RIFL 1.1: A Common Specification Language for Information-Flow Requirements

Technical Report TUD-CS-2017-0225

August 2017

Thomas Bauereiß[1], Simon Greiner[2], Mihai Herda[2], Michael Kirsten[2], Ximeng Li[3], Heiko Mantel[3], Martin Mohr[2], Matthias Perner[3], David Schneider[3], Markus Tasch[3]

[1] German Research Center for Artificial Intelligence (DFKI) Bremen

[2] Karlsruhe Institute of Technology

[3] Technische Universität Darmstadt

a cooperation of:

RIFL 1.1: A Common Specification Language for Information-Flow Requirements

Thomas Bauereiß[1], Simon Greiner[2], Mihai Herda[2], Michael Kirsten[2], Ximeng Li[3], Heiko Mantel[3], Martin Mohr[2], Matthias Perner[3], David Schneider[3], Markus Tasch[3]

[1] German Research Center for Artificial Intelligence (DFKI) Bremen
[2] Karlsruhe Institute of Technology
[3] Technische Universität Darmstadt

Abstract

The RS[3] Information-Flow Specification Language (RIFL) is a policy language for information-flow security. RIFL originated from the need for a common language for specifying security requirements within the DFG priority program Reliably Secure Software Systems (RS[3]) [30]. In this report, we present the syntax and informal semantics of RIFL 1.1, the most recent version of RIFL. At this point in time, RIFL is supported by four tools for information-flow analysis. We believe that RIFL can also be useful as a policy language for further tools, and we encourage its adoption and extension by the community.

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.

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.

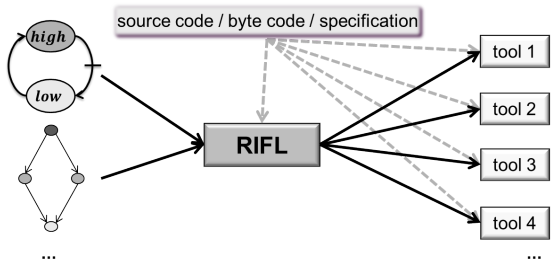

Figure 1: RIFL as an Input Language for Multiple Analysis Tools

RIFL was developed as a tool-independent language. For an analysis tool that does not yet support RIFL by construction, one can add support by developing a front-end that translates RIFL specifications into the policy language of the tool or directly into the tool's internal data structures used for expressing security requirements. At this point in time, the analysis tools Cassandra [19, 8], JoDroid [27], Joana [15], and KeY [1] already support RIFL 1.1 or the previous version of the language, RIFL 1.0. We encourage support of RIFL as a policy language also by further information-flow analysis tools.

In the development of RIFL we decided not to limit the use of RIFL 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 is not only a tool-independent language, but also a security-property-independent language. As a consequence, RIFL 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, 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.

This report defines the syntax and informal semantics of RIFL 1.1, the latest version of RIFL. It supersedes an earlier technical report defining RIFL 1.0 [9].

RIFL 1.0. RIFL 1.0 provides the following features:

- The capability to identify parameters of methods, return values of methods, and fields of objects as sources and sinks in Java source code and Dalvik bytecode.
- The capability to define the interface of a program in terms of sources and sinks as well as, optionally, categories of sources and sinks.
- The capability to declare domains and to define flow relations for specifying information-flow policies.
- The capability to define domain assignments to relate the specification of the interface to the specification of the information-flow policy.

RIFL 1.1. RIFL 1.1 is an update and extension of RIFL 1.0. In comparison to RIFL 1.0, it provides the following features:

- The capability to specify controlled declassification [20] of information in Java source code programs.
- The capability to specify information-flow requirements for Java bytecode programs.
- The capability to specify the content of arrays as a source or a sink in Java source code programs and in Java and Dalvik bytecode programs.
- The capability to specify the fact that an exception is thrown by a method as a source or a sink in Java source code programs and in Java and Dalvik bytecode programs.
- The capability to specify fields as sources more precisely by means of access paths in Java source code programs and in Java and Dalvik bytecode programs.

Structure of this document. In Section 2, we describe the overall structure of the RIFL language. We introduce the language-independent parts of RIFL that can be used to express general concepts from information-flow security in Section 3. We explain how RIFL can be specialized to a particular programming language and used to specify information-flow requirements for a program in such a language in Section 4. The language-specific parts of RIFL for expressing sources and sinks in a program are described in Sections 5 and 6 for three particular target languages. Section 5 describes the specialization of RIFL to Java source code; Section 6 describes the specialization of RIFL to Java and Dalvik bytecode. 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 [11]. In addition to this machine-readable form, we also present each syntactic element using BNF [3], which is a more easily understandable notation for human beings.

2 RIFL – Overview of the Language

The underlying model of RIFL is that a program is executed in an environment from which it may obtain information via sources and to which it in turn may pass information via sinks. The environment may consist, e.g., of the API of the operating system, libraries used by the program, other programs running concurrently, or the user interacting with the system. 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* of information 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 security domains of the source and the sink are determined by the domain assignment.

Exceptions to the restrictions on the flow of information can be specified by *escape hatches*. Intuitively, an escape hatch specifies that certain information may flow to sinks of a specific security domain, even if this is not allowed by the flow relation.

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 programs of a particular target language.

The bottom part of Figure 2 gives an overview of the modules of RIFL. Language-independent modules are represented by white boxes. Language-specific modules are represented by light-grey boxes. A box on top of another box indicates that the module represented by the box on top relies on the module represented by the box underneath.

So far, language-specific modules for specifying sources and sinks in RIFL have been defined for three programming languages, namely Java source code, Java bytecode, and Dalvik bytecode. The language-specific modules for these three languages are presented in Sections 5 (Java source code) and 6 (Java bytecode and Dalvik bytecode), respectively. Specializations of RIFL for further

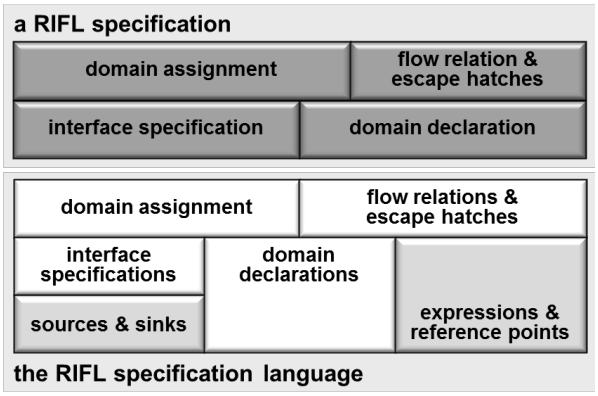

Figure 2: Structure of RIFL and RIFL specifications

languages, such as JavaScript, C, or Python, are envisioned for the future.

