Model-Driven Development of Information Flow-Secure Systems with IFlow

Kuzman Katkalov, Kurt Stenzel, Marian Borek and Wolfgang Reif
Institute for Software and Systems Engineering
Augsburg University, 86135 Augsburg, Germany
{kuzman.katkalov,stenzel,borek,reif}@informatik.uni-augsburg.de

***Abstract*—In our increasingly interconnected world, privacy can seem like an unattainable goal. We are surrounded by countless devices and web services that acquire and collect our personal data as we interact with them. In many cases, the confidentiality of such data is not guaranteed and is frequently (if not always intentionally) violated. Smartphone apps and Internet web services in particular are known to often leak their users' confidential data to other users or (affiliated) third parties. We present a novel model-driven approach called IFlow that allows the development of distributed applications consisting of mobile apps and web services with secure information flow. In IFlow, a UML model of an information flow-sensitive application is used to automatically generate deployable app and web service code as well as a formal model. By employing automatic, language-based information flow control as well as interactive verification, IFlow enables the developer to give verifiable guarantees to the user about how his private data is being treated by the application.**

***Index Terms*—model-driven software development, information flow control, mobile apps, web services**

I. Introduction

As always-connected smartphones become ubiquitous, personal privacy is quickly becoming a fleeting ideal that is very hard to achieve or guarantee. Some even go as far as declaring it unattainable, demanding that we ourselves adapt to and embrace the unavoidable post-privacy world. Indeed, such devices and the web services they communicate with become entrusted with an increasing amount of personal information. This information ranges from the current location of the smartphone's user to the contact data of his friends and business clients, his private photos or even confidential payment data. Inhibitions to share such information are dwindling among users, while those aware of the risks of disclosing their personal data are unable to verify whether it is being handled confidentially. Even worse, such information is frequently collected without the user's knowledge or his explicit consent [1].

Reports of personal data being leaked via smartphone apps and web services frequently surface in the media as well as academic publications [2]. As a user, one has to rely on current smartphone security mechanisms and the privacy policy of the app or service to keep his data private. However, neither can provide verifiable guarantees that his information is indeed handled confidentially.

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).

One mechanism meant to prevent such leaks is a permissions system that can be found on most popular smartphone platforms including Android, iOS and Windows Phone. Every smartphone app developed for any of those platforms must therefore specify a list of permissions which, e.g., grant it "access to the GPS sensor" or "access to the Internet", which the user has to agree with prior to installing the app. However, this security mechanism is far too coarse-grained to guarantee the confidentiality of user's data. If an app has both the permissions to access the web as well as read the GPS sensor output, the user has no way of knowing whether the app keeps his current location private while only using the Internet connection to display ads, or transparently leaks his GPS position to a third party. Other mechanisms such as application sandboxing and access control prevent apps from accessing each other's data or memory, but do not restrict the information flow (IF) when they use proper, intended communication interfaces. Thus, guaranteeing that no information about certain confidential data ever leaks to another app or service requires an information flow control (IFC) mechanism and is impossible to achieve using current smartphone security mechanisms.

We present a novel model-driven approach called IFlow that allows the developer to model real-world distributed applications consisting of smartphone apps and web services and provide verifiable guarantees about their information flow. The approach includes modeling guidelines in order to create a UML model of such a system, which is then used by the IFlow model transformation framework to automatically generate partial Java code as well as a formal model of the system. The generated code is then extended manually with non-IF-critical functionality and can be deployed directly on target hardware.

IFlow employs automatic, language-based information flow control to guarantee the noninterference security of the final system code, while using the formal model to allow the interactive verification of more complex, application-specific information flow properties. Due to the IF-preserving refinement relation between the formal model and the generated code, properties that have been verified using the formal model are also satisfied for the final, deployed application. Since an external reviewer can certify that such properties indeed hold for the application model and final code, the user does not have to trust either the application modeler or programmer.

