instead of its release and allows to track information leakage via implicit flows, i.e., through program control flows. IFC on a model level (e.g., [7, 8, 9]) can be applied to a large number of systems and applications. However, it is not easy to use in practice because the theory is quite intricate, and it is easy to lose track of what information is released.

We describe the formal specification that is generated from the UML model. It turns out that the model-driven approach to IFC has several advantages: information flow properties can be expressed in an intuitive, understandable manner, formal proofs become easier, and specification errors are avoided. This paper is an extended and improved version of [10]. The description of the IFlow approach, the generation of the formal model, and comparison to related work are completely new, and all (sub-)sections have been significantly expanded. After introducing the IFlow approach (Sect. 2) and an example (Sect. 3), the main part of the paper describes the generation of the formal model and its advantages over a hand-written specification (Sect. 4). After comparison with related work (Sect. 5) we conclude (Sect. 6).

## 2 The IFlow Approach

We present a model-driven approach for developing information flow-secure systems consisting of several distributed components like mobile apps and web services. As illustrated in Fig. 1, the first step in creating such a system with IFlow is constructing its abstract model with UML (1). This model must capture both the static and the dynamic views on the system, i.e., its component and communication behavior. Components can also make calls to methods which implementation is not part of the model. The model is then extended with security annotations expressing information flow properties that should hold for the modeled system. Such properties define the confidentiality of data managed or accessed by the modeled components and restrict its flow through the system. IFlow assists the developer by providing custom UML profiles and extensions like a simple domain specific language. Additionally, it is possible to explicitly model the information flow properties by referencing the sources and sinks of sensitive

information as well as whether, which and when such information may flow between them.

From the abstract model, partial Java code is generated (3). This "code skeleton" captures the communication structure of the system components as well as calls to "manual methods" with as of yet undefined local functionality. It can be checked with an IFC tool like JOANA to find information flow property violations introduced during the modeling phase. The goal of IFlow is to generate code that can both be analyzed for IF violations and deployed on actual hardware as a distributed system. This paper focuses on solving the challenge of consolidating those two requirements in an IF-safe fashion. In the next step, the developer "fleshes out" the generated skeleton code by implementing the missing local functionality (i.e., "manual methods") by hand (4). By running the IFC analysis again it can be ensured not only that the code skeleton is IF-secure, but also that this manual code does not introduce additional IF leaks.

The abstract model is also transformed into a formal model (2) which uses an abstract state machine and algebraic specifications to define the system's structure and behavior. The proof goals are generated automatically from the information flow properties expressed in the abstract model and can be verified interactively using the theorem prover KIV [11]. An IFC check on the code level can only guarantee the noninterference property (i.e., confidential data does not interfere with "public" information sinks such as certain system components or attributes); a formal model can then be used to verify more complex information flow guarantees (e.g., confidential information flows to a partially trusted system participant only after explicit user's permission).

## 3 An Example Application

The *travel planner app* is a distributed application consisting of two smartphone apps and several web services. Here we present a simplified version that is only used to book flights, and uses only one airline web service. The user enters his travel details into the app. The app connects to a travel agency web service which in turn contacts an airline web service for suitable flights. The found flights are returned via the travel agency to the app. The user selects a flight and books it with his credit card (which is stored in a *credit card center* app) directly at the airline. Finally the airline pays a commission to the travel agency. Fig. 2 shows this behavior as a sequence diagram.

The lifelines represent the (real human) users, the apps, and the web services. Arrows denote message passing as in UML, but a domain-specific language is used that supports, e.g., assignments to provide more information. Additionally, every message is annotated with a security domain in curly brackets, e.g. {User TravelAgency, Airline} which will be explained later in more detail. The sequence diagram shows only the intended behavior of the system. When the user is asked for a confirmation to really book a flight he can of course decline. This must not be modeled explicitly and will end the travel planner app.

Two information flow properties are of interest:

1. The user's credit card data does not flow to the travel agency.

2. The credit card data flows to the airline only after explicit confirmation and declassification.

These properties can also be expressed graphically in UML (see Fig. 3).

The class diagram in Fig. 4 shows a simplified excerpt from the static part of the Travel Planner system. Each component is depicted as a UML class, annotated with the stereotype *Application*, *Service* or *User* to indicate that the component is either an Android app, a Java web service or the user of the system. The UML class representing the CreditCardCenter app includes an attribute named *ccd*, which holds the user's credit card data. The airline maintains a list of available flight offers as represented with its *flightOffers* list attribute. Message data types are modeled as classes, with their attributes representing