Changes since RIFL 1.0. The changes between version 1.0 and version 1.1 of RIFL are located in the following modules of the language (as shown in the bottom part of Figure 2): The language-independent module for specifying flow relations has been extended by means for expressing escape hatches compared to RIFL 1.0. Furthermore, the language-specific module for specifying declassification expressions and reference points in Java source code is new in RIFL 1.1 compared to RIFL 1.0. Moreover, there is a new language-specific module for specifying sources and sinks in Java bytecode. Finally, the existing language-specific modules for specifying sources and sinks in Java source code and Dalvik bytecode have been extended by means to specify the occurrence of exceptions as sources and sinks, and by means to specify fields as sources using access paths. (The language-specific modules for specifying sources and sinks in different languages are all represented by the box "sources & sinks" in the bottom part of Figure 2.) All other modules are the same in RIFL 1.1 and in RIFL 1.0.

2.3 Structure of a RIFL 1.1 Specification

The top part of Figure 2 gives an overview on the elements of a concrete RIFL specification. The elements of a RIFL specification for a given program are represented by dark-grey boxes. Again, a box on top of another box indicates that the element represented by the box on top relies on the element represented by the box underneath. Furthermore, a RIFL specification builds on both the language-independent and the language-specific modules of RIFL.

Changes since RIFL 1.0. Compared to a RIFL 1.0 specification, a RIFL 1.1 specification may also define escape hatches (element "flow relation & escape hatches" in the top part of Figure 2). Furthermore, the interface specification of

a RIFL 1.1 specification might be different from that of a RIFL 1.0 specification because there are new ways of specifying sources and sinks in the specializations of RIFL 1.1 (element "interface specification" in the top part of Figure 2).

The syntax of a RIFL 1.1 specification 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. For more information on BNF, we refer to its introduction in [3]. For more information on XML and DTDs, we refer to [11].

BNF representation of syntactic elements

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

XML DTD definition of syntactic elements

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

Formally, a RIFL specification consists of definitions of an *interface specification*, a list of *domains*, a *flow relation*, a *domain assignment*, and, optionally, a list of *escape hatches*. In the BNF, these elements appear as non-terminals for which we provide production rules later in the report. In the DTD, the element `riflspec` is the root of a RIFL specification. The subsequent comma-separated list specifies that there must be exactly one child element each of the types `interfacespec`, `domains`, `flowrelation`, and `domainassignment`, and that there may also be an optional child element of the type `hatches`.

At this point, we leave the production rules for non-terminals on the right-hand side of the production rule in the BNF undefined. We also leave the corresponding element type 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. 7)
DOMAINS, `domains`	Section 3.2 (p. 8)
FLOW-RELATION, `flowrelation`	Section 3.2 (p. 8)
DOMAIN-ASSIGNMENT, `domainassignment`	Section 3.3 (p. 9)
HATCHES, `hatches`	Section 3.4 (p. 10)

3 Language-Independent Modules of RIFL

In this section, we present the language-independent modules of RIFL 1.1. These modules support the declaration of domains and categories as well as the definition of flow relations and domain assignments. They further define the frame for the declaration of the interface of a program.

3.1 Interface Specification

The interface specification makes explicit where (in the program code) a program reads input and where (in the program code) a program provides output. *Sources* identify locations in the code where input is read. *Sinks* identify locations in the code where output is provided. *Categories* group sources and sinks with respect to an intuitive similarity. For example, 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

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

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 attlists `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 use of the type `ID` requires the value of the attribute to be unique in an XML document.

An interface specification is a list of *assignables*. An assignable is a pair of a unique handle and a category, source, or sink. It specifies a set of sources or sinks that shall be assigned into a particular domain. The handle of an assignable is used to refer to the 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.

For an interface specification to be well-formed there must be no two equal sources or sinks in the specification. In the abstract syntax specified by the BNF, we consider two sources (and sinks) equal if they are represented by identical strings. In the concrete syntax specified by the DTD, we consider two XML elements as equal if they have the same element name, the same attributes, the

same value for each attribute, and either are a leaf, or have the same number of children and for each child tag of one tag there is exactly one child tag of the other tag such that the two child tags are equal. We require that interface specifications are well-formed.

The concepts of sources and sinks are general ones and are thus independent of any particular programming language. However, the concrete sources and sinks of a program and their syntactic representations in a RIFL specification 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, Java bytecode, and Dalvik bytecode.

SOURCE, `source`	Section 5.1 (p. 17) for Java source code Section 6.1 (p. 30) for Java bytecode Section 6.1 (p. 30) for Dalvik bytecode
SINK, `sink`	Section 5.1 (p. 17) for Java source code Section 6.1 (p. 30) for Java bytecode Section 6.1 (p. 30) for Dalvik bytecode

3.2 Domains and Flow Relation

Security domains model different levels of confidentiality. 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 $(\texttt{d1}, \texttt{d2})$ explicitly appears in the list FLOW-RELATION or if $\texttt{d1} = \texttt{d2}$ 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`. Note that the reflexive closure need not be transitive, i.e. we allow intransitive flow relations.

3.3 Domain Assignment

The domain assignment classifies the sources and sinks from the interface specification into 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 pairs where the first element of each pair is the handle of an assignable 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. This means that each handle must appear exactly once as the 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.

A well-formed domain assignment maps the handle of each assignable to a domain. If the assignable refers to a single source or sink, then this source or sink is classified into the given domain. If the assignable refers to a category, then all sources and sinks grouped in the category, including those recursively grouped in sub-categories, are classified into the given domain.

3.4 Escape Hatches

Controlled declassification allows for exceptions to the strict information-flow restrictions specified by a flow relation and a domain assignment. Three dimensions of declassification have been identified and considered in the literature, viz. *what* information may be declassified (e.g. [20, 31]), *where* it may be declassified (e.g., [25, 26]), and *who* may declassify it (e.g., [21, 35]).

RIFL 1.1 allows for the specification of controlled declassification of information, specifically for the *what* aspect of declassification. Declassification can be expressed in a RIFL specification by means of a list of *escape hatches* [31]. An escape hatch specifies that certain information may flow to a given domain exceptionally, even though this might not be permitted by the flow relation. That is, an escape hatch expresses an exception to the flow relation. An escape hatch consists of a *declassification expression*, an explicit *reference point* [20], and the domain to which information may be declassified according to the escape hatch.

The abstract syntax for defining a list of escape hatches is specified by the following BNF and the concrete syntax is defined by the subsequent DTD.

BNF representation of syntactic elements

$$\begin{aligned} \text{HATCHES} &::= \epsilon \mid \text{HATCH} \mid \text{HATCH} :: \text{HATCHES} \\ \text{HATCH} &::= (\text{EXPRESSION}, \text{REFERENCE-POINT}, \text{DOMAIN}) \end{aligned}$$

XML DTD definition of syntactic elements

```
<!ELEMENT hatches (hatch)*>
<!ELEMENT hatch (expression, referencepoint)>
<!ATTLIST hatch to IDREF #REQUIRED>
```

According to the DTD, an escape hatch is expressed in a RIFL specification by an element of type `hatch`. This element has a mandatory attribute `to` that refers to the domain the information may be declassified to. The declassification expression and the reference point are specified by the two children elements of the types `expression` and `referencepoint`, respectively.

