Formalizing Information Flow Control in a Model-Driven Approach

Kurt Stenzel, Kuzman Katkalov, Marian Borek, Wolfgang Reif

Formalizing Information Flow Control in a Model-Driven Approach[*]

Kurt Stenzel, Kuzman Katkalov, Marian Borek, and Wolfgang Reif

Institute for Software & Systems Engineering,
Augsburg University, Germany

Abstract. Information flow control is a promising formal technique to guarantee the privacy and desired release of our data in an always connected world. However, it is not easy to apply in practice. IFlow is a model-driven approach that supports the development of distributed systems with information flow control. A system is modeled with UML and automatically transformed into a formal specification as well as Java code. This paper shows how the this specification is generated and presents several advantages of a model-driven approach for information flow control.

1 Introduction

Smartphones are useful digital assistants that simplify our life. However, they store larger and larger amounts of private data. This data can and should be accessed by apps, but we would like to have control over the when and how, and what happens with the data afterwards. Information flow control (IFC) is an area of research to achieve this. IFC (e.g., [3, 7, 6]) is stronger than access control, and 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. IFlow is a model-driven approach designed specifically for IFC. The overall approach is fully implemented and explained in [4]. This paper briefly introduces an example (Sect. 2), and then describes the advantages of the model-driven approach for the formal specification: formal proofs become easier, and specification errors are avoided (Sect. 3). Sect. 4 concludes.

2 An Example Application

The *travel planner app* is a typical distributed application consisting of a smartphone app and web services. 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

[*] 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.

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. 1 shows this behavior as a sequence diagram.

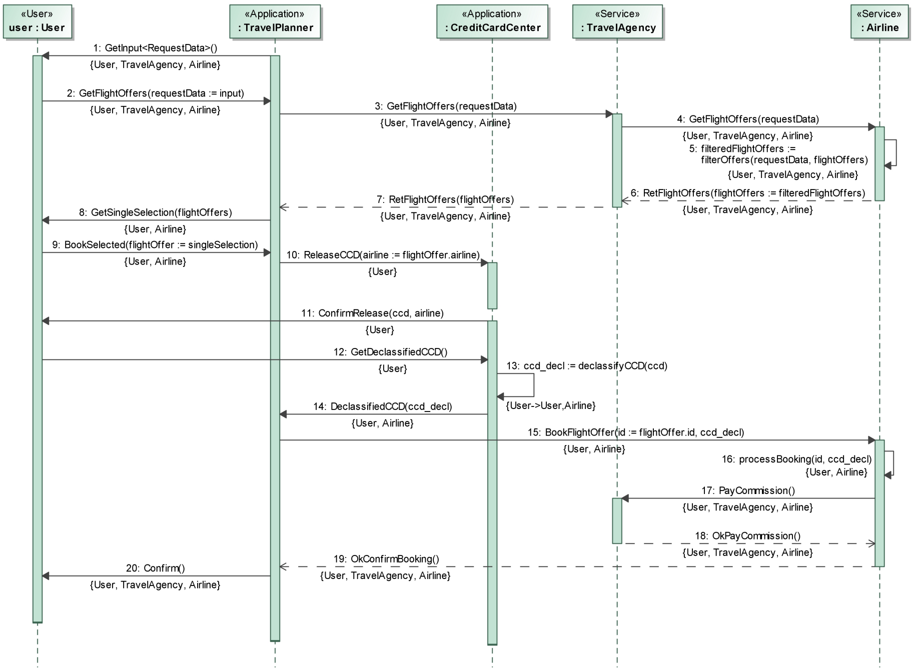
«User»
user : User
«Application»
: TravelPlanner
«Application»
: CreditCardCenter
«Service»
: TravelAgency
«Service»
: Airline
1: GetInput<RequestData>()
{User, TravelAgency, Airline}
2: GetFlightOffers(requestData := input)
{User, TravelAgency, Airline}
3: GetFlightOffers(requestData)
{User, TravelAgency, Airline}
4: GetFlightOffers(requestData)
{User, TravelAgency, Airline}
5: filteredFlightOffers :=
filterOffers(requestData, flightOffers)
{User, TravelAgency, Airline}
6: RetFlightOffers(flightOffers := filteredFlightOffers)
{User, TravelAgency, Airline}
7: RetFlightOffers(flightOffers)
{User, TravelAgency, Airline}
8: GetSingleSelection(flightOffers)
{User, Airline}
9: BookSelected(flightOffer := singleSelection)
{User, Airline}
10: ReleaseCCD(airline := flightOffer.airline)
{User}
11: ConfirmRelease(ccd, airline)
{User}
12: GetDeclassifiedCCD()
{User}
13: ccd_decl := declassifyCCD(ccd)
{User->User,Airline}
14: DeclassifiedCCD(ccd_decl)
{User, Airline}
15: BookFlightOffer(id := flightOffer.id, ccd_decl)
{User, Airline}
16: processBooking(id, ccd_decl)
{User, Airline}
17: PayCommission()
{User, TravelAgency, Airline}
18: OkPayCommission()
{User, TravelAgency, Airline}
19: OkConfirmBooking()
{User, TravelAgency, Airline}
20: Confirm()
{User, TravelAgency, Airline}

Fig. 1. Sequence diagram for the travel planner app.

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}. They are used in the formal information flow framework.

Together with a class diagram (not shown), and a security policy (Fig. 2) the sequence diagram contains enough information and is precise enough to generate a formal specification as well as Java code.[1] 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.

[1] See our web page `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/iflow/` for the full model.

3 The Formal Model

As the formal framework we chose Rushby's intransitive noninterference [7] which is an extension of Goguen's and Meseguer's transitive noninterference [3], and 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.