Section II gives an overview over the IFlow approach. Section III explains the modeling guidelines in detail using

978-0-7695-5137-1/13 $26.00 © 2013 IEEE
DOI 10.1109/SocialCom.2013.14

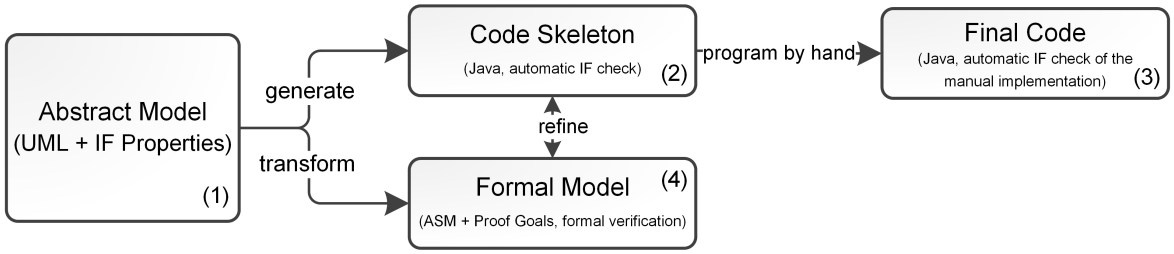


Fig. 1. The IFlow approach

a case study application as an example. Section IV briefly outlines the mapping of the abstract application model to code, and section V briefly describes the formal IFlow framework. Section VI gives an overview over related work, while section VII concludes this paper.

II. The IFlow Approach

In this paper, we present our model-driven approach called IFlow for developing information flow-secure distributed applications. Such applications may consist of components such as smartphone apps as well as web services that communicate with each other in order to accomplish a common goal. The focus of IFlow lies in enabling the developer of those applications to give verifiable guarantees about how they treat confidential data and which application components can learn about it.

Fig. 1 outlines our model-driven approach. The IFlow developer starts off by creating an abstract UML model of his information flow-sensitive application (1). This model must contain all necessary information about the application's components, including a static view that captures the components' structure using class diagrams as well as a dynamic view using sequence diagrams that prescribe how such components communicate with each other and the user. Their local, non-IF-critical functionality is modeled as invocations of component methods, i.e., "manual methods", which have to be implemented by hand. Application-specific IF properties such as "user's confidential data never leaks to any web service" are also modeled in UML using both an intuitive, high-level representation as well as security level annotations of IFlow UML elements that instantiate this property. Such annotations restrict the information flow within the application and can be checked against the application's behavior using automatic IFC combined with formal verification.

From this abstract UML model of the application, a Java "code skeleton" for each application component is generated automatically (2). Such code implements the component structure as well as component communication and calls to the manual methods that have to be coded by hand. The generated code skeleton is linked against the interface of an IFlow-specific Java framework abstracting from platform-specific API calls, of which several implementations exist. The framework enables the developer to deploy the component code on target hardware such as Android smartphones or Play[1] web services, while a platform-independent implementation of the framework allows him to employ an automatic IFC tool such as JOANA[2] in order to check the resulting code for IF leaks. This static IFC check can assure that the modeled application-specific noninterference properties hold for the entire distributed application. Thus, it can be shown that information about data classified as confidential (commonly referred to as *high*) never influences public, visible (i.e., *low*) data locations such as messages sent to non-trustworthy components. Both direct as well as indirect dependencies, which can arise due to direct data assignments (e.g., $low = high;$) or control flow dependent on high values (e.g., $if(high)\{low = 0;\}$) must be considered by the IFC tool. The formalization of the noninterference property for the IFC tool is derived directly from the modeled, application-specific IF properties and security level annotations of the abstract model.

After the code skeleton has been generated and checked for information flow leaks automatically in order to assure that the model does not violate the noninterference property, the developer implements the manual method bodies using Java. This allows the model of the application to stay relatively simple and well-structured, since complex, platform-specific local functionality is abstracted from. However, such manual implementation might introduce new IF leaks due to an incompetent or malicious developer. Therefore, the resulting code must be checked again using IFC. This final code represents the fully functional application, and can be deployed as a distributed system on real target hardware (3).