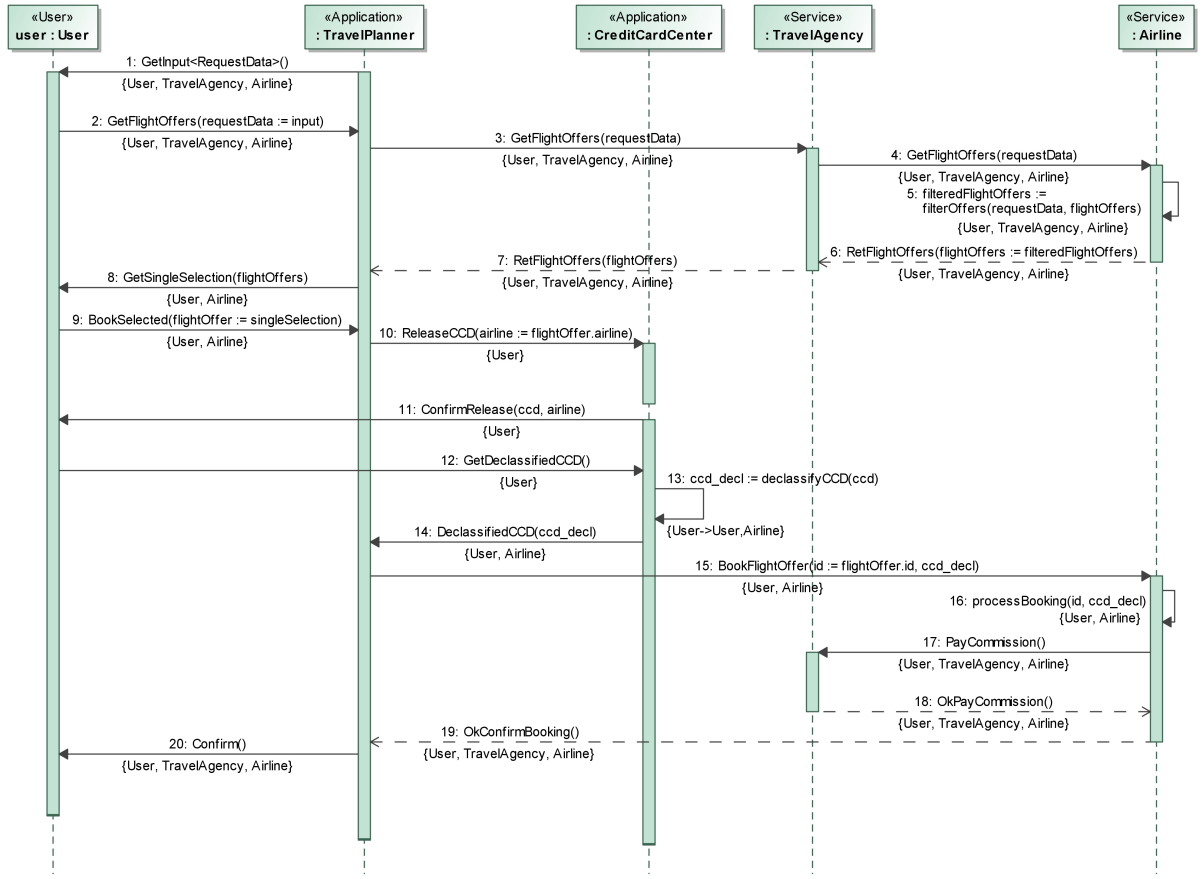


Figure 2: Sequence diagram for the travel planner app.

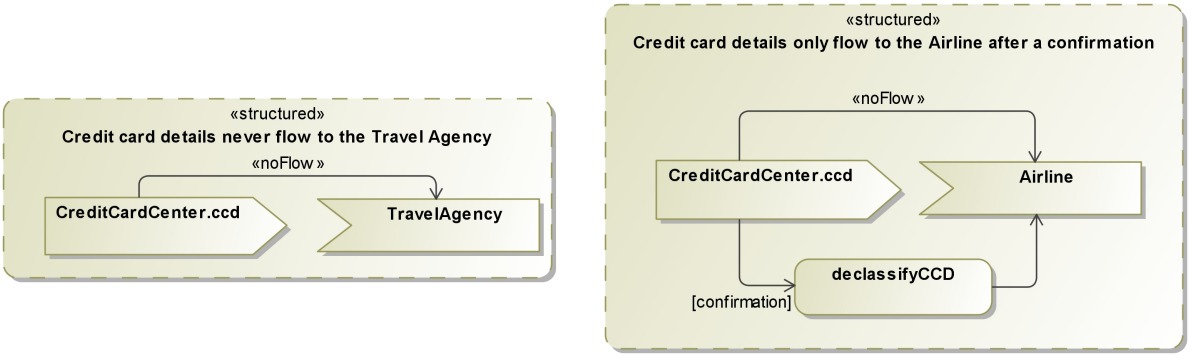


Figure 3: Graphical representation of desired information flow properties.

