# Code abstractions for automatic information flow control in a model-driven approach

**Kuzman Katkalov, Kurt Stenzel, Wolfgang Reif**

**Angaben zur Veröffentlichung / Publication details:**

Katkalov, Kuzman, Kurt Stenzel, and Wolfgang Reif. 2017. “Code abstractions for automatic information flow control in a model-driven approach.” In *Security, Privacy, and Anonymity in Computation, Communication, and Storage: SpaCCS 2017 International Workshops, Guangzhou, China, December 12-15, 2017, proceedings*, edited by Guojun Wang, Mohammed Atiquzzaman, Zheng Yan, and Kim-Kwang Raymond Choo, 209–18. Cham: Springer. https://doi.org/10.1007/978-3-319-72395-2_20.

# Code Abstractions for Automatic Information Flow Control in a Model-Driven Approach

Kuzman Katkalov(✉), Kurt Stenzel, and Wolfgang Reif

Department of Software Engineering and Programming Languages,
University of Augsburg, Augsburg, Germany
{kuzman.katkalov,stenzel,reif}@informatik.uni-augsburg.de

**Abstract.** Automatic information flow control (IFC) can be used to guarantee the absence of information leaks in security-critical applications. However, IFC of real-world, complex, distributed systems is challenging. In this paper, we show how a model-driven approach for development of such applications consisting of mobile apps and web services can help solve those challenges using automatic code abstractions.

**Keywords:** Information flow · IFC · Model-driven development
Security by design · Privacy by design

## 1 Introduction

In our time of connected and ubiquitous mobile devices that aggregate and share our personal information, privacy seems hard to come by. Leaks of sensitive data such as personal photos or payment information due to insecure app implementations are abound [2]. Meanwhile, traditional security mechanisms such as the Android permission system are often not sufficient to prevent them [3].

Information flow control (IFC) is the preferable technique to guarantee that a system does not leak sensitive information. Our model-driven approach called *IFlow* leverages IFC to enable the development of information flow (IF)-secure, distributed applications consisting of mobile Android apps and Java web services (see Fig. 1) [7]. Using the new modeling language MODELFLOW based on UML, the developer can specify the structure, behavior, and IF properties of their application (1) [8]. From this abstract model, a code skeleton is generated automatically (2). It implements the IFlow formal application model based on abstract state machines (4) [15,16] as a Java program, and serves as the basis for the final, distributed application (3). In addition, the code skeleton is used for automatic IFC with JOANA [5], a leading framework for flow-sensitive, context-sensitive, object-sensitive, and lock-sensitive information flow analysis of Java bytecode using program dependence graphs (PDGs).

However, static IFC of distributed, heterogeneous (w.r.t. the deployment platform), and complex applications is challenging. JOANA supports monolithic Java applications with limited size and complexity [6], whereas IFlow applications are distributed, and can consist of several mobile apps and services running on the Android and Java web service platforms.

Section 2 presents a solution to this challenge that leverages the unique advantages of a model-driven approach. It proposes a systems architecture that enables the IF analysis of complex, distributed applications, allowing for information flow-preserving extensions via manually implemented or automatically generated code. Section 3 outlines related work and concludes.

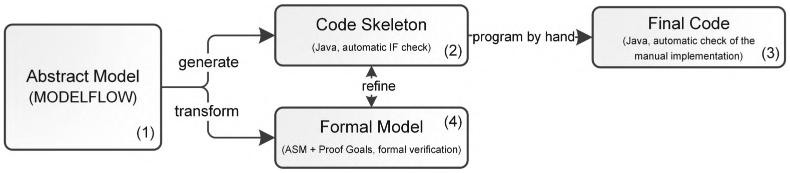

Abstract Model
(MODELFLOW)
(1)
generate
transform
Code Skeleton
(Java, automatic IF check)
(2)
refine
Formal Model
(ASM + Proof Goals, formal verification)
(4)
program by hand
Final Code
(Java, automatic check of the manual implementation)
(3)

