Towards a Common Specification Language for Information-Flow Security in RS^3 and Beyond: RIFL 1.0 – The Language

Technical Report TUD-CS-2014-0115

Sarah Ereth,
Heiko Mantel,
Matthias Perner

TECHNISCHE
UNIVERSITÄT
DARMSTADT

MAIS
Modeling and Analysis
of Information Systems

Towards a Common Specification Language for Information-Flow Security in RS^3 and Beyond: RIFL 1.0 – The Language

Sarah Ereth, Heiko Mantel, Matthias Perner
MAIS, Computer Science, TU Darmstadt

Abstract

We present the syntax and intuition of a novel policy language for information-flow security. The RS^3 Information-Flow Specification Language (brief: RIFL) originated from the need to have a common language for specifying security requirements within the DFG priority program Reliably Secure Software Systems (brief: RS^3). At this point in time, RIFL is already supported by four analysis tools, and we hope that RIFL will be supported by further analysis tools in the future, within RS^3 and beyond.

1 Introduction

In the development of RIFL, our objective was to create a language for specifying information-flow requirements without having to commit to a particular information-flow analysis tool. By being tool independent, RIFL shall facilitate the creation of case studies on information-flow security, consisting of example programs and corresponding security requirements, that are suitable for multiple tools. Figure 1 visualizes this role of RIFL. The left hand side of the figure indicates that RIFL is suitable for expressing flow relations. Flow relations are a common concept for the abstract definition of information-flow security requirements in terms of security domains. The right hand side of the figure indicates that a RIFL specification can be provided as input to any tool supporting RIFL.

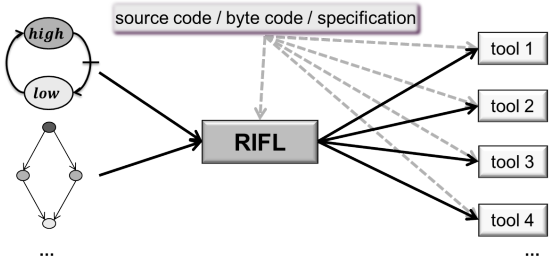

Figure 1: RIFL as Input Language for Multiple Analysis Tools

Having a common language like RIFL is helpful for comparing information-flow analysis tools with each other in experiments and for creating benchmarks that can be used in such comparisons. One could even envision the use of RIFL as glue between multiple analysis tools such that they can be used collaboratively in the information-flow analysis of different parts of a complex program.

At this point in time, preliminary versions of RIFL are already supported by Joana [14], KeY [5], SuSi [20], and the RSCP security analyzer [21]. We hope that RIFL will be adopted by other analysis tools in RS^3 [3] and beyond. For analysis tools that do not yet support RIFL by construction, support can be added by developing a front-end that translates RIFL specifications into the policy language of an analysis tool, or directly into a tool's internal data structures used for expressing security requirements.

We decided to not limit the use of RIFL 1.0 to analysis tools that presume one particular formal definition of information-flow security, because this would limit the scope and, hence, the benefits of RIFL too much. There is a wide spectrum of possible definitions of information-flow security, but no general agreement regarding which formal definition is best. That is, RIFL 1.0 is not only a tool-independent language, but also a security-property-independent language. As a consequence, RIFL 1.0 is a semi-formal language that provides a formally defined syntax with a particular intuition, but without a formal semantics.

In the definition of RIFL 1.0, we distinguish between parts that are independent from the particular language in which programs are written from parts that are specific for each programming language.

Structure of this document. In Section 2, we describe the structure of the RIFL language and the scope of version 1.0 of RIFL. We introduce the language-independent parts of RIFL in Section 3, and explain how RIFL can be specialized, to a particular programming language and to a program in such a language in Section 4. We describe the language-specific parts of RIFL for two particular languages, namely for Java source code and Dalvik bytecode in Sections 5 and 6, respectively. In both sections, we illustrate the use of RIFL for these programming languages using concrete example programs. Section 5 and Section 6 are both self-contained such that each can be read directly after reading Sections 1-4. We discuss related work in Section 7 before concluding in Section 8.

Notation. We define each syntactic element of the RIFL language using XML DTD [4]. In addition to this machine-readable form, we present each syntactic element also in BNF [10], which is a more amenable notation for human beings.

2 RIFL – Overview of the Language

RIFL allows one to specify restrictions on the flow of information from the information sources to the information sinks in a given program.

2.1 Specifying Restrictions on the Flow of Information

RIFL provides a syntax that can be used to identify *sources* and *sinks* in a program. RIFL also provides a syntax for declaring *security domains*, which constitute abstractions of concrete sources and sinks, and for defining *flow relations*, which specify restrictions on the flow of information between security domains. That is, information-flow restrictions are specified in RIFL on a more abstract level than in terms of the individual sources and sinks of a given program.

Restrictions on the flow of information from concrete sources to concrete sinks are induced by a *domain assignment*, which assigns each source and each sink to a security domain. Intuitively, information may flow from a source to a sink if information may flow from the source's security domain to the sink's security domain, where the source's and sink's security domain are determined by the domain assignment.

2.2 Structure of RIFL

While sources and sinks are generic concepts, the particular sources and sinks that may occur in programs depend on the programming language. Other RIFL concepts, e.g., security domains are independent of the programming language. This distinction between language-independent and language-specific elements is reflected in the definition of RIFL.

The definition of RIFL has a modular structure. It comprises modules that are independent from concrete target languages and modules that are specific to a particular language. The language-independent modules offer a uniform syntax for concepts that are relevant for information-flow security across different target languages. These concepts are easy to grasp and allow an intuitive specification of information-flow requirements at an abstract level. The language-specific modules complement the concepts in the language-independent modules by a syntax for identifying concrete entities in a particular target language.

Figure 2 gives an overview of the modules of RIFL. Language-independent modules are represented by white boxes at the bottom. The language-specific module for specifying sources and sinks in a program written in a given programming language is represented by the light-grey box at the bottom of the left figure. A concrete RIFL specification, capturing a security requirement for a given program, builds on both, the language-independent and language-specific modules. Such a concrete specification is indicated by the four grey boxes at the top of the the figure.

So far, language-specific modules for specifying sources and sinks in RIFL have been defined for two programming languages, namely Java source code and Dalvik bytecode. The language-specific modules for these two languages are presented in Sections 5 and 6, respectively. The adaptation to further languages, such as Java bytecode and JavaScript, is envisioned for the future.