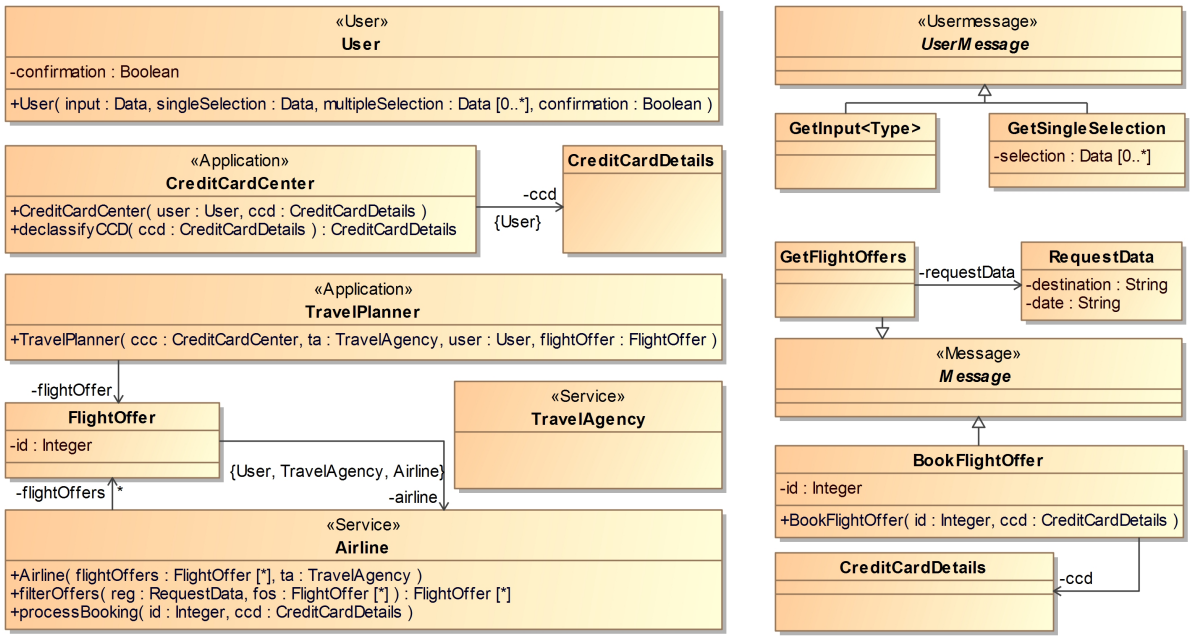


Figure 4: Component diagram of the Travel Planner application

the types of data they can contain. All messages sent to the user component are predefined in an IFlow UML profile and imply standardized GUI elements that can be shown on the user's smartphone.

The class and sequence diagrams contain enough information and are precise enough to generate a Java code skeleton. Together with the properties and a security policy (Fig. 5) a formal specification can be generated as well. The model, formal specification, and proofs can be found on our web page[1].

# 4 The Formal Model

## 4.1 The Basis

The formal framework in the IFlow approach is based on Rushby's intransitive noninterference [8], specifically Rushby's *access control* instance of intransitive noninterference because it uses locations. On the one hand it is very natural to think about information stored in locations that flows to other locations, and on the other hand locations become fields in the generated Java code so that there is a tight connection between the formal model and the code. Rushby's intransitive noninterference is an extension of Goguen's and Meseguer's transitive noninterference [7], and was later improved by van der Meyden [12].

The model-driven approach allows to support different "platforms", i.e., different formal frameworks in this case. This means it is in principle possible to generate another formal model that is based e.g., on Mantel's MAKS framework [9].

First we briefly describe Rushby's framework, and then show how it is used in IFlow. The basic system model is a state transition system with a *structured* state $s$ that consists of locations ("names" in Rushby). Actions $a$ modify the state with a $step$ function $step(s,a)$ that computes the new state. Actions have a security domain $d$ (e.g., *public* or *secret*), and read (*observe*) and/or modify (*alter*) the values of the locations. Security is defined w.r.t. an interference policy defined on domains and a generic

[1] `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/iflow/`

function $output(s,d)$. The interference policy $\rightsquigarrow$ describes allowed information flows, i.e., if $d_1 \rightsquigarrow d_2$ then information may flow from $d_1$ to $d_2$. $output(s,d)$ defines what a security domain can observe in a given system state. The idea is that an attacker has a given security domain and tries to obtain some secret information. More specifically, an attacker should not be able to distinguish (by observing different outputs) between a run of the system where something secret happens and another run where the secret things do not happen.

If $\alpha$ is a sequence of actions then the function $purge(\alpha,d)$ deletes all actions from $\alpha$ that $d$ should not see, i.e., that have a security domain that should not interfere with $d$. If $output(run(\alpha,s_0),d) = output(run(purge(\alpha,d),s_0),d)$ where *run* computes the state that is obtained by applying the actions in $\alpha$ then the system is defined as secure. To put it very simple: *public* observations should not depend on *secret* actions. One definition of *purge* is used for transitive noninterference, a slightly different definition can be used for intransitive noninterference. Unwinding conditions for proving security can be defined that use the information which locations an action reads (*observe*) and modifies (*alter*).

## 4.2 Generating the formal model

This general information flow model is specified in the theorem prover KIV [13, 11, 14] with algebraic specifications (including the proof of the unwinding conditions), and instantiated for a concrete IFlow model. The basis is an abstract state machine ASM [15, 16]. The state is an algebra that is modified by rules of the ASM. In our case the state is a mapping $s$ from locations to values, and the rules are the actions. In order to translate an IFlow model into a formal specification the following aspects must be considered:

1. What are the locations?
2. What are the actions and what is the security domain *dom* of an action?
3. What is the definition of *observe* and *alter*?
4. What is the definition of *output* and $\approx$?