**Fig. 1.** The IFlow approach

## 2 Generating Java Code for Automatic IFC

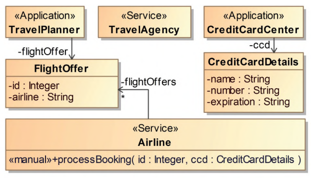
Consider one of our case studies, a mobile travel planning application where the travel agency may not learn the user's credit card data. Using IFlow, it is modeled as a set of *application components* representing mobile apps and web services as MODELFLOW classes (see Fig. 2(a))[1]. In the automatically generated code skeleton, the components are implemented as individual Java classes, and interact with *application modules* as well as an additional Java library provided by IFlow (see Fig. 2(c)) which wraps Android and Java web service APIs to provide platform specific functionality. Application component behavior is modeled explicitly by the developer using sequence diagrams (see Fig. 2(b)), and is translated directly into simple, sequential Java code. MODELFLOW constructs such as variable declarations, assignments, or method calls are mapped to equivalent Java language statements that honor MODELFLOW's copy semantics, while message handling routines are implemented as Java methods.

Application components may interact with *application modules* such as:

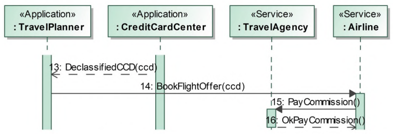
- *manually implemented methods*, e.g., for accessing platform specific information sources or sinks such as the SD card of a mobile device,
- *manually implemented* and *predefined graphical user interfaces*, or
- *predefined operations* such as encryption routines.

Most application modules are underspecified and translated into abstract *code stubs* as part of the code skeleton. A stub implements an information flow over-approximating abstraction of the functional module implementation. This enables automatic IFC of the code skeleton as a simple, sequential, and monolithic Java console application. To this end, both the application components and the module stubs use a version of the IFlow Java library that emulates the information flows of platform specific APIs. The code skeleton can be extended

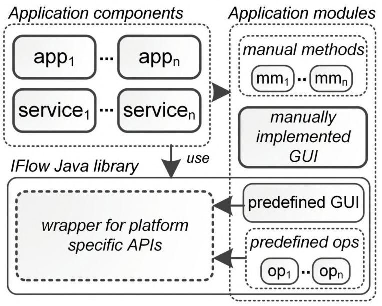
[1] See http://isse.de/iflow for models and code of our case studies.



(a) Travel Planner application model (static view, excerpt)



(b) Travel Planner application model (dynamic view, excerpt)



(c) IFlow application architecture

**Fig. 2.** An IFlow application

to a final, runnable application by replacing the module stubs as well as the IFlow Java library with their functional, platform specific implementations.

However, in order to guarantee that the final application inherits the information flow properties of the code skeleton (i.e., it is an *IF preserving code refinement*), the module stubs must closely model their functional counterparts w.r.t. their information flows.

## 2.1 General Abstraction Techniques

**Information dependencies between variables.** One of the abstraction techniques used in IFlow is the emulation of information flow between several variables, e.g., from input to output parameters (or an optional persistent state) of a module. The goal is a PDG of the code skeleton where all fields of the output object(s) of the module depend on all fields of its input parameter(s). This can represent the worst-case information flow scenario for the module, if the final implementation only accesses a subset of such fields of an input parameter to calculate the output. Provided that the module does not access any further sources or sinks of information, this technique guarantees that its final implementation will never leak more information than its stub.

Our model-driven approach allows us to automatically implement such a dependency by generating the additional methods `extractIF` and `propagateIF` for every modeled MODELFLOW data type. `extractIF` recursively accesses every field of a MODELFLOW object, and returns an integer value that depends on the values of such fields. In turn, `propagateIF` propagates this value recursively to all fields of another MODELFLOW object.

In our pseudo code notation for module stub implementation, we write $out = (in_1...in_n)$ if all the fields of the object $out$ depend on all fields of the objects $in_1...in_n$. The persistent state of a module is summarized as the variable $sl$.

