OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

Modeling information flow properties with UML

Kuzman Katkalov, Kurt Stenzel, Marian Borek, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Katkalov, Kuzman, Kurt Stenzel, Marian Borek, and Wolfgang Reif. 2015. "Modeling information flow properties with UML." In *2015 7th International Conference on New Technologies, Mobility and Security (NTMS), 27-29 July 2015, Paris, France*, edited by Mohamad Badra, Azzedine Boukerche, and Pascal Urien. Piscataway, NJ: IEEE. https://doi.org/10.1109/ntms.2015.7266507.

Nutzungsbedingungen / Terms of use: **licgercopyright**

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Modeling Information Flow Properties with UML

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

***Abstract*—Providing guarantees regarding the privacy of sensitive information in a distributed system consisting of mobile apps and services is a challenging task. Our IFlow approach allows the model-driven development of such systems, as well as the automatic generation of code and a formal model. In this paper, we introduce modeling guidelines for the design of intuitive, flexible and expressive information flow properties with UML. Further, we show how these properties can be guaranteed using a combination of automatic language-based information flow control and model-based interactive verification.**

***Index Terms*—privacy, information flow, model-driven software development**

I. Introduction

Systems spanning across numerous mobile apps and web services are becoming increasingly more common. Keeping sensitive information private in such systems is a difficult challenge, as proved by frequent reports on information leaks that keep appearing in popular mobile apps (see e.g. [1]). Current security mechanisms such as the Android permission system fail to sufficiently address this issue as they are not nearly fine-grained or expressive enough [2]. An information flow (IF)-aware development approach is a way of providing strong privacy guarantees for systems that handle sensitive user information such as payment or location data. IFlow is a model-driven approach to develop distributed applications consisting of mobile apps and web services. It aids the developer in providing strong IF guarantees by combining language-based information flow control (IFC) with model-based formal verification of IF properties.

In order to formally show that a system is secure w.r.t. information flow, the desired security requirements have to be formalized. In information flow theory, this is usually done by defining a security policy and assigning security domains to information locations or system actions [3]. However, such a mapping is not immediately intuitive, since it assumes an understanding of information flow and also requires the user to inspect the entire system model and its security annotations in order to grasp its security guarantees. Neither can this approach be used to precisely express more complex information flow properties needed for realistic applications. Moreover, formalizing several requirements results in one or several policies encompassing all those requirements which are difficult to read and interpret on their own.

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).

Thus, we developed a graphical UML notation that is used to express understandable IF properties including declassification and trace properties. Combined with a single security policy for the entire modeled application, artifacts for IF code analysis and proof obligations for formal verification are generated automatically. Simple and intuitive modeling guidelines for this notation allow the modeler to express relevant IF properties that can still be read and understood by anyone familiar with UML without a background in IF theory.

Section II introduces the IFlow approach, section III outlines our modeling guidelines for IF properties, while section IV describes how such properties can be checked and verified. Section V gives an overview over related work, and section VI concludes.

II. IFlow Approach

Our approach enables the model-driven development of distributed, information flow-secure systems consisting of Android apps and Java web services [4]. As a first step, the developer creates an abstract UML model of the system. Components such as apps and services are modeled as UML classes, while their interactions are modeled with sequence diagrams. Further, components can also call "manual" methods which implementation is not part of the model. Such methods are meant to provide application specific algorithms (e.g., filtering database entries) or platform-specific functionality (e.g., writing to the file system) for the final application. This modeling process is supported by our own simple domain specific language as well as UML profiles that provide IFlow-specific UML stereotypes.

Using automatic model transformations, a code skeleton is generated from the abstract system model that can be both checked for information flow violations using static code analysis as well as deployed on real Android devices and web servers running the Play framework. Manual methods that lack a functional implementation are implemented by hand and checked individually.

A formal model based on an abstract state machine and algebraic specifications is also generated automatically. This model is then used to prove desired information flow properties that could not be checked using static code analysis.

An example for a simple IF-secure application designed with IFlow is "Travel Planner"[1]. This application allows a user

[1]A version of the Travel Planner application model can be found at http://www.isse.de/iflow

to query a travel agency web service for suitable flight offers via a travel planner app, and then book a flight directly with an airline service using private credit card details stored in a credit card manager app. The following section shows how IF properties like "user's credit card details are never leaked to the travel agency" can be intuitively expressed in UML and transformed into checks for a language-based information flow control tool as well as proof obligations for an interactive theorem prover.