These questions are answered in turn.

### 4.2.1 Locations

It must be emphasized that the definition of locations is important for the formal framework. If too few locations are defined, or locations are reused for different kinds of information then this will introduce undesired interference between security domains, and the security proofs will fail. On the other hand every location must also exist in the generated code because otherwise properties do not transfer from the formal model to the code.

An app like the travel planner can be installed on many smart phones. Therefore each component class can be instantiated many times. Components become agents in the formal model, and each agent gets a unique identifier (simply a natural number). Therefore, a location is a tuple consisting of a *name* and an *agent*. The sequence diagram (Fig. 2) is the main source for locations. The following locations are used:

- Attributes of components. E.g., the credit card data of the user is stored in an attribute named `ccd` of the `CreditCardCenter` component and used in messages 11 and 13. This becomes a location *ccd* $\times$ *CreditCardCenter(n)*.

- Mailboxes. Message passing in the formal model uses mailboxes to store incoming messages until they are processed. Every message has its own mailbox, e.g., *Mailbox(GetFlightOffers)* × *Airline(0)* for message 4. This is adequate because in the generated code a different method is called for each message.
- Local variables. Message arguments automatically become local variables, e.g., *requestData* in message 4: *requestData* × *Airline(0)*. Local variables can also be introduced by assignments (e.g., *ccd_decl* in message 13).
- State variables. They are added automatically during generation of the formal model. Examples are shown in Sect. 4.4.

### 4.2.2 Actions

A suitable definition of actions is also essential for the usability of the approach. An action that uses too many locations intended for different kinds of information again may introduce undesired interferences. Conversely, the actions must correspond to the generated code. The basic building block for an action is receiving a message, processing it, and sending the next message (see Sect. 4.4 for more complicated situations). We consider the block of messages 4–6 in the airline lifeline of Fig. 2 as an example. The airline receives a *GetFlightOffers* message (4), filters suitable offers according to the *requestData* (message 5), and sends the filtered offers back (message 6). The corresponding action is shown in listing 1.

```
1  getFlightOffers−Airline ≡
2      let msg = s(Mailbox(GetFlightOffers) × Airline(0)) in {
3          if (isGetFlightOffersMessage(msg)) then {
4              s(Mailbox(GetFlightOffers) × Airline(0)) := null;
5              s(requestData × Airline(0)) := msg.requestData;
6              s(filteredFlightOffers × Airline(0)) :=
7                  filterOffers(s(requestData × Airline(0)),
8                               s(flightOffers × Airline(0)));
9              s(Mailbox(RetFlightOffers) × TravelAgency(0)) :=
10                 RetFlightOffers(s(filteredFlightOffers × Airline(0)));
11 }}
```

Listing 1: Behavior of the *getFlightOffers-Airline* action.

Lines 2–5 correspond to message 4, lines 6–8 to message 5, and lines 9–10 to message 6. The mailbox is read, checked to contain the correct message, and emptied. Then the operation *filterOffers* is called and the result sent back to the travel agency by putting it in the appropriate mailbox. In the code a method with the same behavior is generated as part of the airline web service.

This approach to define actions (one action receives and processes a message and send another) is appropriate if the incoming and outgoing message, and intermediate operations have the same security domain. This domain is then the security domain of the action. In the example it is {User, TravelAgency, Airline} which is annotated below the messages in the sequence diagram. The security domain must be one of the domains that is defined in the security policy (Fig. 5).

When different domains are used things become more complicated. The developer is responsible for the domain assignments. One of the problems when applying information flow control in practice is this assignment. Often it happens that a system is intuitively secure, but the proof fails because the domain assignment introduces interferences that do not really exist. Our strategy is to generate separate

actions when different domains are used. This happens several times in the sequence diagram in Fig. 2. For example, message 7 is annotated with {User, TravelAgency, Airline}, but message 8 with {User, Airline}. Here, things become more secret because intuitively the travel agency is not allowed to see which flight the user chooses. If messages 7 and 8 are contained in one action it is not clear what the domain of the action is, and each choice can introduce interferences. Therefore the action is split into two, one action for receiving message 7, and one action for sending message 8. The sequential control flow is ensured by introducing a state location so that the second action can only occur after the first. A more complicated situation is explained in detail in Sect. 4.4.

#### 4.2.3 *observe* and *alter*

These two functions are used in the unwinding conditions. Their purpose is described in detail in Sect. 4.3. Both are defined for a domain and return sets of locations. To avoid undesired interferences yet again the sets they return should be as small as possible, but still correct. (Otherwise proof of the unwinding conditions will fail.) We define $observe/alter$ as the union of the $observe/alter$ of all actions with a given domain:

$$observe(d) = \bigcup \{observe(a) : dom(a) = d\}$$
$$alter(d) = \bigcup \{alter(a) : dom(a) = d\}$$

*observe* for one action are simply all read locations, in the example

Mailbox(GetFlightOffers) × Airline(0),
requestData × Airline(0),
flightOffers × Airline(0)

and $alter$ are all written locations, in the example

Mailbox(GetFlightOffers) × Airline(0),
requestData × Airline(0),
filteredFlightOffers × Airline(0),
Mailbox(RetFlightOffers) × TravelAgency(0)

It is trivial to compute these sets during the generation of the formal model.