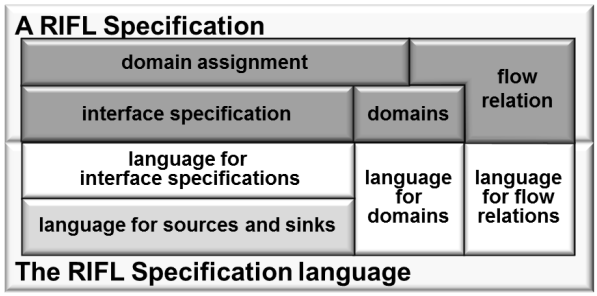

Figure 2: Structure of RIFL

2.3 Scope of RIFL 1.0

RIFL 1.0 supports the declaration of domains and definition of flow relations for specifying information-flow policies. RIFL 1.0 further supports the declaration of sources and sinks – possibly grouped into categories – for specifying the interface of a program. Finally, RIFL 1.0 supports the definition of a domain assignment for relating the specification of the interface to the specification of the information-flow policy.

The syntax of RIFL specifications (in RIFL 1.0) is specified by the following BNF and DTD. Throughout this report, we define a concrete XML syntax for RIFL using DTD to be used in tools. Additionally, we define an abstract syntax for RIFL using BNF. The sole purpose of the abstract syntax is to ease the reader's understanding of the RIFL syntax.

BNF representation of syntactic elements

RIFL-SPEC ::= (DOMAINS, FLOW-RELATION, INTERFACESPEC, DOMAIN-ASSIGNMENT)

XML DTD definition of syntactic elements

```
<!ELEMENT riflspec (domains, flowrelation,
                    interfacespec, domainassignment)>
```

Formally, a RIFL specification comprises definitions of *domains*, a *flow relation*, an *interface specification*, and a *domain assignment*. In the BNF, these elements appear as non-terminals for which we provide production rules later in the report. For more information on BNF, we refer to its introduction in [10]. In the DTD the element `riflspec` defines an XML tag that encloses a RIFL specification. The subsequent comma-separated list specifies that there must be exactly one child node of each of the elements `domains`, `flowrelation`, `interfacespec`, and `domainassignment`. For more information on XML and DTD, we refer to [4].

At this point, we leave undefined the production rules for non-terminals on the right-hand side of production rules in the BNF. We also leave the cor-

responding element declarations in the DTD undefined. The following table points the reader to the sections where these definitions can be found:

INTERFACESPEC, `interfacespec`	Section 3.1 (p. 5)
DOMAINS, `domains`	Section 3.2 (p. 6)
FLOW-RELATION, `flowrelation`	Section 3.2 (p. 6)
DOMAIN-ASSIGNMENT, `domainassignment`	Section 3.3 (p. 7)

RIFL 1.0 is suitable for specifying information-flow requirements, but it does not yet provide all features that one might want from a language for specifying information flow requirements. In particular, RIFL 1.0 does not provide syntax for expressing exceptions to information-flow restrictions, i.e., it does not support controlled declassification.

3 Language-Independent Modules of RIFL

In this section, we present the language-independent module of RIFL 1.0. The language-independent module supports declaration of domains and categories as well as the definition of flow relations domain assignments. It further defines the frame for the declaration of the interface of a program.

3.1 Sources and Sinks

The interface specification makes explicit at which points in a program information might be input, and at which points a program might output information. *Sources* declare at which points a program receives input. *Sinks* declare at which points a program produces output. *Categories* group sources and sinks with respect to an intuitive similarity, e.g. API calls for network communication could be grouped into a category "network". Moreover, categories can be arranged in a tree structure, i.e. a category might have sub-categories. Categories are an optional concept that can be used to structure the interface specification.

The abstract syntax for specifying sources and sinks in RIFL is specified by the following BNF and the concrete syntax is defined by the subsequent DTD.

BNF representation of syntactic elements

$$
\begin{aligned}
\text{INTERFACESPEC} &::= \epsilon \mid \text{ASSIGNABLE} \mid \text{ASSIGNABLE} :: \text{INTERFACESPEC} \\
\text{ASSIGNABLE} &::= (\text{HANDLE}, \text{CATSRCSNK}) \\
\text{CATSRCSNK} &::= \text{CATEGORY} \mid \text{SOURCE} \mid \text{SINK} \\
\text{CATSRCSNKLIST} &::= \epsilon \mid \text{CATSRCSNK} \mid \text{CATSRCSNK} :: \text{CATSRCSNKLIST} \\
\text{CATEGORY} &::= (\text{NAME}, \text{CATSRCSNKLIST})
\end{aligned}
$$

XML DTD definition of syntactic elements

```
<!ELEMENT interfacespec (assignable)*>
<!ELEMENT assignable (category | source | sink)>
<!ATTLIST assignable handle ID #REQUIRED>
<!ELEMENT category (category | source | sink)*>
<!ATTLIST category name ID #REQUIRED>
```

In the DTD the attlist tags `assignable` and `category` define the lists of attributes for the elements `assignable` and `category`. The element `assignable` has a mandatory attribute `handle` of type ID and the element `category` has a mandatory attribute `name` of type ID. The type ID requires the value of the element to be unique in an XML document.

An interface specification is a (possibly empty) list of assignables. An assignable is a pair of a unique handle and a category, source or sink. We will use the handle of an assignable to refer to this assignable in the domain assignment. A category is a tuple comprising a name and a list of further categories, sources and sinks. That is, a category is the root of a tree. Such a tree describes an is-a-relationship, i.e. a child is subsumed by its parent.

The concepts of sources and sinks are universal and thus independent of any particular programming language. However, the concrete sources and sinks of a program and their syntactic representation depend on the concrete programming language. Hence, the symbols SOURCE (`source`) and SINK (`sink`) are part of the language-specific modules.

The following table points to the BNF and DTD for the definition of sources and sinks in Java source code and Dalvik bytecode.

SOURCE, `source`	Section 6.1 (p. 16) for Dalvik bytecode, Section 5.1 (p. 12) for Java source code
SINK, `sink`	Section 6.1 (p. 16) for Dalvik bytecode, Section 5.1 (p. 12) for Java source code

We say that a specification is *compatible* with a program only if the specification covers all sources and sinks that appear in the program. In contrast, we say that a program is not compatible with a program, if a source or sink appears in the program that is not covered by the specification. If a specification is not compatible with a program, then it is not clear from the specification itself whether a source or sink was forgotten during specification or whether it is not part of the specification intentionally (i.e. the flow from the source or into the sink shall not be restricted in any way). Hence, the notion of compatibility induces a sanity check between a specification and a program.