What information may be declassified by the escape hatch is determined by the declassification expression and the reference point. The reference point specifies a set of states during the execution of a program. The declassification expression associates each state during the execution of a program with a value. The information that may be declassified by a program according to a given escape hatch are the values of the declassification expression in the states that are specified by the reference point. The information that may be declassified according to a RIFL specification is any information that may be declassified according to at least one of the escape hatches of the specification.

The syntax for specifying declassification expressions and reference points depends on the concrete programming language. Hence, the non-terminals EXPRESSION (`expression`) and REFERENCE-POINT (`referencepoint`) are part of the language-specific modules. In RIFL 1.1, only the language-specific module for Java source code provides definitions for expressions and reference points.

These definitions are given in Section 5.2 (p. 24). Providing these definitions for instantiations of RIFL to other languages is left for the future.

An example of a RIFL specification that describes controlled declassification by means of escape hatches can be found in Section 5.3.2.

Remark 1. *RIFL specifications generally take a black-box view of the program for which information-flow requirements are being specified. The requirements are specified in terms of sources and sinks belonging to the interface between the program and its environment. Consequently, no knowledge of the code of a program is required to write a RIFL specification for it.*

In contrast to this, the specification of controlled declassification in RIFL may require to partially take a white-box view of the program. It may require to refer to parts of the program in the declassification expression that are not part of the interface to its environment, e.g., local variables. Furthermore, specifying the reference point of an escape hatch may require indicating a specific location in the program. Hence, knowledge of the program's code is required when writing a specification for it that involves controlled declassification.

This change of perspective is intentional, as declassification is used for controlled release of information in special cases. To ensure that only the intended information is declassified, knowledge of the code of the program is necessary.

3.5 Informal Semantics of a RIFL Specification

A complete RIFL specification expresses the following information-flow policy: Information is allowed to flow in a program from a source to a sink only if the source is assigned to a domain `d1` and the sink is assigned to a domain `d2` such that `d1` and `d2` are related by the flow relation. In addition, information about a value may flow to a sink assigned to the domain `d2` if the value is declassified by an escape hatch to a domain `d1` such that `d1` and `d2` are related by the flow relation – even if the value depends on sources classified into a domain `d3` such that `d3` and `d2` are not related by the flow relation.

The informal semantics of a RIFL specification is closely related to the intuition behind noninterference [13, 24]. That is, if no information is allowed to flow from certain sources to certain sinks according to the specification, then the output to these sinks must be independent of the input from these sources.

As an example, consider the RIFL specification in Figure 3. 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 names without any further details. The actual syntax of sources and sinks for Java source code is defined in Section 5 and for Java bytecode and Dalvik bytecode in Section 6. Note that the example in Figure 3 does not demonstrate the features of RIFL that allow for the specification of controlled declassification. Such an example can be found in Section 5.3.2.

The example specification expresses that no information about the location shall be stored to files or sent via an unencrypted HTTP connection, but location

```
<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>
```

Figure 3: An Example of a RIFL Specification

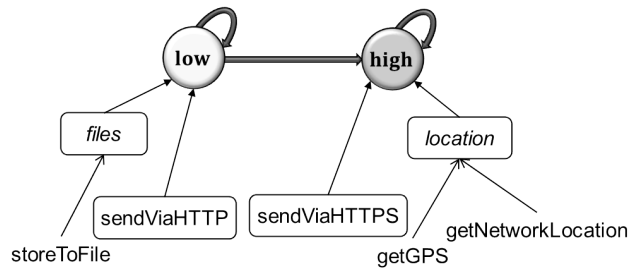

Figure 4: Visualization of the Example RIFL Specification

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 contains the sources `getGPS` and `getNetworkLocation`. The handle `fileshandle` refers to the category `files`. This category contains the sink `storeToFile`. The handle `HTTPhandle` refers to the sink `sendViaHTTP`. The handle `HTTPShandle` refers to the sink `sendViaHTTPS`.

The RIFL specification declares two domains, `high` and `low`. The flow relation specifies that information may flow from `low` to `high` and within each of these 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 sinks identified by the handle `HTTPShandle`, because both handles are assigned to the domain `high` and the flow relation is reflexive. Moreover, no information must flow from the sources identified by the handle `locationhandle` to sinks identified by the handles `fileshandle` and `HTTPhandle`.

Categories, sources, and sinks inherit the domains assigned to their parents. Thus, they also inherit the permitted flow of information. For example, information may flow from `getGPS` to `sendViaHTTPS`. This is because `getGPS` inherits the domain `high` from the category `location` (identified by `locationhandle`). Since the handle of `sendViaHTTPS` is assigned to `high`, too, and the flow relation is reflexive, information may flow from `getGPS` to `sendViaHTTPS`.

Similarly, since categories, sources and sinks inherit the domains assigned to their parents, they also inherit the constraints on the permitted flow of information. For example, no information must flow from `getGPS` to `storeToFile`. This is because `getGPS` inherits the domain `high` from the category `location` (identified by `locationhandle`) and `storeToFile` inherits the domain `low` from the category `files` (identified by `fileshandle`). Since the pair (`high`, `low`) is not in the flow relation and no escape hatch is defined, information flow from `getGPS` to `storeToFile` is not permitted.

Figure 4 visualizes the example policy. The circles represent the domains specified with the `domain` elements. The double arrows between the circles represent the permitted information flows according to the flow relation specified with the `flow` element. The boxes with rounded corners represent assignables. The assignables enclose the categories specified with the `category` elements (represented by text in italics) as well as the sinks specified with the `sink` elements (represented by regular text). The arrows with the open arrow head represent the grouping of sources and sinks specified with the `source` and `sink` elements (representeed by regular text) into categories. Finally, the arrows with the closed arrow head represent the domain assignment as specified by the `assign` elements.

Categories allow very concise definitions of the domain assignment even for larger specifications, because they group sources and sinks with respect to 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 Using RIFL

4.1 Dependencies and Refined Structure of RIFL

The bottom box in Figure 5 shows the dependencies between the different modules of RIFL. A solid arrow indicates a depends-on-relationship, i.e. the definition of the module at the end of the arrow depends on the module at the head of the arrow. Moreover, the modules in the lowest row are language-specific while the modules in the middle row are language-independent. As an example, the language-independent module "escape hatches" depends on the language-independent module "domain declarations" as well as the language-specific modules "expressions" and "reference points".

The top box in Figure 5 shows the dependencies between the different elements of a RIFL specification. Again, a solid arrow indicates a depends-on-relationship. Furthermore, a dotted arrow indicates an is-a-relationship, i.e. the element at the end of a dotted arrow is defined using the syntax of the module at the arrow head. For instance, a concrete "domain assignment" is defined using the syntax of the module "domain assignments".