#### 4.2.4 *output* and ≈

$output(s,d)$ specifies the attacker model. The idea is that an attacker with security domain $d$ obtains some information about state $s$ and computes an output. In IFlow this output function is fixed: The attacker simply outputs all values of all locations he observes:

$$output(s,d) = \{s(l) : l \in observe(d)\}$$

This is the most general definition of *output* that still makes sense (it would not make sense to allow the attacker to look at more secret locations). $s_1 \approx_d s_2$ is used in the unwinding conditions and describes when two states $s_1$ and $s_2$ look alike to an attacker with security domain $d$. $\approx$ is defined by Rushby as

$$s_1 \approx_d s_2 : \Leftrightarrow \forall\, l \in observe(d) : s_1(l) = s_2(l)$$

i.e., two states look identical to an attacker with security domain $d$ if all locations he observes have the same values.

This concludes the formal system specification.

### 4.3 Unwinding theorem

Rushby [8] proves an unwinding theorem that implies security if four unwinding conditions hold. The second condition was later weakened by van der Meyden [12] (we show van der Meyden's version):

- RM1: $s_1 \approx_d s_2 \rightarrow output(s_1, d) = output(s_2, d)$

  If two states look alike to an attacker then the attacker's output in both states is identical.

- RM2: $s_1 \approx_{dom(a)} s_2 \wedge s_1(l) = s_2(l) \wedge l \in alter(dom(a))$
  $\rightarrow step(s_1, a)(l) = step(s_2, a)(l)$

  If two states look alike to the security domain of an action $a$ and this action is executed (with the $step$ function) then every location that is altered by the domain and has the same value in the initial states has the same value in the two new states. (See [12] for a discussion.)

- RM3: $step(s, a)(l) \neq s(l) \rightarrow l \in alter(dom(a))$

  If the execution of an action modifies a location then this location is contained in the $alter$ set for the action's domain.

- AOI: $alter(d_1) \cap observe(d_2) \neq \emptyset \rightarrow d_1 \rightsquigarrow d_2$

  *Alter/observe respects interference:* if a location is altered by one domain $d_1$ and observed by another domain $d_2$ then there is an information flow from $d_1$ to $d_2$. This must be allowed by the interference policy $\rightsquigarrow$.

Condition RM3 basically ensures that the definition of $alter$ is correct: If an action $a$ modifies a location $l$ then $l$ must be contained in the action's domain $alter$ set. Similarly, RM2 ensures that the definition of $observe$ is correct because $\approx$ is defined with $observe$[2]. RM2 and RM3 use the $step$ function, i.e., here every action of the system must be executed (once in the case of RM3 and twice in RM2). Hence, they are the really expensive proof obligations for larger systems. On the other hand they only depend on $alter$ and $observe$ (since $\approx$ is defined with $observe$), but not on $output$ or the interference policy $\rightsquigarrow$.

In IFlow the generation of the formal model from the UML guarantees that

- RM1 is always true because of the fixed definition of $output$.
- RM2 is always true because $observe$ is computed correctly from the UML model.
- RM3 is always true because $alter$ is computed correctly from the UML model.

Therefore, proving

AOI: $alter(d_1) \cap observe(d_2) \neq \emptyset \rightarrow d_1 \rightsquigarrow d_2$

already implies that a system is secure. This shows a great benefit of IFlow's model driven approach.

[2]Consider an action $a \equiv low := high$ with domain $dom(a) = low$. If $high$ is not contained in $observe(low)$ then RM2 does not hold for $l = low$ in two states where $high$ has different values. Hence, RM2 ensures that the set $observe$ is not too small.

### 4.4 Intransitive Noninterference and Implicit Declassification

Intransitive noninterference allows to model secure systems with controlled or partial information release. However, if used excessively or in an arbitrary manner the result may be a system that is secure in terms of the formal definition, but has cryptic or undesired information flows.

Again, the model-driven approach can be helpful because usage of intransitive domains can be controlled. This will be explained with the help of the travel planner example. In general the interference relation may be an arbitrary relation. In IFlow some restrictions apply. Fig. 5 shows the security policy for the example on the left. An edge denotes a (direct) interference, i.e., {User, TravelAgency, Airline} $\rightsquigarrow$ {User, Airline}. The unmarked edges are transitive (i.e., {User, TravelAgency, Airline} also interferes {User}), and the relation is automatically reflexive. Intransitive edges may only be used for declassification (also called downgrading) as shown Fig. 5: Both edges to and from a declassification domain must be intransitive and inverse to the "standard" policy, and a declassification domain must have exactly one incoming and one outgoing edge. In effect, we have a usual transitive policy with possibly some declassification domains. Sect. 4.2 described the standard way to generate actions. When a declassification domain is used (message 13 in the sequence diagram Fig. 2) it must be contained in its own action to avoid undesired interferences. State locations are introduced automatically to ensure that actions/messages 12, 13, and 14 are executed in the correct order.