3.2 Domains and Flow Relation

Domains model different levels of sensitivity. The flow relation specifies between which domains information may flow. No information must flow between two domains that are not related by the flow relation.

The abstract syntax for declaring domains and defining a flow relation is specified by the following BNF and the concrete syntax is defined by the subsequent DTD.

BNF representation of syntactic elements

$$\begin{aligned} \text{DOMAINS} ::= {} & \epsilon \mid \text{DOMAIN} \mid \text{DOMAIN} :: \text{DOMAINS} \\ \text{FLOW-RELATION} ::= {} & \epsilon \mid (\text{DOMAIN}, \text{DOMAIN}) \mid \\ & (\text{DOMAIN}, \text{DOMAIN}) :: \text{FLOW-RELATION} \end{aligned}$$

XML DTD definition of syntactic elements

```
<!ELEMENT domains (domain)*>
<!ELEMENT domain EMPTY>
<!ATTLIST domain name ID #REQUIRED>
<!ELEMENT flowrelation (flow)*>
<!ELEMENT flow EMPTY>
<!ATTLIST flow from IDREF #REQUIRED to IDREF #REQUIRED>
```

The declaration of domains is a list of domains. For a given specification, this list defines the domains that may be used in the specification. The production rules for the non-terminal DOMAIN remain underspecified. The non-terminal DOMAIN ranges over strings as names for domains.

The flow relation is a list of pairs of domains. This list of pairs defines a binary relation on domains. In the concrete XML syntax specified by the DTD, we use IDs for referring to domains. We also use IDs for other language elements. In order for a relation to be a valid flow relation, the IDs in the relation must be names of domains. Since this requirement is not enforced by the syntax, it must be checked additionally.

The reflexive closure of the relation that is defined by a list FLOW-RELATION specifies the permissible information flows. That is, information may flow from a domain `d1` to a domain `d2` if the pair (`d1`, `d2`) explicitly appears in the list FLOW-RELATION or if `d1` = `d2` holds. Otherwise, information flow from `d1` to `d2` is forbidden. Similarly, the information flow permitted by an XML specification is defined as the reflexive closure of the relation on domains that is specified by `flowrelation`.

3.3 Domain Assignment

A domain assignment relates an interface specification to domains by assigning the handles of assignables to from the interface specification to domains.

The abstract syntax for defining a domain assignment is specified by the following BNF and the concrete syntax is defined by the subsequent DTD.

BNF representation of syntactic elements

$$\begin{aligned} \text{DOMAIN-ASSIGNMENT} ::= {} & \epsilon \mid (\text{HANDLE}, \text{DOMAIN}) \mid \\ & (\text{HANDLE}, \text{DOMAIN}) :: \text{DOMAIN-ASSIGNMENT} \end{aligned}$$

XML DTD definition of syntactic elements

```
<!ELEMENT domainassignment (assign)*>
<!ELEMENT assign EMPTY>
<!ATTLIST assign handle IDREF #REQUIRED domain IDREF #REQUIRED>
```

A domain assignment is a list of tuples where the first element is a handle and the second element is a domain. For a domain assignment to be well-formed, the list must define a total function from handles to domains. To this end, each handle must appear exactly once as first element in a tuple from the list.

In the concrete XML syntax specified by the DTD, we use IDs instead of handles and domains. We also use IDs for other language elements. In order for a list to be a valid domain assignment, all IDs that appear as first elements of a tuple in the list must be handles and all IDs that appear as second elements of a tuple in the list must be names of domains.

3.4 Informal Semantics of a RIFL 1.0 Specification

The informal semantics of a RIFL 1.0 specification is closely related to the intuition behind noninterference [13, 16]. That is, if no information must flow from certain sources to certain sinks according to the specification, this means that the output to these sinks must be independent of the input from these sources. Consider the following example of a RIFL specification:

```
<riflspec>
  <interfacespec>
    <assignable handle="locationhandle">
      <category name="location">
        <source name="getGPS" />
        <source name="getNetworkLocation" />
      </category>
    </assignable>
    <assignable handle="fileshandle">
      <category name="files">
        <sink name="storeToFile" />
      </category>
    </assignable>
    <assignable handle="HTTPhandle">
      <sink name="sendViaHTTP" />
    </assignable>
    <assignable handle="HTTPShandle">
      <sink name="sendViaHTTPS" />
    </assignable>
  </interfacespec>
  <domains><domain name="high" /><domain name="low" /></domains>
  <flowrelation><flow from="low" to="high" /></flowrelation>
  <domainassignment>
    <assign handle="locationhandle" domain="high" />
```

```
    <assign handle="HTTPShandle" domain="high" />
    <assign handle="fileshandle" domain="low" />
    <assign handle="HTTPhandle" domain="low" />
  </domainassignment>
</riflspec>
```

Since the syntax for specifying sources and sinks depends on the concrete programming language, we use a simplified syntax in the example: We specify sources and sinks only by a name without any further details. The actual syntax for Java source code is defined in Section 5 and for Dalvik bytecode in Section 6.

The example specification expresses that no information about the location shall be stored to files or sent via an unencrypted HTTP connection, but location information may be sent via an encrypted HTTPS connection. That is, the output to files and HTTP connections must be independent of the input received from the location providers.

The interface specification defines the four assignables `locationhandle`, `fileshandle`, `HTTPhandle` and `HTTPShandle`. The handle `locationhandle` refers to the category `location`. This category subsumes the sources `getGPS` and `getNetworkLocation`. The handle `fileshandle` refers to the category `files`. This category subsumes the sink `storeToFile`. The handle `HTTPhandle` refers to the sink `sendViaHTTP`. The handle `HTTPShandle` refers to the sink `sendViaHTTPS`.

The declaration of domains declares two domains `high` and `low`. The flow relation specifies that information may flow from `low` to `high` and within each of these two domains. This means that no information must flow from `high` to `low`. The domain assignment maps the handles `locationhandle` and `HTTPShandle` to the domain `high`, and the handles `fileshandle` and `HTTPhandle` to the domain `low`. This means that information may flow from the sources identified by the handle `locationhandle` to the sinks identified by the handle `HTTPShandle`. Moreover, no information must flow from the sources identified by the handle `locationhandle` to the sinks identified by the handles `fileshandle` and `HTTPhandle`.