Furthermore, the UML model is also transformed into a formal model based on an abstract state machine (ASM) and algebraic specifications, which represent the structure and behavior of the modeled, distributed application (4). It is used as input for the interactive theorem prover KIV [3], [4] in order to verify IF properties beyond simple noninterference such as constraints on data declassification, i.e., the lowering of its security level. This enables the developer to not only give guarantees that confidential data never leaks to untrusted application components, but also to allow such leaks under certain circumstances such as after user confirmation. This is done since most real-world applications require declassification to

[1]Play framework: http://www.playframework.com

[2]JOANA (Java Object-sensitive ANAlysis) - Information Flow Control Framework for Java, http://pp.ipd.kit.edu/projects/joana/

function correctly (e.g., during authentication, information about the confidential password must be declassified in order to notify the user whether the credentials he provided are correct). However, declassification circumvents the transitive noninterference policy that confidential, high data may never influence low application values, which is why it has to be treated as IF-critical and its use must be analyzed formally. IF-specific annotations in the abstract UML model are used in the formal model to assign security levels to component actions such as receiving an incoming message and to instantiate the noninterference property. Proof goals for the interactive verification are generated automatically from the abstract application model. Due to the refinement relation between the formal model and the generated code, properties shown on the formal level also hold for the final application implementation.

III. Travel Planner: A Case Study

The current app stores for the popular mobile platforms such as Android (Google Play Store), iOS (App Store) and Windows Phone (Windows Phone Store) only provide rudimentary mechanisms and analyses to protect the customer's privacy from the prying eyes of app developers and associated third parties. One possible application of the IFlow approach is developing applications and apps for an app store that focuses on providing apps with formally guaranteed privacy properties to its users.

Travel Planner is a distributed application modeled with IFlow which allows a user to find and book relevant flights for his journey (cf. Fig. 2(b)). Using the *TravelPlanner* app (TP) on his smartphone, the user can query a *TravelAgency* web service (TA) for available and suitable flight offers (Fig. 2(b), (1)). After selecting one offer, he is able to book the flight directly with the *Airline* web service (A) with his credit card data (Fig. 2(b), (2)), which is managed by the *CreditCardCenter* app (CCC) on his smartphone. The Airline concludes by paying a commission to the TA (Fig. 2(b), (3)). The central noninterference IF property of the Travel Planner application that must be ensured is that the user's confidential credit card data never leaks to the TA. Furthermore, the Airline web service must only receive this data after an explicit confirmation by the user. In the following, we will explain the IFlow modeling guidelines using the example of the Travel Planner application[3].

A. Static view

Fig. 2(a) depicts the static view of the application as modeled with UML class diagrams using IFlow. The component diagram models each component of the application as a class. Classes representing mobile smartphone apps, namely `TravelPlanner` and `CreditCardCenter`, are extended with the stereotype *Application* from the UML profile provided by IFlow. Classes representing web services, in this case `TravelAgency` and `Airline`, are extended with the stereotype *Service* from the same profile. The class `User` is predefined and represents an interactive user interface on the smartphone. Application messages are also modeled as classes but omitted in Fig. 2.

Manual methods are modeled as UML class operations such as the `filterOffers(...)` operation of the `Airline` class, which is meant to return a list of flight offers that satisfy the user's request parameters.

B. Dynamic view

In order to define the behavior of the distributed application in IFlow, UML sequence diagrams are used. Fig. 3 outlines the message flow between the components of the Travel Planner application. Each lifeline represents a component from Fig. 2(a). IFlow employs a simple domain specific language (DSL) that allows the modeler to send and receive messages, access component attributes and object fields, create and access local variables and call manual methods. Its statements are textually embedded into the sequence diagram.

Sending and receiving a message is represented as a call to the message class constructor. Parameters depict the message content. Such content can be a sending component's local variable or class attribute. The receiver of the message can then access this content, i.e., the constructor parameter by its name as a local variable. If the constructor parameter is an assignment, this content (i.e., the right side of the assignment) is renamed when the message is received.