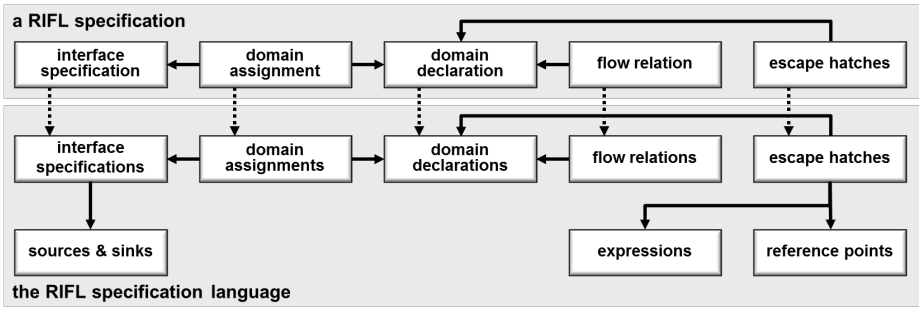

Figure 5: Dependencies in RIFL

Based on the dependence graph in Figure 5, the structure of RIFL, as presented in Figure 2, can be refined as shown in Figure 6. That is, the syntax of the language-independent modules, represented by the white boxes, builds on the syntax of the language-specific modules, represented by the light-grey boxes. Moreover, the definition of the elements of a RIFL specification for a

concrete program, represented by the dark-grey boxes, uses the syntax of the modules of RIFL as indicated by the arrows.

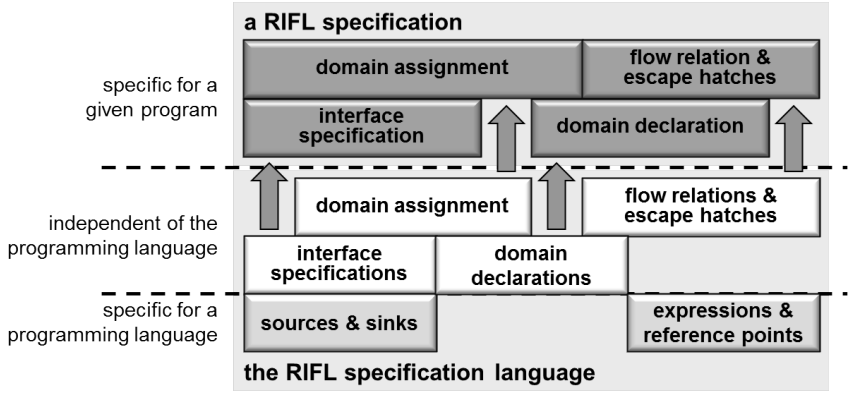

Figure 6: Refined Structure of RIFL

In Section 4.2, we explain how to define the language-specific modules for a programming language, i.e. the bottom layer in Figures 5 and 6. Moreover, in Section 4.3, we explain how to write a security requirement for a concrete program using RIFL, i.e. the top layer in Figures 5 and 6.

Remark 2. *As indicated by the empty grey spot on top of the module "expressions & reference points", RIFL 1.1 does not have language-independent means for defining expressions and reference points for escape hatches. This is due to the fact, that we first want to build up some experience to identify how a language-independent layer can be defined to be useful.*

Remark 3. *For sources, sinks, and domains, we introduced handles such that one can easily refer to those at multiple places in a RIFL specification. Currently, expressions and reference points do not have handles in RIFL, because we did not need this so far. In the future, we might introduce handles for expressions and reference points, if we identify some need for this at some point.*

4.2 Specializing RIFL for a Programming Language

A specialization of RIFL for a particular programming language provides concrete syntax for identifying sources and sinks in programs that are written in this language. Furthermore, it may provide syntax for specifying declassification expressions and reference points.

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 source code, and one might consider fields in the Android framework as sources and also as sinks in Dalvik bytecode.

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 source code program.

The final step is to define a syntax for identifying values that may be declassified and for identifying points in the execution of a program during which these values are determined. For instance, one could use JML expressions [17] to identify values that may be declassified in a Java source code program. Furthermore, one could use labels of program statements to identify points in the execution of a Java source code program.

4.3 Using RIFL for 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 confidentiality, 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, e.g., that information may only flow from low to high and within each domain. Define the flow relation by specifying a relation whose reflexive closure shall be the flow relation.

Specifying the interface of the program:

1. Declare the sources and sinks of the program that are relevant for the security requirement on the program.
2. Optionally: Structure the sources and sinks with respect to some notion of similarity using categories, e.g., group all API calls that send information to the network into one category.
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.

Specifying escape hatches:

If necessary: Specify exceptions to the flow relation by defining escape hatches that allow the controlled declassification of information.

We present example RIFL specifications that result from these steps for example programs written in Java in Section 5.3, and for an example program written in Dalvik in Section 6.3.

Remark 4. *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 of sources and sinks can then be used as the basis for creating the interface specifications in multiple RIFL specifications.*

Assume there is such a library of categorized sources and sinks. 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 1.1 for Java source code [14]. To make this section a self-contained manual for RIFL for Java source code, we introduce the complete language-specific syntax with explanations, even though there is a large overlap with the syntax for Java bytecode and Dalvik bytecode in Section 6.

5.1 Sources and Sinks for Java Source Code

In RIFL 1.1 for Java source code, the following kinds of sources can be specified:

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

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

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

Content and length of arrays The content and the length of arrays can be considered information sources, because input might be received in an array.

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.

Exceptions thrown by external methods If a method outside the program, e.g., in a library, is called by the program, it might terminate abnormally, i.e., throw an exception. Whether or not this happens can be considered an information source, since it can affect the control-flow of the program.

In RIFL 1.1 for Java source code, the following kinds of sinks can be specified:

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 of objects If a field of an object 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.

Static fields If a static field of a class in 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.

Content and length of arrays The content and the length of arrays can be considered information sinks, because the content and the length of the array may be 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.

Exception thrown by methods If a method of the program can be called from outside the program, whether or not that method throws an exception can be considered an information sink because this might be observed from outside the program.

5.1.1 Syntax

The abstract syntax for defining sources and sinks in Java source code programs 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 | EXCEPTION | ACCESSPATH
SINK	::=	PARAMETER | RETURN | FIELD | EXCEPTION
PARAMETER	::=	QNAME.METHOD@N
RETURN	::=	QNAME.METHOD@return
FIELD	::=	QNAME.FIELDNAME
EXCEPTION	::=	QNAME.METHOD@exception
ACCESSPATH	::=	PARAMETER.FIELDNAMES
METHOD	::=	METHODNAME(QNAMES)
QNAMES	::=	ϵ | QNAME | QNAME, QNAMES
FIELDNAMES	::=	FIELDNAME | FIELDNAME.FIELDNAMES

XML DTD definition of syntactic elements

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

The non-terminal QNAME is represented in the concrete syntax by the XML attributes `class`. The non-terminal and the attributes range over all possible fully qualified names as specified by the Java Language Specification [14, §6.7]. Examples of possible values are `double[]` and `package.Class`. The non-terminal FIELDNAME is represented in the concrete syntax by the XML attribute `name`. The non-terminal and the attribute range over identifiers as specified in [14, §3.8]. The non-terminal METHODNAME represents names of methods in a program. It ranges over identifiers as specified in [14, §3.8]. The non-terminal METHOD is represented in the concrete syntax by the XML attribute `method`. The non-terminal and the attribute represent names of methods in a program, including their signature. The possible values of the attribute `method` range over the possible values of the non-terminal METHOD, e.g. `method(java.lang.String)`. Note that in Java source code, method signatures do not include the return type of the method. The non-terminal N and

the values of the attribute `parameter` range over the natural numbers.