Categories, sources, and sinks inherit the assignment of domains from their parents and, thus, the permitted information flows are determined by their parents. For example, information from the source `getGPS` may flow to the sink `sendViaHTTPS`. This is because `getGPS` is subsumed by the category `location`, whose handle is assigned to the domain `high`. Therefore, the source `getGPS` is implicitly assigned to the domain `high`. Since the handle of `sendViaHTTPS` is also assigned to the domain `high` and the flow relation is reflexive, information flow from `getGPS` to `sendViaHTTPS` is permitted.

Analogously, categories, sources, and sinks inherit constraints on the permitted information flows from their parents. For this reason, no information must flow, for instance, from the source `getGPS` to the sink `storeToFile`. This is because the sink `storeToFile` is subsumed by the category `files`. Since the handle of `files` is assigned to the domain `low`, the sink `storeToFile` is implicitly assigned to the domain `low` as well. Because the source `getGPS` is classified

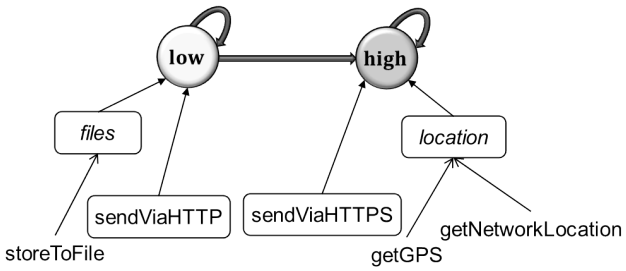

Figure 3: Visualization of the Example RIFL Specification

as `high` (as we argued for before) and the pair (`high`, `low`) is not in the flow relation, information flow from `getGPS` to `storeToFile` is not permitted.

Figure 3 visualizes the example policy. The circles represent the domains specified with the tags `domain`. The double arrows between the circles represent the permitted information flows according to the flow relation specified with the tag `flow`. The boxes with rounded corners represent assignables. The assignables enclose the categories specified with the tags `category` (represented by text in italics) as well as the sinks specified with the tags `sink` (represented by regular text). The arrows with the open arrow head represent the grouping of sources and sinks specified with the tags `source` and `sink` (representeed by regular text) into categories. Finally, the arrows with the closed arrow head represent the domain assignment as specified by the tags `assign`.

Categories allow very concise definitions of the domain assignment even for larger specifications, because they group sources and sinks with respect some notion of similarity, e.g., all API calls for accessing files. We also envision a library of specifications of sources and sinks that are already categorized. Interface specifications can then be created from such a library by choosing a part of the library. In this way, the effort for creating an interface specification is reduced, and one interface specification can be used in multiple RIFL specifications.

4 Specializing and Applying RIFL

4.1 Specializing RIFL for a Programming Language

A specialization of RIFL 1.0 for a particular programming language provides concrete syntax for identifying sources and sinks in programs that are written in said language.

The first step for specializing RIFL to a programming language is identifying what one might consider as sources and sinks of information in this language. For instance, one might consider parameters of methods in third-party libraries as sinks in Java, and one might consider fields in the Android framework as sources and also as sinks in Dalvik.

The next step is to define a syntax for identifying occurrences of relevant

sources and sinks in a program. For instance, one could use the fully qualified name of a field to identify occurrences of the field in a Java program.

To facilitate the specification of policies for programs in a given programming language, one can build up a library of categorized sources and sinks for frameworks and libraries that are often used in the programming language. This library can then be used as basis for creating the interface specifications in multiple RIFL specifications.

4.2 Applying RIFL to a Concrete Program

Writing a RIFL specification for a concrete program comprises the following steps (not necessarily in this order):

Specifying domains and flow relation:

1. Define domains that model different levels of sensitivity, e.g. low and high for a two-level security policy distinguishing only between public and private information.
2. Define a flow relation on domains that captures the permissible flows of information between distinct domains. The reflexive closure of the relation specifies the permissible information flows, e.g. information may only flow from low to high (and implicitly within each domain).

Specifying the interface of the program:

1. Declare all sources and sinks that might appear in the program. The sources and sinks in the specification must include all sources and sinks that actually appear in the program, but it may also contain further sources and sinks that do not appear in the program.
2. Optionally: Structure the sources and sinks with respect to some notion of similarity using categories, e.g. all API calls that send information to the network.
3. Assign handles to each root element in the interface specification.

Specifying the domain assignment:

Define a domain assignment that maps each handle to a domain. The domain assignment must be a total function, i.e. each handle must be mapped to exactly one domain.

We present example RIFL specifications that result from these steps for an example program written in Java in Section 5.2, and for an example program written in Dalvik in Section 6.2.

Remark. Assume there is a library of categorized sources and sinks of the kind mentioned in Section 4.1. To create an interface specification for a concrete program from such a library, one has to include categories, sources and sinks from the library. If different children of a category in the library shall be treated differently wrt. permitted information flows, i.e., shall be assigned to

different domains, then one must include the children of the category individually in the interface specification instead of including the parent category itself. This is due to the fact that RIFL only supports assigning the roots of trees comprising categories, sources, and sinks to domains, whereas all other elements in each tree are implicitly assigned to the domain of the tree's root. Therefore, no inconsistencies can be introduced between explicitly assigned domains and domains inherited from parents. This design choice does not limit expressiveness, because domains can be assigned to any node of a tree by following the aforementioned process.

5 Specialization of RIFL for Java Source Code

In this section, we present the language-specific module of RIFL 1.0 for Java source code [15]. To make this section a self-contained manual for RIFL for Java, we introduce the complete syntax with explanations even though there is a large overlap with the syntax for Dalvik bytecode in Section 6.

5.1 Sources and Sinks for Java

RIFL 1.0 for Java offers the possibility to specify formal parameters of methods, return values of methods and fields of objects as information sources and sinks.

The abstract syntax for defining concrete sources and sinks is specified by the following BNF and the concrete syntax is defined by the subsequent DTD.