III. Designing information flow properties

A. Security policy

The most basic notion needed to instantiate an information flow property is a security policy. Such policy defines several security domains that data locations or actions can be mapped to. The interference relation between those domains restricts the allowed information flow within the application. We model such domains as UML activity nodes and specify the interference relation between domains with UML control flows that are implicitly transitive. It is possible to model an intransitive policy by introducing intransitive control flows and annotating them with the ≪`intransitive`≫ stereotype. Security domains can then be applied to modeled attributes (such as `ccd` and `purchaseHistory` in Fig. 2(a)), exchanged messages or individual actions such as method invocations. Fig. 1(a) illustrates a security policy which, e.g., forbids flows from data locations labeled with the {*User*} security domain to those labeled with {*User,TravelAgency,Airline*}, and only allows flows from {*User*} to {*User,Airline*} via the intransitive domain {*User*}→{*User,Airline*}. Since the security policy is only needed for formal verification, the user of an IFlow application is not required to look at it in order to understand the guaranteed IF properties.

B. Transitive noninterference properties

A security policy and a mapping of security domains to system actions or locations are already sufficient to describe simple noninterference properties. However, in IFlow we strive for intuitively understandable application models and go beyond security annotations by focusing on what may or may not happen to sensitive information that is relevant to the user.

We therefore define modeling guidelines to specify understandable, individual noninterference properties using UML activity diagrams. Such diagrams reference the information locations (such as system components, component attributes or platform specific sources and sinks) that pertain to the desired property and allow the modeler to define the allowed/forbidden information flow in a flexible, fine-granular way. In this section, we focus on transitive noninterference properties which state that secret information can never become more public by interfering with public sinks.

Fig. 1(b) shows the information flow property "Travel Agency never learns the user's credit card data" as modeled in IFlow. Information sources, i.e., locations that generate or hold information, are modeled as UML send signal actions. Information sinks, i.e., locations that receive information, are modeled as UML accept event actions. We support such locations to be system components or their attributes. A control flow edge with the applied stereotype ≪`noFlow`≫ between a source and a sink specifies that there should be no information flow from the source to the sink. Thus, `CreditCardCenter.ccd` in Fig. 1(b) is an information source and refers to the `ccd` attribute of the component `CreditCardCenter` holding the user's credit card data, while `TravelAgency` is modeled as an information sink where this data may never flow to. By default, all information flows between modeled components are allowed.

If the property forbids all information flows from a specific source to all other sinks, those sinks can be summarized with a placeholder annotated with the ≪`otherSinks`≫ stereotype (or vice versa with the ≪`otherSources`≫ stereotype). The placeholder represents every information location in the system other than those explicitly identified in the property as source or sink. In order to express exceptions to such strong properties, it is possible to introduce explicitly allowed flows within the same property by annotating a control flow edge with the ≪`allowedFlow`≫ stereotype. As such, the property shown in Fig. 1(c) forbids any flow from the `ccd` attribute of the `CreditCardCenter` component to any location within the system except to the `TravelPlanner` component[2].

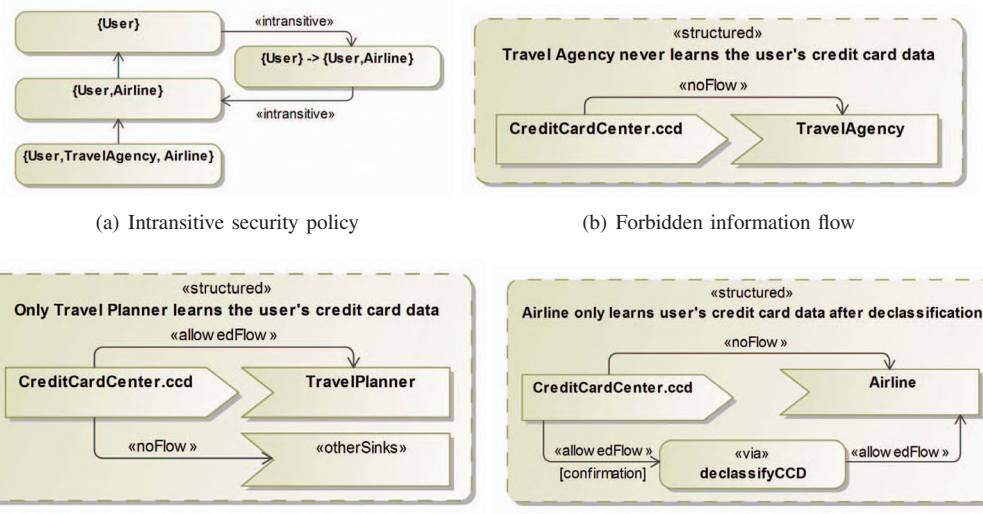
C. Declassification properties

Declassification is often necessary when information should be released, i.e., when transitive noninterference policies are too restrictive and need to be relaxed. In some cases, secret information must become more public via specific routines in order to ensure correct functionality. E.g., a password check routine must disclose information about the correct password in order to return a result.