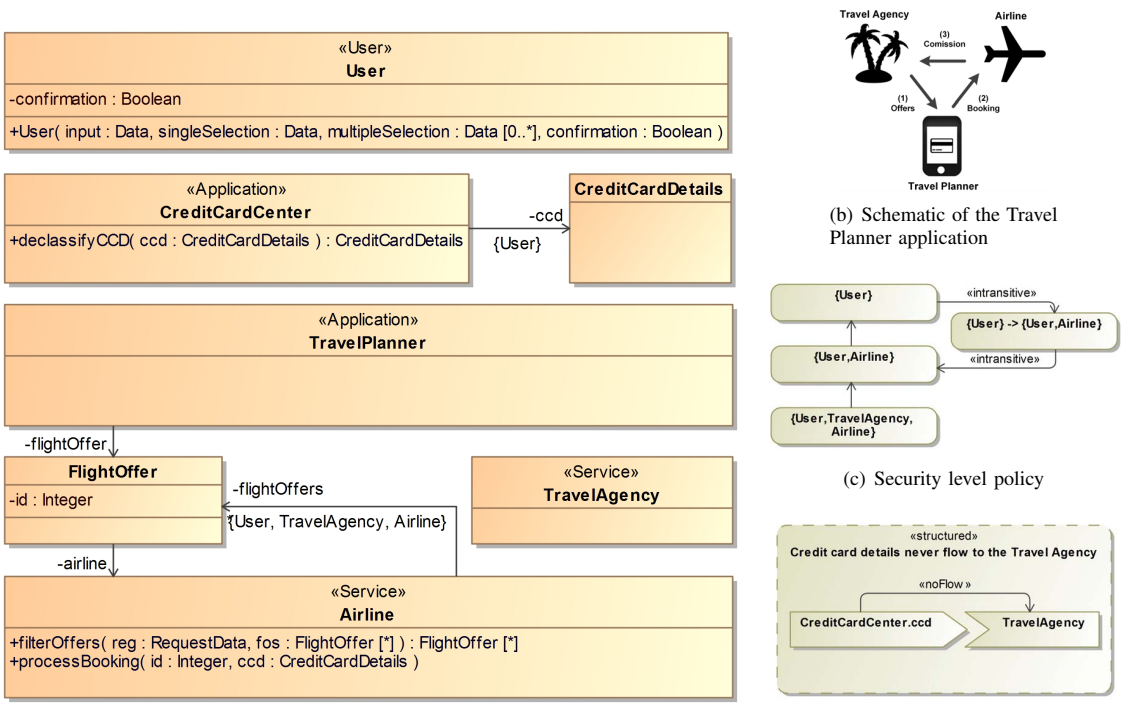
The following paragraphs explain the behavior and the message flow of the Travel Planner application in detail.

a) Requesting relevant flight offers: In Fig. 3, the first message send by the TP app after it has been launched is the predefined user message `GetInput<RequestData>()` to the User lifeline (1). It requests the user to input his journey parameters such as date and destination, which are modeled as attributes of a `RequestData` class. Since the `GetInput<Type>` class has no attributes, the constructor call has no parameters. The User component replies with the `GetFlightOffers` message (2), which is modeled to hold one attribute of type `RequestData`. The message must therefore include an instance of the `RequestData` class.

The user's GUI input and selection are represented as the class attributes of the User component such as `input` and `singleSelection`. The User component therefore writes its `input` parameter into the `GetFlightOffers` message, which is then renamed to `requestData`. The TP app forwards the user's request data to the TA service (3), which in turn forwards it to the Airline service (4).

Local functionality such as setting a component's attribute, creating a new local variable or calling a manual method is modeled as to-self UML messages. Upon receiving the user's request data, the Airline calls its manual method `filterOffers` (5) by providing it with the received request data and the Airline component attribute `flightOffers` representing a list of flight offers. Its return value is assigned to an implicitly defined local variable `filteredFlightOffers`. After those offers are returned to the TP (6-7), they are presented to the

[3]The full UML model can be found at http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/iflow/



(a) Excerpt from the component diagram

(b) Schematic of the Travel Planner application

(c) Security level policy

(d) Noninterference property "credit card details never flow to the TA"