BNF representation of syntactic elements

$$
\begin{aligned}
\text{SOURCE} &::= \text{PARAMETER} \mid \text{RETURN} \mid \text{FIELD} \\
\text{SINK} &::= \text{PARAMETER} \mid \text{RETURN} \mid \text{FIELD} \\
\text{PARAMETER} &::= \text{CLASS.METHOD@N} \\
\text{RETURN} &::= \text{CLASS.METHOD@return} \\
\text{METHOD} &::= \text{METHOD-NAME}(\text{PARAMETERS}) \\
\text{PARAMETERS} &::= \epsilon \mid \text{PARAMETER} \mid \text{PARAMETER}, \text{PARAMETERS} \\
\text{FIELD} &::= \text{CLASS.FIELD}
\end{aligned}
$$

XML DTD definition of syntactic elements

```
<!ELEMENT source (parameter | returnvalue | field)>
<!ELEMENT sink (parameter | returnvalue | field)>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter class CDATA #REQUIRED
  method CDATA #REQUIRED parameter CDATA #REQUIRED>
<!ELEMENT returnvalue EMPTY>
<!ATTLIST returnvalue class CDATA #REQUIRED
  method CDATA #REQUIRED>
<!ELEMENT field EMPTY>
<!ATTLIST field class CDATA #REQUIRED name CDATA #REQUIRED>
```

The non-terminals CLASS and PARAMETER as well as the possible values of `class` range over fully qualified names of classes and interfaces [15, §6.7], e.g. `package.Class`. The non-terminals METHOD-NAME and FIELD as well as the possible values of the attribute `name` range over identifiers as specified in [15, §3.8]. These identifiers represent method names and field names in a given class. The possible values of the attribute `method` range over method signatures as specified by the non-terminal METHOD, e.g. `method(java.lang.String)`, and identify a method in a given class.

The non-terminal N and the values of the attribute `parameter` range over the natural numbers. A number identifies the paramter by its position in the list of formal parameters. The input parameters of a method start at position 1 and the "this" pointer of an object is identified by postion 0.

The syntax is suitable for specifying the following types of sources:

Formal parameters of methods If a method of the program can be called from outside the program and receives values via its formal parameters, these parameters can be considered information sources.

Fields If a field of the program is accessible from the outside the program, the field can be considered an information source, because input might be received as value of the field.

Return values of external methods If a method outside the program, e.g. in a library, is called by the program, the return value of the method can be considered an information source, because the value returned by the method might be used as input.

The syntax is suitable for specifying the following types of sinks:

Return value of methods If a method of the program can be called from outside the program, the return value of the method can be considered an information sink.

Fields If a field of the program is accessible from outside the program, the field can be considered an information sink, because values written to the field are observable from outside the program.

Formal parameters of external methods If a program calls a method outside the program, the formal parameters of the method call can be considered information sinks.

5.2 Example of a RIFL Specification for Java Source Code

5.2.1 Simple Password Program in Java Source Code

Consider the Java source code program in Listing 1 that implements a simple password prompt. The password is input via the command line. The security requirement is that the password read from the command line with the method call of `readLine()` in line 9 should be kept secret.

Listing 1: Example Java Program

```
1  package de.spp_rs3;
2
3  public class Main{
4    public static void main(String[] args) {
5      try {
6        BufferedReader br = new BufferedReader(
7          new InputStreamReader(System.in));
8        System.out.println("Please enter your password:");
9        String password = br.readLine();
10       System.out.println(password);
11     } catch (IOException e) {
12       e.printStackTrace();
13     }
14   }
15 }
```

5.2.2 RIFL Specification for the Simple Password Program

The XML specification in Listing 2 captures the desired information-flow requirement for the program in Listing 1.

The sources of the program in Listing 2 are the formal parameters of the method `main`, the return value of the called method `readLine` in line 9, and the fields `System.in` and `System.out`. The sinks of the program are the parameters of the method calls to `println` in line 8 and 11, and of the constructors of `InputStreamReader` and `BufferedReader`. As an example, consider the source

```
<source>
  <parameter class="de.spp_rs3.Main"
    method="main(java.lang.String[])" parameter="1" />
</source>
```

from the specification. The attribute `class="de.spp_rs3.Main"` corresponds to the fully qualified class name, i.e. lines 1-3 in the program. The attribute `method="main(java.lang.String)"` corresponds to the method signature, i.e. line 4 in the program. In the signature, `main` corresponds to the method name and `java.lang.String[]` corresponds to the fully qualified array type name of the method parameter. Finally, the attribute `parameter="1"` refers to the first actual parameter of the method.

The specification declares two domains `low` and `high` and defines a flow relation $\{(\texttt{low},\texttt{high}),(\texttt{low},\texttt{low}),(\texttt{high},\texttt{high})\}$. That means information may flow within each domain and from `low` to `high`, but no information must flow from `high` to `low`.

Since we want to keep the password entered via command line secret, the domain assignment maps the handle of the respective source, i.e. the handle

Listing 2: Example RIFL Specification for Java

```
<riflspec>
  <interfacespec>
    <assignable handle="cmdinputhandle">
      <source>
        <returnvalue class="java.io.BufferedReader"
          method="readLine()" />
      </source>
    </assignable>
    <assignable handle="cmdoutputhandle">
      <sink>
        <parameter class="java.io.PrintStream"
          method="println(java.lang.String)" parameter="1" />
      </sink>
    </assignable>
    <assignable handle="envinputhandle">
      <category name="envinput">
        <source>
          <parameter class="de.spp_rs3.Main"
            method="main(java.lang.String[])" parameter="1" />
        </source>
        <source><field class="java.lang.System" name="in" /></source>
        <source><field class="java.lang.System" name="out" /></source>
      </category>
    </assignable>
    <assignable handle="envoutputhandle">
      <category name="envoutput">
        <sink>
          <parameter class="java.io.InputStreamReader"
            method="InputStreamReader(java.io.InputStream)"
            parameter="1" />
        </sink>
        <sink>
          <parameter class="java.io.BufferedReader"
            method="BufferedReader(java.io.Reader)" parameter="1" />
        </sink>
      </category>
    </assignable>
  </interfacespec>
  <domains><domain name="high" /><domain name="low" /></domains>
  <flowrelation><flow from="low" to="high" /></flowrelation>
  <domainassignment>
    <assign handle="cmdinputhandle" domain="high" />
    <assign handle="cmdoutputhandle" domain="low" />
    <assign handle="envinputhandle" domain="low" />
    <assign handle="envoutputhandle" domain="low" />
  </domainassignment>
</riflspec>
```

`cmdin` of the source

```
<source>
  <returnvalue class="java.io.BufferedReader" method="readLine()" />
</source>
```

to `high`, and all other handles to `low`.

The example specification illustrates how the use of categories in the interface specification enables a concise specification of the domain assignment. Due to the grouping of the sources and sinks in the categories `envinput` and `envoutput`, we only need four assign tags in the domain assignment instead of seven assign tags.

6 Specialization of RIFL for Dalvik Bytecode

In this section, we present the language-specific module of RIFL 1.0 for Dalvik bytecode [1]. To make this section a self-contained manual for RIFL for Dalvik, we introduce the complete syntax with explanations even though there is a large overlap with the syntax for Java source code in Section 5.