IFlow allows the modeling of declassification properties by introducing exceptional, allowed flows from an information source to a sink via a specific declassification method. Fig. 1(d) shows such a declassification property that states that the user's credit card data may only flow to the Airline after being declassified via the declassification method `declassifyCCD(...)`. The direct information flow from source to sink is forbidden as indicated by the control flow between them with the ≪`noFlow`≫ stereotype. However, indirect IF via `declassifyCCD` is allowed as specified by control flows annotated with ≪`allowedFlow`≫ to and from an activity annotated with the ≪`via`≫ stereotype. This node can contain the name of a method which output parameter will be declassified.

We are thus able to model the *What*-dimension of declassification as identified by Sabelfeld and Sands [5] by explicitly specifying the source of information (but not yet its form) that can be declassified. The *Where*-dimension of declassification is modeled by using explicit declassification statements that

[2]This property subsumes the property shown in Fig. 1(b), and is violated if the property in Fig. 1(d) holds.



(a) Intransitive security policy

(b) Forbidden information flow

(c) Forbidden information flow with exceptions

(d) Allowed information flow after user confirmation and declassification

Fig. 1. Modeling information flow properties

are localized in UML diagrams specifying the behavior of the system.

D. Trace properties

In some cases, an information flow should only be allowed if a certain action has taken place on the trace of actions leading to disclosure of this information. In IFlow we allow the modeler to specify that an information flow is only allowed after the user's explicit confirmation. Fig. 1(d) includes a UML guard `confirmation` (a *guarding action*) applied to the allowed flow to `declassifyCCD` (a *guarded action*) to indicate that the user must have confirmed the declassification prior to releasing the information to the Airline, thus specifying the *When*-dimension of declassification.

E. Platform-specific properties

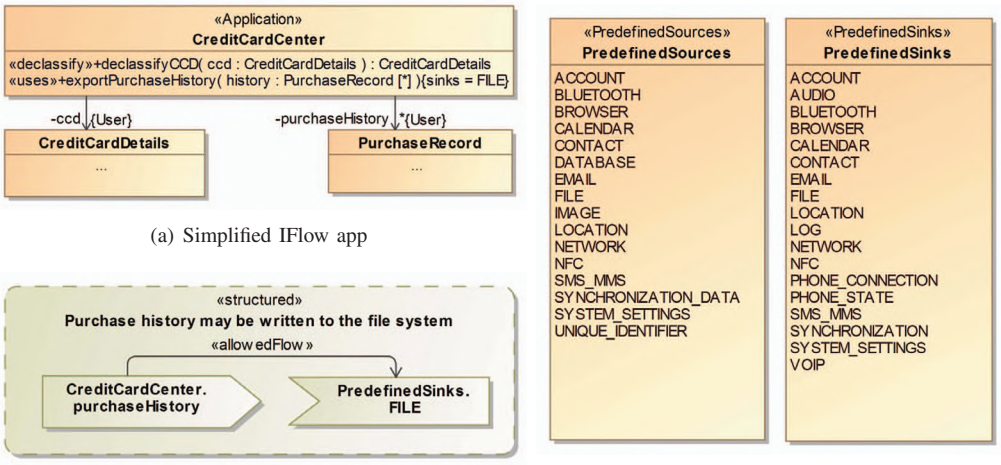
On any modern smartphone, location sinks and sources include sensors (GPS, camera etc.), file systems (internal storage, SD card etc.) and more. Naturally, we provide a way for the modeler to refer to such platform-specific locations in IF properties. The IFlow UML profile includes two predefined, application-independent enumerations named `PredefinedSource` and `PredefinedSink` as shown in Fig.2(c) which list categories of sources and sinks pertaining to phone sensors and data locations (e.g., `LOCATION`, `FILE` etc.). Any of those locations can be used as source or sink in an IF property. Additionally, manual methods that access those locations must be annotated with the ≪`uses`≫ stereotype, listing the categories of locations it reads or writes as stereotype tags. This mechanism is illustrated in Fig. 2(a) that specifies the manual method `exportPurchaseHistory` of the CreditCardCenter app which implementation may access the device storage in order to export user's purchase history. For any IFlow application, default IF properties are generated automatically which state that no information must leak to or from platform-specific locations. Those properties can be weakened by explicitly modeling allowed flows to or from platform-specific locations; e.g., Fig. 2(b) explicitly allows the user's purchase history to be saved to the file system. However, network access via modeled component communication is automatically allowed and can only be revoked by explicitly forbidding IF to or from `NETWORK`.