5.1.2 Informal Semantics

Each possible child element type of `source` and its analog in the abstract syntax identifies a source, i.e., a location in the code of a program where input is read. Analogously, each possible child element type of `sink` and its analog in the abstract syntax identifies a sink, i.e., a location in the code of a program where output is provided.

In the following, the meaning of each element type and each corresponding non-terminal of the BNF is explained.

`parameter` / PARAMETER

Consider the following XML element of the type `parameter` and its analog in the abstract syntax:

`<parameter class="`c`" method="`m`" parameter="`n`" />`

$c.m@n$

If $n > 0$, c is the fully qualified name of a class, and the class c defines an implementation of the method m, then this identifies the n-th formal parameter of the implementation of the method m defined by c as a source or sink. The formal parameters are enumerated beginning at 1.

If $n = 0$, c is the fully qualified name of a class, and the class c defines a non-static implementation of the method m, then this identifies the "this" pointer referencing the object on which the method is called as a source or sink.

In all other cases, the informal semantics of the element is undefined.

`returnvalue` / RETURN

Consider the following XML element of the type `returnvalue` and its analog in the abstract syntax:

`<returnvalue class="`c`" method="`m`" />`

$c.m$@return

If c is the fully qualified name of a class and the class c defines an implementation of the method m, then this identifies the return value of the implementation of the method m defined by the class c as a source or sink.

In all other cases, the informal semantics of the element is undefined.

`field` / FIELD

Consider the following XML element of the type `field` and its analog in the abstract syntax:

`<field class="`c`" name="`f`" />`

$c.f$

Fields:
If c is the fully qualified name of a class and the class c defines a static field f, then this identifies the static field f of the class c as a source or sink.

If c is the fully qualified name of a class and the class c defines a non-static field f, then this identifies the fields f of all instances of the class c as a source or sink.

Arrays:
If c is the qualified name of an array type and f is `"content"`, then this identifies the content of all arrays of the type c as a source or sink.

If c is the qualified name of an array type and f is `"length"`, then this identifies the length of all arrays of the type c as a source or sink.

In all other cases, the informal semantics of the element is undefined.

Remark 5. *Note that both elements of type* `field` *and of the type* `path` *can identify fields of objects as sources. The precedence between these two mechanisms for identifying fields as sources is clarified in Section 5.1.3.*

`exception` / EXCEPTION

Consider the following XML element of the type `exception` and its analog in the abstract syntax:

`<exception class="`c`" method="`m`" />`

$c.m$@exception

If c is the fully qualified name of a class and c defines an implementation of the method m, then this identifies the fact whether the implementation of the method m defined by the class c throws an exception as a source. Since we assume that observing the fact that an exception was thrown incurs observing the type of the exception, this also identifies the type of any exception thrown by the implementation of the method m defined by the class c as a source or sink.

In all other cases, the informal semantics of the element is undefined.

`path` / ACCESSPATH **(only for identifying sources)**

Consider the following XML element of the type `path` and its analog in the abstract syntax:

`<path>`
`  <parameter class="`c_0`" method="`m`" parameter="`n`" />`
`  <field class="`c_1`" name="`f_1`" />`

⋮
<field class="c_n" name="f_n" />
</path>

$c_0.m@n.f_1 \cdots f_n$

This identifies the field as a source that is accessed by evaluating the given access path directly after entering the method. Evaluating an access path means to dereference each field in the access path in the order of the access path starting at the object referred to by the parameter.

Remark 6. *Note that both elements of the type* `path` *and of the type* `field` *can identify fields of objects as sources. The precedence between these two mechanisms for identifying fields as sources is clarified in Section 5.1.3.*

5.1.3 Precedence in the Presence of Access Paths

Precedence of Access Paths over Class-wide Fields. In RIFL 1.1, fields can be specified as sources at the granularity of classes (using `field`) as well as at the granularity of single objects (using `path`). Specifications at the granularity of single objects have precedence over specifications at the granularity of classes. The rationale behind this definition of the precedence is that the specification at the granularity of classes can be seen as the default case for all objects of a given class. In contrast, the specification at the granularity of objects refers to the field of one particular object of a class and thus constitutes a special case that should take precedence over the default case.

As an example, consider a RIFL specification that declares the following two sources, the first source being at the granularity of classes and the second source being at the granularity of single objects:

```
<source>
  <field class="C" name="x" />
<source>
```

```
<source>
  <path>
    <parameter class="A" method="m(B)" parameter="1" />
    <field class="B" name="c" />
    <field class="C" name="x" />
  </path>
</source>
```

Both these sources identify the field `x` of instances of the class `C`. In this example, whenever the field `x` of an object of class `C` can be accessed by evaluating the access path `param.c.x` (where `param` is the name of the first parameter of `m()`) upon entry of method `m()`, the field is considered as a source according to the second specified source. If the field `x` of an object of class `C` cannot be accessed by evaluating the access path `param.c.x` upon entry of method `m()`, the field is considered as a source according to the first specified source.

Access Paths and Aliasing. In the following, we clarify the assignment of domains to sources specified by access paths in the presence of aliasing.

Using access paths one can have different source specifications referring to the same field of the same object due to aliasing. Consider, for instance, the following example program and source specification:

```
1 public int m(C a, C b){
2   return a.x;
3 }
```

```
<assignable handle="h1" >
  <source>
    <path>
      <parameter class="A" method="m(C, C)" parameter="1" />
      <field class="C" name="x" />
    </path>
  </source>
</assignable>
<assignable handle="h2" >
  <source>
    <path>
      <parameter class="A" method="m(C, C)" parameter="2" />
      <field class="C" name="x" />
    </path>
  </source>
</assignable>
```

If the method `m` is called with the same actual parameter for the two formal parameters, i.e. `a` and `b`, then the two sources `h1` and `h2` refer to the field `x` of the same object.

In such a case, the domain assignment might seem ambigous, as it can assign different security levels to the same field due to the different source specifications. However, the domain assignment is not really ambigous due to the following reason. When writing a RIFL specification, the security domains assigned to the sources carry the implicit assumption that any information provided via a source is at most as confidential as specified by the assigned security domain. Hence, if two parameters are aliased and two different security domains are assigned to a field via two different access paths, then the information in this field must be at most as confidential as specified by both security domains. In other words, it must be permissible that the information in the field flows to both security domains. In consequence, a lower security domain for a field specified with an access path is implicitly dominating a higher security domain specified for the same field with a different access path in case of aliasing. To make this more concrete, consider, for instance, the following flow relation and domain assignment for the aforementioned program:

```
<flowrelation><flow from="low" to="high" /></flowrelation>
<domainassignment>
  <assign handle="h1" domain="low" />
```

```
  <assign handle="h2" domain="high" />
</domainassignment>
```

This specification assigns the domain `low` to the source `h1`, i.e. the field `a.x`. Furthermore, this specification assigns the domain `high` to the source `h2`, i.e. the field `b.x`.