**Component communication.** IFlow applications are distributed systems consisting of one or several Android apps and Java web services. Currently, JOANA does not support the analysis of such distributed systems. However, even if the code of the individual apps and services is merged into a single application and analyzed, the results of the static information flow analysis would be too imprecise.

The reason for this imprecision is the way such apps and web services handle incoming messages. Both implement a general message handling routine which checks the type of an incoming message before executing the appropriate, application-, and message-specific handling method. This presents a challenge for static code analyzers, since they cannot determine the type of the incoming message. Consider the pseudo code snippet in Listing 1.1 that depicts the simplified behavior of the application components from Fig. 2(b). Even though sensitive credit card information stored in `ccd` never flows to the public web service *TravelAgency*, the static analysis will not be able to differentiate between the branches in ll.9-10 and thus deem the application insecure.

**Listing 1.1.** Generic message handling methods as a challenge for static code analyzers

```
1   class TravelPlanner{
        void declassifiedCCD(var ccd){
3           msg.type = 1;
            msg.content = ccd;
5           Airline.handleIncomingMsg(msg); }}
    class Airline{
7       void handleIncomingMsg(var msg){
            if(msg.type==1) bookFlightOffer(msg);
9           else getFlightOffers(msg); }
        void bookFlightOffer(var msg){ TravelAgency.payCommision(); }
11      void getFlightOffers(var msg){ TravelAgency.retFlightOffers(msg); }}
```

Here, the model-driven approach offers another advantage: instead of using the generic message handling routines, the automatically generated application component code directly invokes the appropriate handling method for the modeled message, allowing for more precise information flow analysis (see Listing 1.2).

**Listing 1.2.** Component communication via specific message handling methods

```
class TravelPlanner{
    void declassifiedCCD(var ccd){
        msg.type = 1;
        msg.content = ccd;
        Airline.bookFlightOffer(msg); }}
```

In order to implement the modeled communication functionality in the final application, the code of the receiving component is replaced with an automatically generated proxy implementation (see Listing 1.3) which uses platform specific API to deliver the message as an Android intent or an HTTPS query.

**Listing 1.3.** Component communication via an automatically generated proxy

```
class Airline{
    void bookFlightOffer(var msg){ IFlowLib.sendMsg("Airline", msg); }
    void getFlightOffers(var msg){ IFlowLib.sendMsg("Airline", msg); }}
```

**User and attacker simulation.** IFlow applications may request and receive user input, and execute different sequences of actions based on that input. In IFlow, we assume that the user's decision is arbitrary, and simulate it in the code skeleton using a pseudo random number generator. In cases where the execution is branched over the user's input, this approach guarantees that both branches are taken into account by the static information flow analysis. In our pseudo code notation, the user's input is summarized as the variable $u$.

In addition, an attacker may try to communicate directly with a modeled app or web service without adhering to the modeled application behavior in order to extract sensitive data. E.g., a malicious app could prompt the *CreditCardCenter* app in Fig. 2(b) to disclose the user's credit card data via message *DeclassifiedCCD*. To prevent information leakage, IFlow apps may therefore only be invoked by other apps from the same modeled application that are cryptographically signed with the same private key. This is guaranteed by employing signature-level protection on Android permissions for IPC communication. In order to detect such information leakage due to direct queries of modeled web services, the code skeleton includes an additional component *Attacker* that is generated automatically from the application model.

*Attacker* implements a persistent state $sl$ which it uses to assemble new messages and store web service responses. Let $ws_1$`(var msg)`...$ws_n$`(var msg)` represent all message handling methods of modeled web service components, while $in_1$`(var in)`...$in_n$`(var in)` handle their response messages. The code skeleton emulates a random number of web service queries by an attacker in a random order as shown in Listing 1.4 by interweaving them with actions of legitimate application users. Using this approach, IFlow can detect potential information leakage by annotating the internal state $sl$ of *Attacker* as a public sink.