IV. Verifying and Checking Information Flow Properties

A. Formal verification and semantics

To check whether an IFlow application is secure w.r.t. the modeled properties we use both language- and model-based-approaches to IFC. While transitive noninterference can be checked automatically on the code level, we use formal verification to guarantee intransitive and trace properties. To this end, we automatically generate a formal model based on algebraic specifications and an abstract state machine from the application model. The state is an algebra that is modified by the rules of the ASM. We use the interactive theorem prover KIV [6], but the results of the verification do not depend on the chosen tool. An in-depth discussion of the formal model, formal framework and verification techniques used in IFlow can be found in [7].

1) Guaranteeing noninterference properties: Noninterference properties are given formal semantics by translating them to interferences between security domains in a formal framework based on intransitive noninterference. We use Rushby's access control instance of model-based intransitive noninterference [3] which allows us to use the concept of *locations*. Locations are accessed by *actions* which coarsely correspond to message handling methods in the code. The modeled security policy (see Sect. III-A) is used as a basis for the assignment



(a) Simplified IFlow app

(b) Allowed information flow to a platform-specific sink

(c) Platform-specific sources and sinks

Fig. 2. Allowing access to predefined, platform-specific sources and sinks

of security domains to actions. We apply this approach to verify declassification properties specifying partial or controlled information release. As such, we prove that user's credit card data is leaked to the airline component (cf. declassification property in Fig. 1(d)) only via the `declassifyCCD(...)` method if it is the only method annotated with the intransitive security domain $\{User\} \rightarrow \{User, TravelAgency\}$ from modeled security policy (cf. Fig. 1(a)). Transitive noninterference properties can also be verified using the same formal model and framework in case the code check fails due to false positives.

2) Guaranteeing trace properties: To prove that certain actions only occur in a specific order given any action trace allowed by the modeled application, we automatically generate counters that are incremented by ASM rules when certain actions like user confirmation occur. Given a trace property such as one depicted in Fig. 1(d), we generate a proof obligation which states that the counter associated with the guarded action (e.g., declassification via `declassifyCCD(...)`) must be smaller than the counter associated with the guarding action (e.g., `confirmation` via user prompt) for every possible trace of actions. This obligation requires that every guarded action may only occur after the guarding action.

B. Automatic code analysis

To guarantee transitive noninterference properties automatically we use the information flow control tool for Java applications called JOANA [8]. However, since JOANA currently doesn't support the analysis of distributed applications that run on different platforms, we have implemented a mechanism that allows us to use the tool and still be able to deploy the modeled application as a distributed system of mobile apps and web services.

The abstract IFlow system model is transformed into the code skeleton of a single Java-application which represents IFlow components as Java classes and simulates component communication via messages as direct calls to message handling methods of the receiving component. The generated code is implemented against the interface of an "IF framework" provided by IFlow. We provide one platform-independent and two platform-specific implementations of this framework. The platform-independent version provides an IF simulation of the platform-specific functionality used by the IFlow application, which allows us to analyze the generated Java application with JOANA. If the check succeeds, the developer can exchange the platform-independent version with the platform-specific versions of the framework. This allows for the deployment of the application as Android apps and Play web services without introducing any illegal information flows that the automatic code analysis did not account for [4].

1) Guaranteeing transitive noninterference properties: We derive and combine requirements about allowed and forbidden information flows from the modeled transitive noninterference properties and calculate an IF table which specifies for all combinations of sources and sinks whether IF is allowed between them (see Tab. I for a simplified table calculated from properties depicted in Fig. 1(b), 1(c) and 2(b)). Properties that conflict with the modeled security policy or each other are reported to the modeler. By default, all flows between modeled components and component attributes are allowed, whereas flows to and from platform-specific sources and sinks are forbidden. For each "forbidden" entry in the IF table we use JOANA to check whether the generated code leaks information from source to sink.