Fig. 2. Static view of the Travel Planner application

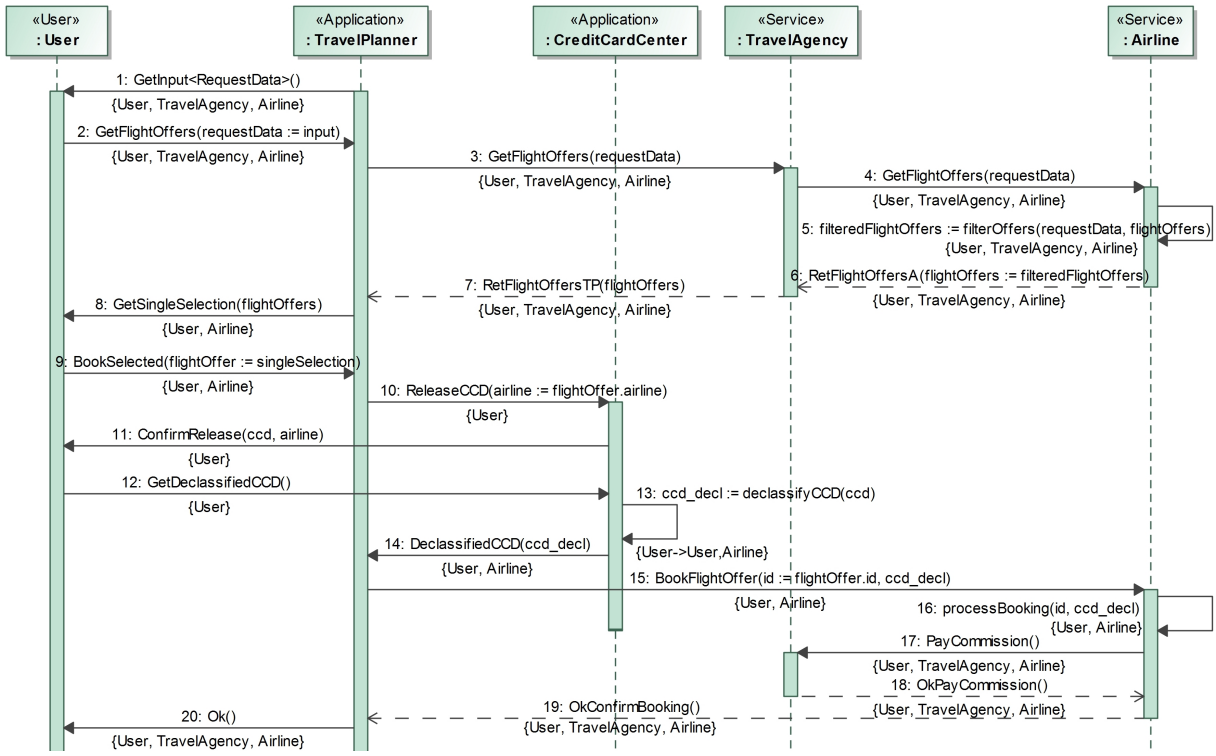


Fig. 3. Dynamic view of the Travel Planner application

user via the predefined `GetSingleSelection` message as a list of elements. The user is prompted to select one of the available offers (his selection is represented as the `singleSelection` attribute of the User component), which is then sent to the TP via the `BookSelected` message (9).

b) Releasing the credit card details: Before the TP app is ready to book the selected flight offer, it first has to request the user's credit card data (ccd) from the CCC app. It does so by informing the CCC of the intended receiver of the ccd, i.e., the airline (10). In turn, the CCC inquires the user whether he agrees with disclosing his ccd to the airline (11).

c) Booking the flight: Once the user agrees to sharing his ccd with the airline (12), they are sent to the TP (14) which in turn forwards them to the Airline component together with the selected flight offer id in order to book the flight (15). To process the order, the Airline service calls the manual method `processBooking` (16) and pays the TA for placing a customer (17). The user is then notified of a successful booking (18-20).

C. IF properties and security annotations

a) Modeling the IF properties: In order to guarantee that sensitive data stays confidential during every possible program run, the application modeler first has to define which data he considers private. IFlow allows the intuitive modeling of IF properties using activity diagrams. Activity nodes embedded in structured nodes, each representing an IF property, depict "sources" and "sinks" of information which can be connected using UML control flows, restricting the information flow between them. Such nodes can represent IFlow application components or one of their attributes, thus stating that certain data stored in such attributes, i.e., "sources" of information never leaks to specified component, i.e., the "sink" of information.