**Listing 1.4.** Simulating information flows due to an attacker

```
class Attacker {
    var sl;
    void init(){
        while(u){
            msg = (sl,u);
            if(u == 1) ws_1(msg);
            ...
            if(u == n) ws_n(msg);}}
    void in_1(var in){ sl = (sl,in); }
    ...
    void in_n(var in){ sl = (sl,in); }
}
```

## 2.2 Application Modules

This section describes application modules supported in IFlow, and defines the assumptions that are made about their functional implementations. Further, it shows their stub implementations based on those assumptions that allow for information flow analysis of an IFlow application with few false positives, and how those assumptions are guaranteed for the final implementation.

In the following, information flow dependencies within a module are expressed in the sense of the predecessor relation $\rightarrow^*$ in the PDG [5]. E.g., $in \rightarrow^* out$ for the parameters $in$ and $out$ of a method denotes that the value of $out$ (and all of its fields) depends on the value of $in$ (and all of its fields) due to paths existing in the PDG from all nodes representing $in$ to all nodes representing $out$. $M$ denotes all program locations which may hold information (such as variables or class attributes) that are accessed by the module, whereas $C$ denotes all such locations accessed by the modeled application components. $Q$ and $S$ denote sets of platform specific sources and sinks (e.g., SMS), while the input and output parameters of a module are written as sets $\{in_1..in_n\}$ and $\{out_1..out_m\}$. The code generation of the application components guarantees that they interact with the modules only by writing their inputs and reading their outputs. Unless stated otherwise, we make the following default assumptions for every module:

- $d_1$: $M \cap C = \{in_1..in_n\} \cup \{out_1..out_m\}$
  The module interacts with the application components only by reading and writing its own input and output parameters
- $d_2$: $M \cap Q = \emptyset \wedge M \cap S = \emptyset$
  The module does not access any platform specific sources or sinks.
- $d_3$: The module is stateless

**Manual methods.** IFlow allows the modeler to specify and use methods which behavior is implemented manually using Java. This enables the developer to flexibly implement and reuse complex, possibly platform specific functionality that is not provided by the IFlow Java library. Using stub implementations of such methods for IFC reduces the complexity of the resulting PDG, allowing for more effective IF analysis. Using MODELFLOW annotations [8], the modeler specifies the sets $Q_e \subseteq Q$ and $S_e \subseteq S$ of platform specific sources and sinks that the manual implementation of the method may access.

The stub of a manual method $mm$ with the input parameters $\{in_1...in_n\}$ and the return parameter $out$ as shown in Listing 1.5 assumes the following:

- $m_1$: $M \cap Q = Q_e \wedge M \cap S = S_e$
  The module only accesses allowed, platform specific sources and sinks.
- $m_2$: $\forall_{x \in \{1..n\}} in_x \rightarrow^* out$
  All input method parameters interfere the output parameter.

**Listing 1.5.** Module stub of a manual method *mm*

```
static var mm(var in_1 , .. , var in_n){
    return out = (in_1 ,.. ,in_n); }
```

Assumptions $m_1$, $d_3$, and $d_3$ are checked by statically analyzing the functional module implementation using the Soot framework [11], while $M$, $C$, $Q_e$ and $S_e$ are calculated automatically from the application model [8]. $Q$ and $S$ are provided by the SuSi tool [13] that calculates a list of Android sources and sinks using machine learning. Assumption $m_2$ denotes a worst case over-approximation of information flows between input and output parameters of the module, while the MODELFLOW copy semantics guarantee that the manual method implementation cannot introduce additional flows to objects passed to the method as input parameters. Further, the formalization of MODELFLOW information flow properties assumes the parameters of a manual method as platform specific sources and sinks if the method is allowed to access those [8].

**Predefined operations.** MODELFLOW provides a range of predefined operations that can be used by the modeled application, e.g., in order to access the GPS sensor data on a mobile device, or encrypt sensitive information. Most such operations are handled similarly to manual methods, both on the modeling level as well as w.r.t. their stub implementations. By providing such predefined functionality and stubs by default, we improve the precision of the IF analysis.