Sources↓ , *Sinks*→	`TP`	`TA`	`FILE`	...
`ccd`	allowed	forbidden	forbidden	...
`purchaseHistory`	forbidden	forbidden	allowed	...
...	...	...	...	...

TABLE I
GENERATED INFORMATION FLOW TABLE

In order to analyze a Java application with JOANA, one has to annotate nodes in the program dependency graph (PDG) of

the application as sources and sinks of information. We use the JOANA *IFC Console* tool which allows to create PDGs and annotate Java class attributes and method parameters directly by internally mapping them to relevant PDG nodes. Using the calculated IF table, we map IFlow sources and sinks to Java methods and attributes of the generated code as follows:

- A source or sink referencing an IFlow component attribute (such as `CreditCardCenter.ccd` in Fig. 1(b)) is mapped directly to the corresponding field of the Java class representing this component.
- A sink referencing an IFlow component (such as `TravelAgency` in Fig. 1(b)) is mapped to all fields and the input parameters of all message handling methods of the Java class representing this component.
- A source referencing an IFlow component is mapped to all of its fields, as well as the input parameters of the message handling methods called by the Java class representing this component.
- A predefined sink (such as `PredefinedSinks.FILE` in Fig. 2(b)) is mapped to the input parameters of methods accessing this sink (such as `exportPurchaseHistory(...)` in Fig. 2(a)).
- A predefined source is mapped to the return parameters of methods accessing those sources.
- Sinks and sources annotated with the ≪`otherSinks`≫ and ≪`otherSources`≫ stereotypes are automatically expanded to relevant IFlow components and attributes, and mapped to Java class fields and method parameters as outlined above.

2) Guaranteeing platform specific properties: Platform-specific, predefined sources and sinks may be accessed via methods which bodies are implemented manually (such as `exportPurchaseHistory(...)` in Fig. 2(a)). In order to assure that the manual implementation indeed only accesses the specified sources and sinks, it needs to be analyzed separately. We check whether this manual implementation only invokes such API methods that correspond to sources or sinks explicitly allowed by the ≪`uses`≫ stereotype. To do this, we utilize the results of the SuSi tool [9] which maps such sources and sinks to Android API methods that can be used to access them.

V. Related Work

We are not aware of attempts similar to ours that tackle the challenge of designing rigorous, yet intuitive guidelines for modeling IF properties.

Jürjens [10] proposes a model-driven approach that focuses on developing cryptographically secure applications, but also allows for basic information flow annotations. IF properties are not modeled explicitly, nor is code level security taken into account.

Seehusen [11] presents a model-driven approach for developing systems that are secure wrt. information flow. However, he does not focus on designing intuitive IF properties, no code is generated, and consideration of IF properties on the code level is left as future work.

Heldal [12] introduces a UML profile that allows to annotate UML elements with Jif labels. However, such annotations are not at all intuitive [13], the entire model needs to be considered, and no automatic Jif code generation is supported.

VI. Conclusion and Outlook

IFlow is a model-driven approach for developing information flow-secure distributed applications consisting of mobile apps and services. In this paper, we have presented how intuitive information flow properties including declassification and trace properties can be modeled using UML and verified using a combination of automatic code analysis and formal verification.

We will continue the work on our modeling language for information flow properties. Our plan is to support dynamic properties which would allow users to specify their desired level of privacy. Further, we will add the ability to model the behavior of declassification methods in order to be able to specify and verify more fine-grained declassification properties that allow for partial information release.

References

[1] 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.

[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638.

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

[4] 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, 2013.

[5] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, june 2005, pp. 255 – 269.

[6] 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.

[7] K. Stenzel, K. Katkalov, M. Borek, and W. Reif, "A model-driven approach to noninterference," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 5, no. 3, pp. 30–43, September 2014.

[8] 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)*, ser. Lecture Notes in Informatics (LNI) 215. Springer Berlin, Feb. 2013.

[9] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," *Proceedings of the Network and Distributed System Security Symposium*, 2014.

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

[11] 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.

[12] R. Heldal, S. Schlager, and J. Bende, "Supporting confidentiality in UML: A profile for the decentralized label model," in *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*, Munich, Germany, 2004, pp. 56–70, TU Munich Technical Report TUM-I0415.

[13] 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 *Data Privacy Management and Autonomous Spontaneous Security*. Springer LNCS 7732, 2013.