In case the method `m` is called with the same object `o` as the first and the second parameter, then the sources `h1` and `h2` are aliased. In consequence, `o.x` on the callee-site must be of a security level of at most `low`, because `o` is passed to the method for the parameter `a` and the security domain `low` is assigned to `a.x`. If this were not the case, the method call would not have been in accordance with the RIFL specification and, thus, the RIFL specification would not adequately capture the level of confidentiality of the inputs.

5.2 Escape Hatches for Java Source Code

In RIFL 1.1 for Java source code, expressions of the Java Modeling Language (JML) [17] are used as declassification expressions and labels of program statements are used as reference points. An example of the use of escape hatches for Java source code in a RIFL specification is given in Section 5.3.2.

Syntax. The abstract syntax for defining declassification expressions and reference points of escape hatches for Java source code is specified by the following BNF and the concrete syntax is defined by the subsequent DTD.

BNF representation of syntactic elements

EXPRESSION ::= JML-EXPRESSION

REFERENCE-POINT ::= CNAME.METHOD:LABEL

XML DTD definition of syntactic elements

```
<!ELEMENT expression (#PCDATA) >
<!ELEMENT referencepoint EMPTY>
<!ATTLIST referencepoint
          class CDATA #REQUIRED
          method CDATA #REQUIRED
          label CDATA #REQUIRED>
```

The DTD specifies that in RIFL for Java source code, an element of type `expression` may have exactly one child element, which must be character data. The non-terminal JML-EXPRESSION as well as the valid character data content of XML elements of the type `expression` ranges over the set of JML specification expressions [17, Section 12.2]. We require that the JML specification expression is well-defined at the reference point, i.e. all local variables in the expression must be in the scope at the reference point. The non-terminal CNAME as well as the possible values of the attribute `class` range over fully qualified names of classes and interfaces [14, §6.7], e.g. `package.Class`. The possible

values of the attribute `method` range over method signatures as specified by the non-terminal METHOD in Subsection 5.1, e.g. `method(java.lang.String)`, and identify a method in a given class. The non-terminal LABEL as well as the possible values of `label` range over identifiers as specified in [14, §3.8]. We require these identifiers to be names of labels of statements in a Java program.

Informal Semantics. The value of a JML expression in a given program state is defined as in the JML language specification [17, Section 12.4]. A reference point consisting of the name of a label in a specific method denotes all states in the execution of a program in which a statement labeled with this label in the given method is immediately about to be executed. Hence, the information that may be declassified by a Java source code program according to a given escape hatch are the values of the JML specification expression in all states in which a statement labeled with the reference point label is immediately about to be executed.

Note that, since any Java statement may be labeled [14, §14.7], reference points can identify arbitrary statements in a Java program.

5.3 Examples

5.3.1 Simple Password Program

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 }
```

RIFL Specification for the Simple Password Program. The RIFL 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`, since input can be obtained by the program from the Java library by reading this parameter, return values, and fields. The sinks of the program are the parameters of the method calls to `println` in line 8 and 10, and of the constructors of `InputStreamReader` and `BufferedReader`, since information may leave the program and enter the Java library by passing it to one of these methods. 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 `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.

5.3.2 Password Checker with Declassification

Consider the Java source code program in Listing 3 that contains a method for checking whether a given user ID and hash of a password is valid. The valid combinations of user IDs and password hashes are stored in a database implemented by a list of entries. The security requirement is that all user

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>
```

IDs and all password hashes shall be kept secret. As an exception, the only information about the user IDs and the hashes that the method may reveal is whether the given password hash is the correct one for the given user ID. On the implementation level, this means that there is an index in the array `db` such that the value of the field `user` of the instance stored at this index matches the given user ID and the value of the field `pwHash` of the instance matches the given password hash. This is an example of controlled declassification.

Listing 3: Example Java Program Requiring Declassification

```
1  package de.spp_rs3;
2
3  class DatabaseEntry {
4    public int user;
5    public int pwHash;
6  }
7
8  class PasswordChecker {
9    private DatabaseEntry[] db;
10
11   public boolean check(int user, int pwHash) {
12     declass:
13     for (int i = 0; i < names.length; i++) {
14       if (db[i].user == user && db[i].pwHash == pwHash)
15         return true;
16     }
17
18     return false;
19   }
20 }
```

RIFL Specification for the Password Checker. The RIFL specification in Listing 4 captures the desired information-flow requirement for the program in Listing 3.

The fields of the entries of the array `db` are declared as sources that are assigned to the domain `high`. Furthermore, the return value of the method `check` is declared as a sink that is assigned to the domain `low`. The flow relation does not allow any flow of information from `high` to `low`. Hence, the RIFL specification captures the requirement that the method may not reveal any information about the user IDs or password hashes.

The RIFL specification uses an escape hatch to specify the exception to this strict information-flow policy. The declassification expression

```
<expression>
  <![CDATA[
  \exists int i; db[i].user == user && db[i].pwHash == pwHash
  ]]>
</expression>
```

Listing 4: Example RIFL Specification for Java using Declassification

```
<riflspec>
  <interfacespec>
    <assignable handle="h">
      <category name="highFields">
        <source>
          <field class="de.spp_rs3.DatabaseEntry" name="user" />
        </source>
        <source>
          <field class="de.spp_rs3.DatabaseEntry" name="pwHash" />
        </source>
      </category>
    </assignable>
    <assignable handle="l">
      <sink>
        <returnvalue class="de.spp_rs3.PasswordChecker"
          method="check(int,int)" />
      </sink>
    </assignable>
  </interfacespec>
  <domains><domain name="high" /><domain name="low" /></domains>
  <flowrelation><flow from="low" to="high" /></flowrelation>
  <domainassignment>
    <assign handle="h" domain="high" />
    <assign handle="l" domain="low" />
  </domainassignment>
  <hatches>
    <hatch to="low">
      <expression>
        <![CDATA[
        \exists int i; db[i].user == user && db[i].pwHash == pwHash
        ]]>
      </expression>
      <referencepoint class="de.spp_rs3.PasswordChecker"
        method="check(int,int)" label="declass" />
    </hatch>
  </hatches>
</riflspec>
```

specifies that the fact whether the given user ID and password hash are a valid combination may be declassified, i.e., whether there is an entry in the array `db` such that the user ID and password hash stored in this entry match the given ones. (The `CDATA` section around the JML expression enables the use of special characters such as `&` or `<` in the declassification expression.) Furthermore, the reference point

```
<referencepoint class="de.spp_rs3.PasswordChecker"
  method="check(int,int)" label="declass" />
```

specifies points in the execution of the program s.t. the value of the declassification expression at these points is the value that may be declassified. In this example, these are the points in the execution whenever the statement labeled with the label `declass` in the method `check` is about to be executed. The placement of the `declass` label in the method `check` means that the value of the declassification expression directly after the execution of the method has started may be declassified. Finally, the attribute `to="low"` of the escape hatch specifies that this information may be declassified to the domain `low`. Hence, it is also valid for this information to flow to the return value of the method `check`, which is assigned to the domain `low`.