However, there is another complication. Generating the formal model as described in Sect. 4.2 results in an insecure system, i.e., the only remaining unwinding condition AOI does not hold. The problem are messages 15–17 in Fig. 2: The airline receives a booking message with the credit card details (message 15) that is labeled {User, Airline}, and processes the booking (message 16) at the same security level. However, in message 17 the airline pays a commission to the travel agency that is labeled {User, TravelAgency, Airline}. If messages 15–17 are contained in one action with domain {User, Airline} the action writes to domain {User, TravelAgency, Airline} (the mailbox containing the PayCommission message) which is an illegal information flow.

However, our desired property (*Credit card details never flow to the travel agency*) holds because the PayCommission message does not include the credit card details. The travel agency (or, to be more precise the {User, TravelAgency, Airline} domain) does learn that a booking took place (otherwise it would not receive a commission) but not the parameters of the booking message. Technically, there is an information flow from the {User, Airline} domain to the less secure {User, TravelAgency, Airline} domain, but it does not contain the credit card details. So we want to modify the formal system so that it is secure, but would still detect that the credit card data flows to the travel agency in a faulty model.

The solution is to introduce a new declassification domain from {User, Airline} to {User, TravelAgency, Airline} that leaks only the information that a booking took place but nothing more. Fig. 6 shows how this works.

The original action is split into three actions, one that receives the booking and processes the booking (messages 15 and 16), but then sets a boolean flag to true (donePB) to indicate that it is finished. Then the declassification domain {HighLow} sets the flag doPayC to true to indicate that PayCommission can happen. The third action checks the flag and pays the commission. Both flags are reset after they are read (not shown in Fig. 6). The second action actually is

```
if (s(donePB × Airline(0))) then {
    s(donePB × Airline(0)) := false;
    s(doPayC × Airline(0)) := true; }
```

The action leaks only the information that donePB was true. This modified system is secure, i.e., the unwinding condition AOI holds. On the other hand, an illegal information flow would still be detected. Assume that the PayCommission message includes the credit card details as parameter (PayCommis-

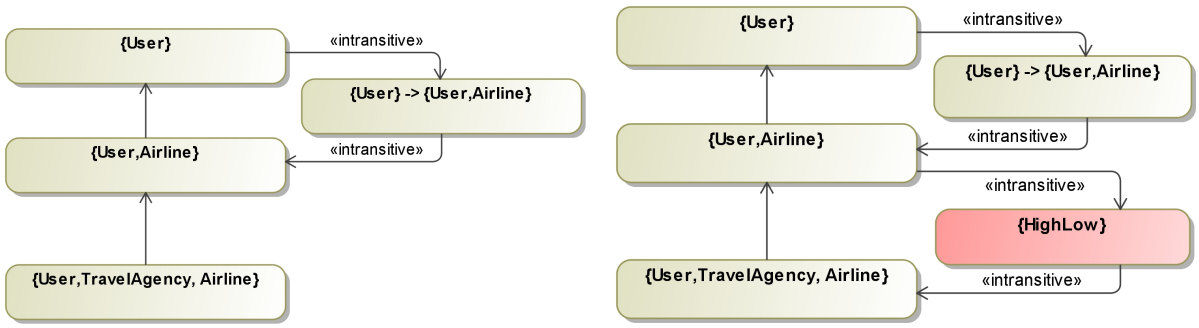


Figure 5: Original security policy (on the left) and new policy with an additional domain for implicit declassification (on the right).

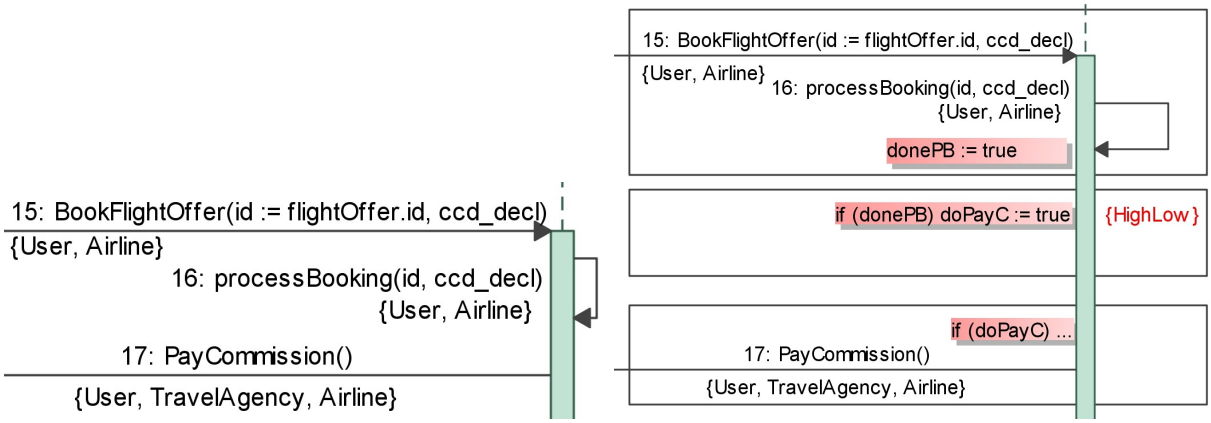


Figure 6: Introduction of new actions.