Cryptographic operations represent a special case of predefined operations, as they pose two challenges for static IFC: (1) their real, complex implementation results in larger PDGs, where (2) secret plaintext interferes public ciphertext. However, (2) does not capture the intuitive property of secure encryption: one cannot deduce the plaintext from the resulting ciphertext without knowing the secret encryption key. We therefore use a stub implementation of such operations as part of the information flow emulating version of the IFlow Java library as shown in Listing 1.6. Those stubs implement the *ideal* cryptographic functionality as proposed by Küsters in [9,10] (see also [4]).

**Listing 1.6.** Module stub of the predefined encryption functionality

```
class Decryptor {
    Map sl;
    var decrypt(var in_c){
        if(sl.containsKey(in_c)) return out_p = sl.get(in_c);
        else return out_p = random(); }
    Encryptor getEncryptor(){ return new Encryptor(sl); }
}
class Encryptor {
    Map sl;
    Encryptor(Map sl){ this.sl = sl; }
    var encrypt(var in_p){
        var rand = random();
        sl.put(in_p, rand);
        return out_c = rand; }
}
```

This stub implementation of the module assumes the following:

- $m_1$: The module is stateful and has the persistent state $sl \in M$
- $m_2$: $in_p \rightarrow^* sl \wedge in_p \not\rightarrow^* out_c \wedge (in_c = out_c \implies sl \rightarrow^* out_p)$

The plaintext input of *encrypt* influences the internal state of the module, but *not* its ciphertext output. Further, the result of the decryption of this output is interfered by the internal state, and, transitively, by the original plaintext

The functional implementation of the predefined encryption module is provided by the platform specific versions of the IFlow Java library. It assures that the assumptions $d_{1-2}$ and $m_1$ hold, while providing real encryption functionality as per [9,10] to establish cryptographic indistinguishability property for the application, which means i.a. that the private plaintext remains secret for a public observer of the corresponding ciphertext (assumption $m_2$).

**Predefined and manual graphical user interfaces.** MODELFLOW provides a predefined user interface (represented as the predefined component *User*) that allows the developer to model a number of predefined user interactions such as requesting user input or dialog confirmation. To implement a more complex or application specific user interface, the modeler can in addition use the GUI module (represented as a component with the *«GUI»* stereotype) which internal behavior is implemented manually using Java.

Graphical user interfaces present additional challenges to static IFC: (1) they employ callback methods that are triggered by the underlying platform, requiring its IF analysis, (2) on mobile devices, user interaction is not limited to GUI elements of the active app, and (3) modern mobile apps have sophisticated user interfaces that result in larger PDGs, making the IF analysis more expensive. IFlow therefore provides stub implementations of both predefined and manual user interfaces.

Listing 1.7 shows the stub of a predefined GUI, where every requested user interaction is represented as an automatically generated handling method $in_x$ that presents the contents of its input parameter to the user. *showUI* implements a stub of the predefined IFlow Java library method for displaying the user interface on the screen. Based on user action, it explicitly triggers the callback method $out_x$ which is the return message handling method of the caller app.

**Listing 1.7.** Module stub of the predefined user interface

```
class User {
    var in₁(var in₁){
        var callback = (var out) -> out₁(outₙ);
        showUI(in₁, callback);  }
    ...
    var inₙ(var inₙ){
        var callback = (var out) -> outₙ(outₙ);
        showUI(inₙ, callback); }
    var showUI(var in, var callback){
        var out = (in, u);
        if(u) callback(out); }
}
```

The stub makes the following assumptions about the real implementation:

- $m_1$: $\forall_{x \in \{1 \ldots n\}} in_x \rightarrow^* out_x \wedge u \rightarrow^* out_x$
  The module output is interfered by the corresponding input request as well as the user input.
- $m_2$: The user may provide input to the interface, or cancel the input request, returning to the beginning of the modeled application sequence

In the functional implementation of the module, only the *showUI* method is replaced with its platform specific version that uses Android Fragments to display the user interface. Its predefined, stateless implementation is guaranteed to only access the provided module and user input to provide the module output via the $out_x$ callback method (assumptions $d_{1-3}$ and $m_1$). User interaction via the hardware navigation buttons on the mobile device is propagated to the invoking app, prompting it to reset the current behavior sequence (assumption $m_2$).