It works like this: Actions a modify the state with a *step* function $\mathit{step}(s, a)$ 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 function $\mathit{output}(s, d)$. 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. $\mathit{output}(s, d)$ defines what an attacker with security domain d 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.

For a given IFlow UML model an instance of Rushby's framework is generated for the theorem prover KIV [5, 1] that is based on Abstract State Machines (ASMs [2]) and algebraic specifications.

3.1 Unwinding theorem

Rushby [7] proves an unwinding theorem that implies security if four unwinding conditions hold. The second condition was later weakened by van der Meyden [8] (we show van der Meyden's version):

- RM1: $s_1 \approx_d s_2 \rightarrow \mathit{output}(s_1, d) = \mathit{output}(s_2, d)$
  If two states look alike to an attacker with domain d ($\approx_d$) then the attacker's output is the same in both states.
- RM2: $s_1 \approx_{\mathit{dom}(a)} s_2 \wedge s_1(l) = s_2(l) \wedge l \in \mathit{alter}(\mathit{dom}(a))$
  $\rightarrow \mathit{step}(s_1, a)(l) = \mathit{step}(s_2, a)(l)$
  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 [8] for a discussion.)
- RM3: $\mathit{step}(s, a)(l) \neq s(l) \rightarrow l \in \mathit{alter}(\mathit{dom}(a))$
  If the execution of an action modifies a location then this location is contained in the *alter* set for the action's domain.
- AOI: $\mathit{alter}(d_1) \cap \mathit{observe}(d_2) \neq \emptyset \rightarrow d_1 \rightsquigarrow d_2$
  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. 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 $\leadsto$.

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

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

(which is trivial) already implies that a system is secure. This shows a great benefit of IFlow's model driven approach that cannot be achieved otherwise.

3.2 Automatic 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. 2 shows the security policy for the example. An edge denotes a (direct) interference, i.e., {User, TravelAgency, Airline} $\leadsto$ {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. 2: 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.

When a declassification domain is used (message 13) it must be contained in its own action to avoid undesired interferences. A program counter is introduced automatically to ensure that actions/messages 12, 13, and 14 are executed in the correct order. There is another complication. Generating the formal model results in an insecure system, i.e., the only remaining unwinding condition AOI does not hold. The problem are messages 15–17 in Fig. 1: 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. This may or may not be seen as a problem, but is independent from the desired property. 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. 2 shows how this works.

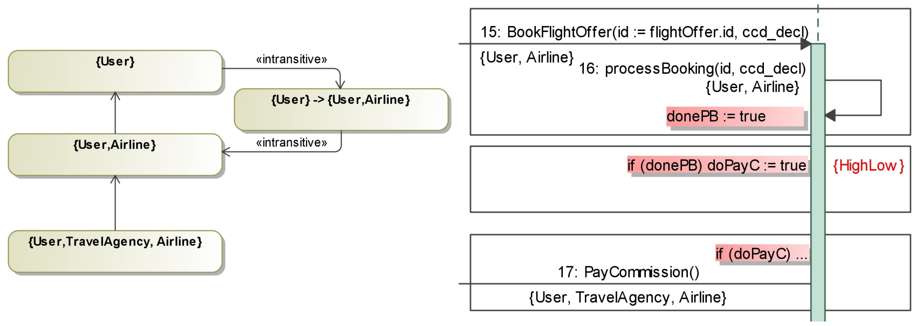
{User}
«intransitive»
{User} -> {User,Airline}
«intransitive»
{User,Airline}
{User,TravelAgency, Airline}
15: BookFlightOffer(id := flightOffer.id, ccd_decl)
{User, Airline}
16: processBooking(id, ccd_decl)
{User, Airline}
donePB := true
if (donePB) doPayC := true
{HighLow}
if (doPayC) ...
17: PayCommission()
{User, TravelAgency, Airline}

Fig. 2. Security policy and new actions.

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. 2).

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 will still be detected. Assume that the PayCommission message includes the credit card details as parameter (PayCommission(ccd_decl)). Then the third action in Fig. 2 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 {User, TravelAgency, Airline} interferes {User, Airline} directly which is forbidden by the policy. And the {HighLow} domain obviously does not leak the credit card details either. So 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. A user specifying this by hand could easily make mistakes. This again shows the benefits of a model-driven approach.

4 Conclusion

Information flow control is a promising technique, but difficult to use in practice. IFlow is a model-driven approach that supports the development of distributed applications with guaranteed and intuitive information flow properties. The resulting UML specification is automatically transformed into a formal model based on intransitive noninterference where IF properties can be proved, and into Java code with the same IF properties. To the best of our knowledge IFlow is the only work that uses a model-driven approach for IFC (a broader comparison with related work must be omitted due to lack of space). We showed that the model-driven approach has additional benefits for IFC: proofs become simpler because functions like *observe* and *alter* can be computed, and specification errors can be avoided by the automatic introduction of declassifications with guaranteed properties.

References

1. 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.
2. E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
3. J.A. Goguen and J. Meseguer. Security Policy and Security Models. In *Symposium on Security and Privacy*. IEEE, 1982.
4. K. Katkalov, K. Stenzel, M. Borek, and W. Reif. Model-driven development of information flow-secure systems with IFlow. In *Proceedings of 5th ASE/IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT)*. IEEE Press, 2013.
5. KIV homepage. `http://www.informatik.uni-augsburg.de/swt/kiv`.
6. H. Mantel. Possibilistic definitions of security - an assembly kit. In *IEEE Computer Security Foundations Workshop*. IEEE Press, 2000.
7. J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.
8. R. van der Meyden. What, indeed, is intransitive noninterference? In *Proc. European Symposium on Research in Computer Security*. Springer LNCS 4734, 2007.