sion(ccd_decl)). Then the third action in Fig. 6 observes ccd_decl (which must be a local variable), and the first action alters it (because it writes the credit card details to ccd_decl). In effect there would be a direct information flow from {User, Airline} to {User, TravelAgency, Airline} which is forbidden by the policy, i.e. the security proof would fail. Since the {HighLow} domain does not leak the credit card details we have exactly the desired effect.

The main point is that all this (adding a new declassification domain and splitting the original action into three) is done automatically during generation of the formal model. This guarantees that the {HighLow} domain is not used anywhere else (by generating a new unique name) and that the new action only accesses the two flags, and nothing else. An unexperienced user could easily make mistakes if he would model this by hand. To summarize: intransitive noninterference is very flexible, but can lead to undesired results. This can be remedied in part by the model-driven approach.

## 5 Related Work

Several approaches exist that tackle similar challenges as IFlow.

UMLSec [17] is a model-driven approach that allows the development of secure applications with UML. It focuses on cryptographic security but allows basic IF annotations (without declassification) as well as checking the policy. Compared with IFlow, UMLSec is not focused on IFC, and therefore does not have a systematic approach towards IFC. Furthermore, UMLSec does not support automatic code

generation nor code-based IFC.

Seehusen [18] also integrates IF security into a model-driven architecture framework using UML state machines and sequence diagrams. Like IFlow, it includes a formal analysis of the considered IF properties, but describes the automatic code generation and the consideration of IF properties on the code level as future work. Alghathbar et al. [19] describes the modeling of IF in applications using UML. IF policies are specified separately using Horn clauses and checked against the information obtained from the UML diagrams. Compared with IFlow, it does not model the IF policies with UML, nor does it consider the IF on the code level.

Heldal et al. [20, 21] introduce a UML profile that incorporates the decentralized label model [22] into the UML. It allows the annotation of UML elements with Jif labels in order to automatically generate Jif code from the UML model. However, the Jif-style annotation already proved to be non-trivial on the code level (see [23, 24]), while [21] notes that the actual automatic Jif code generation is future work.

To the best of our knowledge, IFlow is the only model-driven approach for information flow control that includes both verification in a formal model as well as automatic code generation.

# 6 Conclusion

Information flow control is a possibility to guarantee the privacy and desired release of our data in an always connected world. Noninterference is one formal technique to ensure information flow control. It guarantees that in a system where private and public things happen the private actions do not interfere with the public ones. Since sometimes information like a user's credit card data should be released the notion of intransitive noninterference was introduced. However, it is difficult to apply this formal framework to actual applications. The proofs may fail for a secure system because actions or domain assignments were unwisely chosen. On the other hand, if too much information is declassified then the proofs may succeed, but the application is intuitively insecure.

A possible solution to this problem is a model-driven approach that supports the developer in the application of the formal framework. IFlow is such a model-driven approach that supports the development of distributed applications consisting of mobile apps and web services with guaranteed and intuitive information flow properties. The application is modeled with UML using class and sequence diagrams. From the model partial Java code can be generated as well as a formal specification based on intransitive noninterference. The generation takes care of the definition of what an action is and of two important auxiliary definitions needed for the verification. This greatly simplifies the proofs, and supports the developer in avoiding errors because he can concentrate on the important aspects, i.e. the information flow properties. Since the code and the formal model are generated together from the same model properties proven for the formal specification also hold for the code.

# References

[1] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. of the 20th USENIX conference on Security (SEC'11), Berkeley, California, USA*. USENIX Association, 2011, pp. 21–21.

[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. of the 18th Conference on Computer and Communications Security (CCS'11), Chicago, Illinois, USA*. ACM, October 2011, pp. 627–638.

[3] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS - Operating Systems Review*, vol. 22, no. 4, pp. 36–38, October 1988.

[4] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proc. of the 18th Network and Distributed System Security Symposium (NDSS'11), San Diego, California, USA*. The Internet Society, February 2011, pp. 1–17.

[5] K. Katkalov, K. Stenzel, M. Borek, and W. Reif, "Model-driven development of information flow-secure systems with IFlow," *ASE Science Journal*, vol. 2, no. 2, pp. 65–82, 2013.

[6] K. Katkalov, K. Stenzel, M. Borek, and W. Reif, "Model-driven development of information flow-secure systems with IFlow," in *Proc. of the 5th International Conference on Social Computing (SocialCom'13), Alexandria, Virginia, USA*. IEEE, September 2013, pp. 51–56.

[7] J. Goguen and J. Meseguer, "Security policy and security models," in *Proc. of the 3rd Symposium on Security and Privacy (SP'82), Oakland, California, USA*. IEEE, April 1982, pp. 11–20.