Fig. 2(d) depicts the noninterference property "user's credit card details never flow to the Travel Agency". The activity node on the left specifies the `ccd` attribute of the `CreditCardCenter` component as a confidential information "source". It is connected via a directed control flow annotated with the IFlow *noFlow* stereotype to the node representing the `TravelAgency` component, thus forbidding any kind of information flow from the user's confidential credit card data to the untrustworthy TA.

Even though this particular property can be seen to hold simply by looking at the modeled behavior, IFlow allows to give formal and provable guarantees that the application is IF secure even if it is significantly more complex than the presented case study.

b) Modeling the security level policy: In order to instantiate the modeled IF property, IFlow model elements must be annotated with security levels. Such levels or "domains" which are part of a security level policy are used in many model- and language based IF theories (e.g., [5]–[9]) to restrict the information flow between the annotated model or source code elements. E.g., the simple policy $high \not\rightsquigarrow low$ contains the security levels low and $high$, forbidding the IF from information "sources" annotated as $high$ to "sinks" annotated as low. In IFlow, the policy can contain more than two security levels in order to be able to encompass several properties as well as declassification.

Fig. 2(c) depicts the IFlow UML representation of the security policy as used in the Travel Planner case study. The security levels of this policy are modeled as activity nodes, which are connected by implicitly transitive, directed control flows indicating the allowed information flow. Such levels are named as sets of application components; generally speaking, the more components are included in the set, the less confidential is the corresponding security level. The intuition behind such sets conveys which agents are able to access the information belonging to that security level. Here, the security level $\{$*User, TravelAgency, Airline*$\}$ is less confidential than $\{$*User, Airline*$\}$, which in turn is less confidential than $\{$*User*$\}$.

Declassification, i.e., the lowering of the security level of data plays a major role in real world applications. It is reflected in the IFlow security policy as activity nodes containing two sets of components separated with an arrow, such as the $\{$*User*$\}->\{$*User, TravelAgency*$\}$ security level in Fig. 2(c). Control flows annotated with the *intransitive* stereotype leading to and from this node indicate the intransitive, allowed IF from a higher to a lower security level.

D. Annotating the model with security levels

After defining the desired IF property and an appropriate security policy, some of the elements of the IFlow application model have to be annotated with security levels from the security policy in order to instantiate the noninterference IF property. Such annotations are reflected in both the static (see Fig. 2(a)) and dynamic (see Fig. 3) views as UML constraints on class associations and sequence messages. By applying the $\{$`User`$\}$ UML constraint to the `ccd` association of the `CreditCardCenter` component class (see Fig. 2(a)), the modeler defines the security level of user's credit card data as highly confidential. In the sequence diagram, annotations on the sequence messages define the upper security level bound of information which can be sent using those messages. Thus, by annotating every message directed at the untrustworthy TA with $\{$`User, TravelAgency, Airline`$\}$ (Fig. 3, messages (3), (6) and (17)), the modeler specifies that it may never receive any information derived from the confidential ccd annotated with $\{$`User`$\}$. I.e., such annotations restrict the IF within the distributed application; we will consider their automatic generation in future work.

Furthermore, the annotations in Fig. 3 enforce that such ccd only flow to the Airline (Fig. 3, messages (4), (5), (15), (16) and (18)) after they have been declassified using a declassification method (Fig. 3, message (13)) annotated with the declassification security level from the modeled security policy.

IV. Mapping to Code

The IFlow approach allows the developer to generate source code from his application model automatically by employing multi-stage model-to-model and model-to-code transformations. Since such code is missing the real implementation

of manual methods, we refer to it as "skeleton code". It is used both for automatic IF analysis as well as deployment on real hardware, and must therefore implement modeled functionality on the target platform (as Android apps and Play web services) while still being analyzable with an IFC tool such as JOANA, which is only able to analyze the bytecode of platform-independent, non-distributed Java applications. In order to achieve this goal, we generate code that is implemented against the interface of a platform-independent Java "IF framework" tailored to IFlow applications, abstracting from platform-specific functionality and API calls. IFlow provides separate framework implementations that allow to deploy the resulting code as Android apps and Play web services, or a platform-independent, local Java application that can be checked with JOANA.