6 Specialization of RIFL for Java and Dalvik Bytecode

In this section, we present the language-specific module of RIFL 1.1 for Java bytecode [18] and Dalvik bytecode [6]. Due to the close similarity of the two bytecode languages in most aspects apart from their instruction set, the syntax and informal semantics described in this section are suitable for specifying information-flow requirements for programs in both languages. To make this section a self-contained manual for RIFL for Java bytecode and Dalvik bytecode, 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 Java and Dalvik Bytecode

In RIFL 1.1 for Java and Dalvik bytecode, the following kinds of sources can be specified:

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

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

Content and length of arrays The content and the length of arrays can be considered an information source, because input might be received in an array.

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.

Exceptions thrown by external methods If a method outside the program, e.g., in a library, is called by the program, it may terminate abnormally, i.e., throw an exception. Whether or not this happens can be considered an information source, since it can affect the control-flow of the program.

In RIFL 1.1 for Java and Dalvik bytecode, the following kinds of sinks can be specified:

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 of objects If a field of an object 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.

Static fields If a static field of a class in 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.

Content and length of arrays The content and the length of arrays can be considered an information sink, because the content and the length of the array may be 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.

Exception thrown by methods If a method of the program can be called from outside the program, whether or not that method throws an exception can be considered an information sink because this may be observed from outside the program.

6.1.1 Syntax

The abstract syntax for defining sources and sinks for Java and Dalvik bytecode 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 | EXCEPTION | ACCESSPATH
SINK ::= PARAMETER | RETURN | FIELD | EXCEPTION
PARAMETER ::= TYPDESC->METHOD@N
RETURN ::= TYPDESC->METHOD@return
FIELD ::= TYPDESC->FIELDNAME
EXCEPTION ::= TYPDESC->METHOD@exception
ACCESSPATH ::= PARAMETER.FIELDNAMES
METHOD ::= METHODNAME(TYPDESCS)TYPDESC
TYPDESCS ::= ϵ | TYPDESC | TYPDESC TYPDESCS
FIELDNAMES ::= FIELDNAME | FIELDNAME.FIELDNAMES

XML DTD definition of syntactic elements

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

The non-terminal TYPDESC is represented in the conrete syntax by the XML attributes `class`. The non-terminal and the attributes represent descriptors of types in a program. For Java bytecode, they range over all possible field descriptors [18, ch. 4.3.2]. For Dalvik bytecode, they range over all possible type descriptors [7]. Examples of possible values (for both Java and Dalvik bytecode) are `[D` and `Lpackage/Class;`.

The non-terminal FIELDNAME is represented in the concrete syntax by the XML attribute `name`. The non-terminal and the attribute represent names of

fields in a program. For Java bytecode, they range over the possible unqualified names [18, ch. 4.2.2]. For Dalvik bytecode, they range over simple names [7].

The non-terminal METHODNAME represents names of methods in a program. For Java bytecode, it ranges over the possible unqualified names as specified in [18, ch. 4.2.2]. For Dalvik bytecode, it ranges over simple names [7].

The non-terminal METHOD is represented in the concrete syntax by the XML attribute `method`. The non-terminal and the attribute represent names of methods in a program, including their signature. For both Java bytecode and Dalvik bytecode, the possible values of the attribute `method` range over the possible values of the non-terminal METHOD, e.g., `method(Ljava/lang/String;)V`. Note that in Java and Dalvik bytecode, method signatures include the return type of the method.

The non-terminal N is represented in the concrete syntax by the XML attribute `parameter`. The non-terminal and the attribute represent indices of formal parameters of methods. They range over the natural numbers.

6.1.2 Informal Semantics

Each possible child element type of `source` and its analog in the abstract syntax identifies a source, i.e., a location in the code of a program where input is read. Analogously, each possible child element type of `sink` and its analog in the abstract syntax identifies a sink, i.e., a location in the code of a program where output is provided.

In the following, the meaning of each element type and each corresponding non-terminal of the BNF is explained.

`parameter` / PARAMETER

Consider the following XML element of the type `parameter` and its analog in the abstract syntax:

`<parameter class="`c`" method="`m`" parameter="`n`" />`

c->m@n

If $n > 0$, c is the descriptor of a class, and the class c defines an implementation of the method m, then this identifies the n-th formal parameter of the implementation of the method m defined by c as a source or sink. The formal parameters are enumerated beginning at 1.

If $n = 0$, c is the descriptor of a class, and the class c defines a non-static implementation of the method m, then this identifies the "this" pointer referencing the object on which the method is called as a source or sink.

In all other cases, the informal semantics of the element is undefined.

`returnvalue` / RETURN

Consider the following XML element of the type `returnvalue` and its analog in the abstract syntax:

`<returnvalue class="`c`" method="`m`" />`

c->m@return

If c is the descriptor of a class and the class c defines an implementation of the method m, then this identifies the return value of the implementation of the method m defined by the class c as a source or sink.

In all other cases, the informal semantics of the element is undefined.

`field` / FIELD

Consider the following XML element of the type `field` and its analog in the abstract syntax:

`<field class="`c`" name="`f`" />`

c->f

Fields:
If c is the descriptor of a class and the class c defines a static field f, then this identifies the static field f of the class c as a source or sink.

If c is the descriptor of a class and the class c defines a non-static field f, then this identifies the fields f of all instances of the class c as a source or sink.

Arrays:
If c is the descriptor of an array type and f is `"content"`, then this identifies the content of all arrays of the type c as a source or sink.

If c is the descriptor of an array type and f is `"length"`, then this identifies the length of all arrays of the type c as a source or sink.

In all other cases, the informal semantics of the element is undefined.

Remark 7. *Note that both elements of type type* `field` *and of the type* `path` *can identify fields of objects as sources. The precedence between these two mechanisms for identifying fields as sources is clarified in Section 5.1.3.*

`exception` / EXCEPTION

Consider the following XML element of the type `exception` and its analog in the abstract syntax:

`<exception class="`c`" method="`m`" />`

c->m@exception

If c is the descriptor of a class and c defines an implementation of the method m, then this identifies the fact whether the implementation of the method m defined by the class c throws an exception as a source. Since we assume that observing the fact that an exception was thrown incurs

observing the type of the exception, this also identifies the type of any exception thrown by the implementation of the method m defined by the class c as a source or sink.

In all other cases, the informal semantics of the element is undefined.