6.1 Sources and Sinks for Dalvik Bytecode

RIFL 1.0 for Dalvik offers the possibility to specify formal parameters of methods, return values of methods and fields of objects as information sources and sinks.

The abstract syntax for defining concrete sources and sinks is specified by the following BNF and the concrete syntax is defined by the subsequent DTD.

BNF representation of syntactic elements

SOURCE	::=	PARAMETER | RETURN | FIELD
SINK	::=	PARAMETER | RETURN | FIELD
PARAMETER	::=	CLASS−>METHOD@N
RETURN	::=	CLASS−>METHOD@return
METHOD	::=	METHOD-NAME(PARAMETERS)PARAMETER
PARAMETERS	::=	ϵ | PARAMETER | PARAMETERPARAMETERS
FIELD	::=	CLASS−>FIELD

XML DTD definition of syntactic elements

```
<!ELEMENT source (parameter | returnvalue | field)>
<!ELEMENT sink (parameter | returnvalue | field)>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter class CDATA #REQUIRED
   method CDATA #REQUIRED parameter CDATA #REQUIRED>
<!ELEMENT returnvalue EMPTY>
<!ATTLIST returnvalue class CDATA #REQUIRED
   method CDATA #REQUIRED>
<!ELEMENT field EMPTY>
<!ATTLIST field class CDATA #REQUIRED name CDATA #REQUIRED>
```

The non-terminals CLASS and PARAMETER as well as the values of `class` range over type descriptors of classes and interfaces [2], e.g. `Lpackage/Class;`. The non-terminals METHOD-NAME and FIELD as well as the possible values of the attribute `name` range over simple names as specified in [2]. These simple names represent method names and field names in a given class. The possible values of the attribute `method` range over method signatures as specified by the non-terminal METHOD, e.g. `method(Lpackage/Class;)V`, and identify a method in a given class.

The non-terminal N and the values of the attribute `parameter` range over the natural numbers. A number identifies the paramter by its position in the list of formal parameters. The input parameters of a method start at position 1 and the "this" pointer of an object is identified by postion 0.

The syntax is suitable for specifying the following types of sources:

Formal parameters of methods If a method of the program can be called from outside the program and receives values via its formal parameters, these parameters can be considered information sources.

Fields If a field of the program is accessible from the outside the program, the field can be considered an information source, because input might be received as value of the field.

Return values of external methods If a method outside the program, e.g. in a library, is called by the program, the return value of the method can be considered an information source, because the value returned by the method might be used as input.

The syntax is suitable for specifying the following types of sinks:

Return value of methods If a method of the program can be called from outside the program, the return value of the method can be considered an information sink.

Fields If a field of the program is accessible from outside the program, the field can be considered an information sink, because values written to the field are observable from outside the program.

Listing 3: Example Dalvik Program

```
de.spp_rs3.Main.main:([Ljava/lang/String;)V
0000: new-instance v0, java.io.BufferedReader
0002: new-instance v1, java.io.InputStreamReader
0004: sget-object v2, java.lang.System.in:Ljava/io/InputStream;
0006: invoke-direct {v1, v2},
      java.io.InputStreamReader.<init>:(Ljava/io/InputStream;)V
0009: invoke-direct {v0, v1},
      java.io.BufferedReader.<init>:(Ljava/io/Reader;)V
000c: sget-object v1, java.lang.System.out:Ljava/io/PrintStream;
000e: const-string v2, "Please enter your password:"
0010: invoke-virtual {v1, v2},
      java.io.PrintStream.println:(Ljava/lang/String;)V
0013: invoke-virtual {v0},
      java.io.BufferedReader.readLine:()Ljava/lang/String;
0016: move-result-object v0
0017: sget-object v1, java.lang.System.out:Ljava/io/PrintStream;
0019: invoke-virtual {v1, v0},
      java.io.PrintStream.println:(Ljava/lang/String;)V
001c: return-void
001d: move-exception v0
001e: invoke-virtual {v0}, java.io.IOException.printStackTrace:()V
0021: goto 001c
tries:
try 0000..001c
catch java.io.IOException -> 001d
```

Formal parameters of external methods If a program calls a method outside the program, the formal parameters of the method call can be considered information sinks.

6.2 Example of a RIFL Specification for Dalvik Bytecode

6.2.1 Simple Password Program in Dalvik Bytecode

Consider the Dalvik bytecode in Listing 3 that implements a simple password prompt. The assembly was created using dexdump and was simplified for better readability. The password is input via the command line. The security requirement is that the password read from the command line with the method call of `readLine()` at position 0013 should be kept secret.

6.2.2 RIFL Specification for the simple password program

The XML specification in Listing 4 captures the desired information-flow requirement for the program in Listing 3.

The sources of the program in Listing 4 are the formal parameters of the method `main`, the return value of the called method `readLine` at position 0013, and the fields `System.in` and `System.out`. The sinks of the program are the formal parameters of the constructors of `InputStreamReader` and `BufferedReader`

Listing 4: Example RIFL Specification for Dalvik

```
<riflspec>
  <interfacespec>
    <assignable handle="cmdinputhandle">
      <source>
        <returnvalue class="Ljava/io/BufferedReader;"
          method="readLine()Ljava/lang/String;" />
      </source>
    </assignable>
    <assignable handle="cmdoutputhandle">
      <sink>
        <parameter class="Ljava/io/PrintStream;"
          method="println(Ljava/lang/String;)V" parameter="1" />
      </sink>
    </assignable>
    <assignable handle="envinputhandle">
      <category name="envinput">
        <source>
          <parameter class="Lde/spp_rs3/Main;"
            method="main([Ljava/lang/String;)V" parameter="1" />
        </source>
        <source><field class="Ljava/lang/System;" field="in" /></source>
        <source><field class="Ljava/lang/System;" field="out" /></source>
      </category>
    </assignable>
    <assignable handle="envoutputhandle">
      <category name="envoutput">
        <sink>
          <parameter class="Ljava/io/InputStreamReader;"
            method="<init>(Ljava/io/InputStream;)V" parameter="1" />
        </sink>
        <sink>
          <parameter class="Ljava/io/BufferedReader;"
            method="<init>(Ljava/io/Reader;)V" parameter="1" />
        </sink>
      </category>
    </assignable>
  </interfacespec>
  <domains><domain name="high" /><domain name="low" /></domains>
  <flowrelation><flow from="low" to="high" /></flowrelation>
  <domainassignment>
    <assign handle="cmdinputhandle" domain="high" />
    <assign handle="cmdoutputhandle" domain="low" />
    <assign handle="envinputhandle" domain="low" />
    <assign handle="envoutputhandle" domain="low" />
  </domainassignment>
</riflspec>
```

and the parameter of the method calls to `println` at positions 0010 and 0019. As an example, consider the source

```
<source>
  <parameter class="Lde/spp_rs3/Main;"
    method="main([Ljava/lang/String;)V" parameter="1" />
</source>
```

from the specification. The attribute `class="Lde/spp_rs3/Main;"` corresponds to the type descriptor of the class containing the method, i.e. lines 1 in the program. The attribute `method="main([Ljava/lang/String;)V"` corresponds to the method signature, i.e. line 1 in the program. In the signature, `main` corresponds to the simple name of the method, `[Ljava/lang/String;` corresponds to the type descriptor of the method parameter, and `V` corresponds to the type descriptor of the return value of the method. Finally, the attribute `parameter="1"` refers to the first actual parameter of the method.