[8] J. Rushby, "Noninterference, Transitivity, and Channel-Control Security Policies," SRI International, Tech. Rep. CSL-92-02, 1992, available at http://www.csl.sri.com/papers/csl-92-2/.

[9] H. Mantel, "Possibilistic definitions of security - an assembly kit," in *Proc. of the 13th IEEE Computer Security Foundations Workshop (CSFW'00), Cambridge, England*. IEEE, July 2000, pp. 185–199.

[10] K. Stenzel, K. Katkalov, M. Borek, and W. Reif, "Formalizing information flow control in a model-driven approach," in *Proc. of the 2nd International Conference on Information and Communication Technology (ICT-EurAsia'14), Bali Indonesia, LNCS*, vol. 8407. Springer-Verlag, April 2014, pp. 456–461.

[11] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums, "Formal system development with KIV," in *Proc. of the 3rd International Conference on Fundamental Approaches to Software Engineering (FASE'00), Berlin, Germany, LNCS*, vol. 1783. Springer-Verlag, March–April 2000, pp. 363–366.

[12] R. van der Meyden, "What, indeed, is intransitive noninterference?" in *Proc. of the 12th European Symposium on Research in Computer Security (ESORICS'07), Dresden, Germany, LNCS*, vol. 4734. Springer-Verlag, September 2007, pp. 235–250.

[13] KIV homepage, http://www.informatik.uni-augsburg.de/swt/kiv.

[14] D. Haneberg, S. Bäumler, M. Balser, H. Grandy, F. Ortmeier, W. Reif, G. Schellhorn, J. Schmitt, and K. Stenzel, "The User Interface of the KIV Verification System — A System Description," 2006, http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/2006-UITP05-KIV/.

[15] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1995, pp. 9–36.

[16] E. Börger and R. F. Stärk, *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[17] J. Jürjens, *Secure systems development with UML*. Springer Verlag, 2005.

[18] F. Seehusen, "Model-driven security: Exemplified for information flow properties and policies," Ph.D. dissertation, Faculty of Mathematics and Natural Sciences, University of Oslo, January 2009.

[19] K. Alghathbar, C. Farkas, and D. Wijesekera, "Securing UML information flow using FlowUML," *Journal of Research and Practice in Information Technology*, vol. 38, no. 1, pp. 111–121, 2006.

[20] R. Heldal and F. Hultin, "Bridging model-based and language-based security," in *Proc. of the 8th European Symposium on Research in Computer Security (ESORICS'03),Gjøvik, Norway, LNCS*. Springer-Verlag, October 2003, vol. 2808, pp. 235–252.

[21] R. Heldal, S. Schlager, and J. Bende, "Supporting confidentiality in UML: A profile for the decentralized label model," in *Proc. of the 3rd International Workshop on Critical Systems Development with UML (CSDUML'04), Lisbon, Portugal*, October 2004, pp. 56–70.

[22] A. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, pp. 410–442, 2000.

[23] C. Hammer, "Experiences with PDG-based IFC," in *Proc. of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS'10), Pisa, Italy, LNCS*, vol. 5965. Springer-Verlag, February 2010, pp. 44–60.

[24] K. Katkalov, P. Fischer, K. Stenzel, N. Moebius, and W. Reif, "Evaluation of Jif and Joana as information flow analyzers in a model-driven approach," in *Proc. of the 7th International Workshop on Data Privacy Management and Autonomous Spontaneous Security (DPM'12), Pisa, Italy, LNCS*. Springer-Verlag, 2013,

vol. 7731, pp. 174–186.

---

## Author Biography

**Kurt Stenzel** is Professor of Computer Science at Columbia University. He received his Ph.D. from NYU Courant Institute in 1979 and has been on the faculty of Columbia ever since. He has published extensively in the areas of parallel computing, AI knowledge-based systems, data mining and most recently computer security and intrusion detection systems. His research has been supported by DARPA, NSF, ONR, NSA, CIA, IARPA, DHS and numerous companies and state agencies over the years while at Columbia.

**Kuzman Katkalov** was born in Rostov-on-Don, Russia on February 22, 1986, and moved to Germany in 1996. He graduated at the University of Augsburg (Germany) in 2011, where he has been working since as a research assistant. His main research interests are in model-driven development as well as protocol and information flow security. He is now working on the project IFlow within the Reliably Secure Software Systems priority programme of the German Research Foundation.

**Marian Borek** was born in Boskovice, Czech Republic on July 06, 1986, and moved to Germany in 1990. He graduated at the University of Augsburg (Germany) in 2011, where he has been working since as a research assistant. His main research topics of interest are security-critical applications and model driven development. He is now working on the project SecureMDD that is funded by the German Research Foundation.

**Wolfgang Reif** studied Computer Sciences in Karlsruhe, Germany, and began working as a research assistant there. After a period at the University of Ulm, Germany, he moved to Augsburg University where he received his Ph.D. in computer sciences in 2005. Since then he is working in Augsburg on formal Java verification, interactive theorem proving, model-driven development of security-critical systems, and information flow control.