The stub of a manual GUI follows a similar structure as Listing 1.7. However, it also introduces a persistent state and additional flows between all inputs, outputs, and the module state, making the IF analysis less precise than for the predefined GUI. Its manual implementation is checked statically like that of a manual method, with $Q_e = S_e = \emptyset$.

## 3 Conclusion and Related Work

Information flow analysis can be used to guarantee that an application does not leak sensitive user information. However, static IFC of complex, distributed, real-world systems is challenging. There are several model-driven approaches for developing IF-secure applications such as [1,14], however, they do not use IFC on the code level to automatically guarantee IF properties. Code abstractions for IFC are used in JOANA for native methods in the Java standard library, while its Android specific version JoDroid [12] generates an entry method which simulates the Android framework by invoking all callbacks of the analyzed app. [4,9,10] propose how automatic IFC can be used to provide cryptographic privacy guarantees by employing ideal crypto functionality.

We propose a modular systems architecture that enables automatic IFC of real-world applications consisting of Android apps and Java web services using code abstractions. We show how such abstractions can be generated automatically in a model-driven development approach, and be securely expanded to obtain a final, deployable implementation of the modeled application. Using our approach, we are able to successfully provide IF guarantees for distributed applications such as the Travel Planner using automatic IFC.

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

## References

1. Ben Said, N., Abdellatif, T., Bensalem, S., Bozga, M.: Model-driven information flow security for component-based systems. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) ETAPS 2014. LNCS, vol. 8415, pp. 1–20. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54848-2_1
2. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, p. 21. USENIX Association (2011)

3. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 627–638. ACM (2011)
4. Graf, J., Hecker, M., Mohr, M., Snelting, G.: Checking applications using security APIs with JOANA. In: 8th International Workshop on Analysis of Security APIs, July 2015
5. Hammer, C.: Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs. Ph.D. thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3
6. Hammer, C.: Experiences with PDG-based IFC. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 44–60. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11747-3_4
7. Katkalov, K., Stenzel, K., Borek, M., Reif, W.: Model-driven development of information flow-secure systems with IFlow. ASE Sci. J. **2**(2), 65–82 (2013)
8. Katkalov, K., Stenzel, K., Borek, M., Reif, W.: Modeling information flow properties with UML. In: 2015 7th International Conference on New Technologies, Mobility and Security (NTMS). IEEE Conference Publications (2015). https://doi.org/10.1109/NTMS.2015.7266507
9. Küsters, R., Scapin, E., Truderung, T., Graf, J.: Extending and applying a framework for the cryptographic verification of Java programs. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 220–239. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_12
10. Küsters, R., Truderung, T., Graf, J.: A framework for the cryptographic verification of java-like programs. In: Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium, CSF 2012, pp. 198–212. IEEE Computer Society, Washington, DC (2012)
11. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop, Galveston Island, TX, October 2011
12. Mohr, M., Graf, J., Hecker, M.: JoDroid: adding android support to a static information flow control tool. In: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.–18. März 2015. CEUR Workshop Proceedings, vol. 1337, pp. 140–145. CEUR-WS.org (2015)
13. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing android sources and sinks. In: NDSS (2014)
14. Seehusen, F.: Model-driven security: exemplified for information flow properties and policies. Ph.D. thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, January 2009
15. Stenzel, K., Katkalov, K., Borek, M., Reif, W.: Formalizing information flow control in a model-driven approach. In: Linawati, Mahendra, M.S., Neuhold, E.J., Tjoa, A.M., You, I. (eds.) ICT-EurAsia 2014. LNCS, vol. 8407, pp. 456–461. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55032-4_46
16. Stenzel, K., Katkalov, K., Borek, M., Reif, W.: Declassification of information with complex filter functions. In: Proceedings of the 2nd International Conference on Information Systems Security and Privacy, pp. 490–497 (2016)