The specification declares two domains `low` and `high` and defines a flow relation $\{(\texttt{low}, \texttt{high}), (\texttt{low}, \texttt{low}), (\texttt{high}, \texttt{high})\}$. That means information may flow within each domain and from `low` to `high`, but no information must flow from `high` to `low`.

Since we want to keep the password entered via command line secret, the domain assignment maps the handle of the respective source, i.e. the handle `cmdin` of the source

```
<source>
  <returnvalue class="Ljava/io/BufferedReader;"
    method="readLine()Ljava/lang/String;" />
</source>
```

to `high`, and all other handles to `low`.

The example specification illustrates how the use of categories in the interface specification enables a concise specification of the domain assignment. Due to the grouping of the sources and sinks in the categories `envinput` and `envoutput`, we only need four assign tags in the domain assignment instead of seven assign tags.

7 Related Work

Information-flow control is an established research area (see, e.g. [18, 19], for two surveys) and a wide range of tools has been proposed for different programming languages. This variety of tools comes with a variety of different specification languages for information-flow requirements that shall be checked with these tools. Giving an overview of all existing languages is out of the scope of this report. Nevertheless, we want to present a selection of languages that are used in existing tools for Java and Dalvik.

JFlow/JIF and Paragon. JFlow/JIF [8, 9] is an extension of the Java programming language with security types. The security types are represented

as labels attached to data types in Java. These labeled types can occur at almost every place where data types may appear, e.g. field declarations, variable declarations, and parameters of methods. For instance, a field declaration `int{o1: r1, r2; o2: r1} x;` declares a field `x` with two owners `o1` and `o2` that may be read by its owners as well as all readers on which the owners agree, namely `r1` but not `r2`. Information may flow from one container, e.g. a variable, to a second container, e.g. a field, only if the label of the second container is at least as restrictive as the label of the first container.

Paragon [11] is another extension of the Java programming language with security types. Similar to JFlow/JIF's labels, policies in Paragon label an information container according to where the information from this container may flow. In contrast to JFlow/JIF's labels, policies in Paragon enable one to specify explicitly where information from a container may flow while in JFlow/JIF information may flow to any container that has a more restrictive label than the origin of the information. Paragon additionally has an explicit state under which a policy is evaluated. This state is modeled with locks that can be opened and closed using designated instruction in the program. For instance, information from an information container labeled with `policy p = { File f: Owns(f, alice) }` may flow to every file `f` that is owend by `alice`. The ownership is modeled with a lock `Owns(f, alice)`. Intuitively, a file `f` is owned by `alice` in the current state of the policy, if the lock is open. Dually, the file `f` is not owned by `alice` in the current state of the policy, if the lock is closed.

One difference between RIFL and the policy languages of JFlow/JIF and Paragon is that RIFL policies are separate from the program code while the policies of JFlow/JIF and of Paragon are a part of the program code. Both approaches have advantages and disadvantages. With the policies being part of the program, it is possible to treat some parts of the policies as first-class citizens in the language. This enables a programmer to encode security decisions based on the policy inside the program. On the other hand, having the policy and program code in separate files provides a clearer separation between specifying the security concerns and implementing functionality. In particular, a program can easily be checked against several policies without changing the program code, which is beneficial, for instance, when different users of a program have different security concerns.

IFT. The Information Flow Type-Checker (IFT) [17] verifies the information-flow security of Android apps given as Java source code. IFT determines which flows of information are permitted based on a flow-policy file, and on source code annotations in the analyzed program.

The flow-policy file specifies a transitive, binary relation between predefined sources and sinks, e.g., `LOCATION -> INTERNET`. If a source is in relation with a sink, then information may flow from this source to this sink. The sources and sinks include all resources protected by Android permissions, like the device's location and the network. Further sources and sinks cover resources like user input, the accelerometer, and the device's display. Moreover, some sources and

sinks are parametric to allow for a fine-grained specification of flow policies, e.g., `FILESYSTEM("notes/")`.

The source code annotations `@Source` and `@Sink`, respectively, assign sources and sinks to any occurrence of data types in Java programs. In particular, a programmer annotates the declaration of fields, the declaration of formal parameters of methods, and the declaration of the return type of methods. The annotations of remaining occurrences of data types, e.g., in the declaration of local variables, are usually defaulted or inferred by the analysis. Intuitively, `@Source` declares possible origins of values stored in the annotated resource, whereas `@Sink` declares possible destinations to which the stored values may be sent. Annotations may also be used on the data types of cast operations. In this special case, they allow to explicitly declassify information, i.e., to implement a flow that otherwise violates the flow policy.

The predefined sources and sinks in the policy language of IFT roughly correspond to one possible use of categories in RIFL. One particularly interesting feature of IFT's sources and sinks is that some can be parameterized for a more fine-grained categorisation. In contrast to RIFL, the IFT-policy files define the permitted flows directly on the predefined categories while RIFL permits an additional abstraction step to domains that group all categories that should be treated similarly with respect to security, e.g. sources that introduce information which should be kept confidential. Hence, RIFL enables very concise definitions of the flow policy. In contrast to JFlow's and Paragon's labels, the source-code annotations in IFT describe a grouping with respect to functionality and not with respect to security concerns. Hence, these annotations can be used to check a given program against different policies, similar to categories RIFL.