V. Formal Framework

The formal specification is generated automatically, and is a faithful representation of the distributed system that is modeled in the abstract UML model. The IFlow approach guarantees that properties proved in the formal model also hold for the running code. It is possible to prove simple noninterference properties, but this is not necessary if the automatic IFC check on the code succeeds. However, it is also possible to prove other properties, e.g. noninterference between different instances of one component (this is very difficult to check on the code level because it requires an analysis that can handle information flows between objects), or exactly what is declassified, or functional properties like "booking data is sent to the *correct* airline".

The formal specification uses van der Meyden's definition of intransitive noninterference [5] which is based on Rushby [6] which in turn is an extension of Goguen and Meseguers transitive noninterference [9]. This general information flow model is specified in the theorem prover KIV [3], [4] with algebraic specifications, and instantiated for a concrete IFlow model. The state transition system is defined with Abstract State Machines (ASM [10]). The main noninterference property (see Fig. 2(d)) becomes a proof obligation

$$\{User\} \not\rightsquigarrow \{User,\ TravelAgency,\ Airline\}$$

as described in section III-D which is proved automatically.

VI. Related Work

Several approaches exist that tackle similar challenges as IFlow. Kraemer [11] introduces a model-driven approach for creating Android apps. It allows the specification of Android app "building blocks" in UML, which can be combined and reused to develop a functional Android application. However, the security of the resulting app is not considered.

UMLSec [12] is a model-driven approach that allows the development of secure applications with UML. It focuses on cryptographic security but also allows basic IF annotation of UML elements as well as the formal verification of the corresponding IF properties. Compared with IFlow, the approach does not include automatic code generation nor code-based security.

Seehusen [13] 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.

To the best of our knowledge, IFlow is the only approach that allows the model-driven development of distributed applications that includes both verification on a formal model as well as automatic IFC check of automatically generated, deployable code.

VII. Outlook and Conclusion

IFlow is a model-driven approach that enables the developer to model distributed applications consisting of smartphone apps and web services as well as their IF properties using intuitive modeling guidelines. The resulting UML specification is then automatically transformed into a Java code skeleton of the application that can be checked automatically as well as a formal model which can be used for formal verification. In the future, we plan to expand the modeling guidelines to allow the modeling of such IF properties, consider external and malicious application components and formally prove the refinement relation between the formal model and the generated code.

References

[1] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications." in *ACM Conference on Computer and Communications Security'11*, 2011, pp. 639–652.

[2] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 21–21.

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

[4] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums, "Formal system development with KIV," in *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[5] R. van der Meyden, "What, indeed, is intransitive noninterference? (extended abstract)," in *Proc. European Symposium on Research in Computer Security*, vol. 4734. Springer LNCS, 2007, pp. 235–250.

[6] J. Rushby, "Noninterference, Transitivity, and Channel-Control Security Policies," SRI International, Tech. Rep. CSL-92-02, 1992.

[7] J. Graf, M. Hecker, and M. Mohr, "Using JOANA for information flow control in java programs - a practical guide," in *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*. Springer LNI 215, 2013.

[8] H. Mantel, "Possibilistic definitions of security - an assembly kit," in *IEEE Computer Security Foundations Workshop*. IEEE Press, 2000.

[9] J. Goguen and J. Meseguer, "Security Policy and Security Models," in *Symposium on Security and Privacy*. IEEE, 1982, pp. 11 – 20.

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

[11] F. A. Kraemer, "Engineering Android Applications Based on UML Activities," in *Model Driven Engineering Languages and Systems*. Springer LNCS 6981, 2011.

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

[13] F. Seehusen, "Model-Driven Security: Exemplified for Information Flow Properties and Policies," Ph.D. dissertation, Faculty of Mathematics and Natural Sciences, University of Oslo, Jan. 2009.