Information Flow Policies for Java-Enabled Smart Cards. In [12], the authors present a policy language to define information-flow policies for Java bytecode and propose to verify that a class file satisfies such a policy with a custom class loader. A policy defines what fields of an object may contain secrets and to which other classes secrets from this class may flow. Furthermore, an object of a class may implicitly share its secrets with all other objects of the same class. For instance, the policy `Ca fsa, fsb; Cb fs1 fs2;` defines that the fields `fsa` and `fsb` of class `Ca` as well as the fields `fs1` and `fs2` of class `Cb` may contain secrets while all other fields (including fields of other classes) must not contain secrets. This policy can be extended with a statement `Ca shares with Cb` to allow that secrets from class `Ca` may flow to class `Cb`. That is, information from a field that may contain secrets in class `Ca` may flow to a field that may contain secrets in class `Cb`. The policy statement `Cb strict secret` specifies that the secrets of one instance of class `Cb` must not be shared with other instances of the same class.

In this policy language, the fields of classes are considered as information sources and sinks, like in RIFL. The current specializations of RIFL to Dalvik bytecode and Java source code are more general in the sense that the parameters of methods can also be considered as sources and sinks. The policy language

in [12] allows one to specify that secrets between different instances of classes should not be shared in addition. This is currently not possible in RIFL, but could be introduced in the future if the need arises.

SCF. The SideChannelFinder (SCF) [6] is a tool for the detection of possible timing side channels in Java implementations of cryptographic implementations. For this purpose, SCF employs a security type system that is parametric in an information-flow requirement specified in XML [7]. The policy language of the SCF enables assigning security domains to fields of classes as well as to parameters of methods (including the return parameter).

The types of sources and sinks in the SCF are identical to the types of sources and sinks in the Java-specific definitions of RIFL 1.0. Unlike RIFL, the information-flow requirement implicitly assumes two security domains, namely 0 and 1 where 0 is a domain for public information and 1 is a domain for secret information. The implicit flow policy in SCF allows information flows within each domain and from 0 to 1. Having the declaration and definition of the security domains as well as the flow relation explicit in the specification language, like in RIFL, enables the specification of a wider range of policies, e.g. multi-level security policies.

8 Conclusion

In this report, we defined version 1.0 of RIFL as a semi-formal specification language for information-flow requirements, i.e. a language with formal syntax and informal semantics. The concrete syntax of RIFL facilitates the specification of information-flow requirements for different programming languages in terms of sources and sinks that occur in a concrete program. The generic parts of RIFL are independent of a concrete programming language. In order to specialize RIFL for a concrete programming language, one needs to provide a concrete syntax for specifying sources and sinks. We have provided specializations of RIFL for Java source code and Dalvik bytecode. We have also illustrated how this specializations can be used for specifying information-flow requirements for concrete programs.

RIFL 1.0 is already supported by the RSCP security analyser and its integration into Cassandra [21]. Moreover, earlier versions of RIFL are already supported by Joana [14], SuSi [20], and KeY [5]. We hope that RIFL will be adopted by other information-flow analysis tools in RS^3 [3] and beyond. Moreover, RIFL shall serve as a basis for using different tools in combination. Furthermore, we envision that the use of RIFL in different research groups leads to a library of example programs with information-flow requirements that can be used to test information-flow analysis tools.

For the future, we plan to augment the scope of RIFL by integrating syntax for specifying exceptional information flows. This will allow more expressive policies as it enables the specification of controlled release of secrets. We also

plan to specialize RIFL for further programming languages, e.g. for Java bytecode and JavaScript. Finally, we might provide formal semantics for RIFL in the future. The design choice for informal semantics was deliberate such that RIFL can be supported by information-flow analysis tools that enforce different noninterference-like security conditions. To address this, we could provide several alternative semantics for RIFL to choose from in the specification of an information-flow requirement.

Acknowledgments. We thank all researchers in RS^3 for valuable feedback and discussions on earlier versions of RIFL. In particular, we want to thank Steven Arzt, Daniel Bruns, Steffen Lortz, Martin Mohr, Siegfried Rasthofer, Christoph Scheben, and David Schneider who already implemented support for earlier versions of RIFL in tools from their research group and provided feedback based on these implementations. This work was funded by the DFG under the project RSCP (MA 3326/4-2) in the priority program RS^3 (SPP 1496).

References

[1] `https://source.android.com/devices/tech/dalvik/`. accessed November 14, 2014.

[2] `https://source.android.com/devices/tech/dalvik/dex-format.html`. accessed November 14, 2014.

[3] `http://www.spp-rs3.de/`. accessed November 14, 2014.

[4] `http://www.w3.org/TR/2008/REC-xml-20081126/`. accessed November 14, 2014.

[5] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY Platform for Verification and Analysis of Java Programs. In *Proceedings of the 6th Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 1–17, 2014.

[6] Alexander Lux and Artem Starostin. A Tool for Static Detection of Timing Channels in Java. *Journal of Cryptographic Engineering*, 1(4):303–313, 2011.

[7] Alexander Lux and Heiko Mantel and Matthias Perner and Artem Starostin. Side Channel Finder (Version 1.0). Technical Report TUD-CS-2010-0155, TU Darmstadt, October 2010.

[8] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th Symposium on Principles of Programming Languages 1999*, pages 228–241, 1999.

[9] Owen Arden, Stephen Chong, Jed Liu, Andrew C. Myers, Nate Nystrom, Krishnaprasad Vikram, Steve Zdancewic, Danfeng Zhang, and Lantian Zheng. Jif: Java information flow. Software release: `http://www.cs.cornell.edu/jif/`, July 2014.

[10] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language algol 60. *Numerische Mathematik*, 2(1):106–136, 1960.

[11] Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of 11th Asian Symposium on Programming Languages and Systems*, pages 217–232, 2013.

[12] Dorina Ghindici and Isabelle Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. In *Proceedings of the 8th International Conference on Smart Card Research and Advanced Applications*, pages 32–47, 2008.

[13] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[14] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages*, pages 123–138, 2013.

[15] James Gosling and Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification - Java SE 8 Edition*. `http://docs.oracle.com/javase/specs/jls/se8/html/index.html`, accessed 2014-03-03.

[16] Heiko Mantel. Information Flow and Noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 605–607. 2011.

[17] Michael D. Ernst and René Just and Suzanne Millstein and Werner M. Dietl and Stuart Pernsteiner and Franziska Roesner and Karl Koscher and Paulo Barros and Ravi Bhoraskar and Seungyeop Han and Paul Vines and Edward X. Wu. Collaborative Verification of Information Flow for a High-assurance App Store. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 1092–1104, November 4–6, 2014.

[18] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[19] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[20] Siegfried Rasthofer and Steven Arzt and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the 18th Symposium on Network and Distributed System Security*, 2014.

[21] Steffen Lortz and Heiko Mantel and Artem Starostin and Timo Bähr and David Schneider and Alexandra Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104, 2014.