`path` / ACCESSPATH **(only for identifying sources)**

Consider the following XML element of the type `path` and its analog in the abstract syntax:

```
<path>
  <parameter class="c0" method="m" parameter="n" />
  <field class="c1" name="f1" />
  ⋮
  <field class="cn" name="fn" />
</path>
```

c_0->m@n.$f_1 \cdots f_n$

This identifies the field as a source that is accessed by evaluating the given access path directly after entering the method. Evaluating an access path means to dereference each field in the access path in the order of the access path starting at the object referred to by the parameter.

Remark 8. *Note that both elements of the type* `path` *and of the type* `field` *can identify fields of objects as sources. The precedence between these two mechanisms for identifying fields as sources is clarified in Section 6.1.3.*

6.1.3 Precedence in the Presence of Access Paths

Precedence of Access Paths over Class-wide Fields. In RIFL 1.1, fields can be specified as sources at the granularity of classes (using `field`) as well as at the granularity of single objects (using `path`). Specifications at the granularity of single objects have precedence over specifications at the granularity of classes. The rationale behind this definition of the precedence is that the specification at the granularity of classes can be seen as the default case for all objects of a given class. In contrast, the specification at the granularity of objects refers to the field of one particular object of a class and thus constitutes a special case that should take precedence over the default case.

As an example, consider a RIFL specification that declares the following two sources, the first source being at the granularity of classes and the second source being at the granularity of single objects:

```
<source>
  <field class="LC;" name="x" />
<source>
```

```
<source>
  <path>
    <parameter class="LA;" method="m(LB;)V" parameter="1" />
    <field class="LB;" name="c" />
    <field class="LC;" name="x" />
  </path>
</source>
```

Both these sources identify the field `x` of instances of the class `C`. In this example, whenever the field `x` of an object of class `C` can be accessed by evaluating the access path `param.c.x` (where `param` is the name of the first parameter of `m`) at entry of method `m`, the field is considered as a source according to the second specified source. If the field `x` of an object of class `C` cannot be accessed by evaluating the access path `param.c.x` at entry of method `m`, the field is considered as a source according to the first specified source.

Access Paths and Aliasing. In the following, we clarify the assignment of domains to sources specified by access paths in the presence of aliasing.

Using access paths one can have different source specifications referring to the same field of the same object due to aliasing. Consider, for instance, the following example program and source specification:

```
1 public int m(C a, C b){
2   return a.x;
3 }
```

```
<assignable handle="h1" >
  <source>
    <path>
      <parameter class="LA;" method="m(LC;LC;)I" parameter="1" />
      <field class="LC;" name="x" />
    </path>
  </source>
</assignable>
<assignable handle="h2" >
  <source>
    <path>
      <parameter class="LA;" method="m(LC;LC;)I" parameter="2" />
      <field class="LC;" name="x" />
    </path>
  </source>
</assignable>
```

If the method `m` is called with the same actual parameter for the two formal parameters, i.e. `a` and `b`, then the two sources `h1` and `h2` refer to the field `x` of the same object.

In such a case, the domain assignment might seem ambigous, as it can assign different security levels to the same field due to the different source specifications. However, the domain assignment is not really ambigous due to the follow-

ing reason. When writing a RIFL specification, the security domains assigned to the sources carry the implicit assumption that any information provided via a source is at most as confidential as specified by the assigned security domain. Hence, if two parameters are aliased and two different security domains are assigned to a field via two different access paths, then the information in this field must be at most as confidential as specified by both security domains. In other words, it must be permissible that the information in the field flows to both security domains. In consequence, a lower security domain for a field specified with an access path is implicitly dominating a higher security domain specified for the same field with a different access path in case of aliasing. To make this more concrete, consider, for instance, the following flow relation and domain assignment for the aforementioned program:

```
<flowrelation><flow from="low" to="high" /></flowrelation>
<domainassignment>
  <assign handle="h1" domain="low" />
  <assign handle="h2" domain="high" />
</domainassignment>
```

This specification assigns the domain `low` to the source `h1`, i.e. the field `a.x`. Furthermore, this specification assigns the domain `high` to the source `h2`, i.e. the field `b.x`.

In case the method `m` is called with the same object `o` as the first and the second parameter, then the sources `h1` and `h2` are aliased. In consequence, `o.x` on the callee-site must be of a security level of at most `low`, because `o` is passed to the method for the parameter `a` and the security domain `low` is assigned to `a.x`. If this were not the case, the method call would not have been in accordance with the RIFL specification and, thus, the RIFL specification would not adequately capture the level of confidentiality of the inputs.

6.2 Escape Hatches for Java and Dalvik Bytecode

RIFL 1.1 does not yet provide a syntax for specifying escape hatches in Java bytecode and Dalvik bytecode. This feature is left for future versions of RIFL.

6.3 Examples

6.3.1 Simple Password Program in Dalvik Bytecode

Consider the Dalvik bytecode in Listing 5 that implements a simple password prompt. The mnemonic code in the listing was created by compiling the Java source code program from Listing 1 and then using `dexdump` on the resulting Dalvik bytecode binary file. It 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.

Listing 5: 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
```

RIFL Specification for the Simple Password Program. The XML specification in Listing 6 captures the desired information-flow requirement for the program in Listing 5.

The sources of the program in Listing 6 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`, since input can be obtained by the program from the Android framework by reading this parameter, return values, and fields. The sinks of the program are the formal parameters of the constructors of `InputStreamReader` and `BufferedReader` and the parameter of the method calls to `println` at positions 0010 and 0019, since information may leave the program and enter the Android framework by passing it to one of these methods. 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;` corre-

Listing 6: 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;" name="in" /></source>
        <source><field class="Ljava/lang/System;" name="out" /></source>
      </category>
    </assignable>
    <assignable handle="envoutputhandle">
      <category name="envoutput">
        <sink>
          <parameter class="Ljava/io/InputStreamReader;"
           method="&lt;init&gt;(Ljava/io/InputStream;)V" parameter="1" />
        </sink>
        <sink>
          <parameter class="Ljava/io/BufferedReader;"
           method="&lt;init&gt;(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>
```

sponds 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. [32, 33], 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 [28, 2] 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 [5] 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 owned 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) [10] 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) [23] 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 [22]. 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.

JML. In [34] and further development [4], the authors introduce a specification language for information-flow contracts in JML. An example specification in the style of [4] is given below.

```
/*@ \determines \result, this.x+this.y \by a */
int someMethod(int a, int b) {...}
```

The specification consists of two lists of expressions. The first list, \result, this.x+this.y, states that the return value and the sum of the fields x and y must hold low information after the execution of someMethod. The second list, a, states that via parameter a low information is provided. As the example indicates, the specification language allows very precise specification and declassification of information. The reason is: nearly arbitrary JML expressions are allowed in the two lists of expressions.

In [16], the authors build on JML-style specification to provide non-interference specifications for component-based systems and sketch a specification language for JavaEE programs. Information-flow specification for individual services of a component is supported via a notion of dependency clusters, representable using lists of expressions similar to those in [34]. Verified dependency clusters can be used as building blocks to obtain complex, domain-driven security specifications for entire components and component-based systems. While dependency clusters support similar precision as JML specifications, escape hatches cannot be expressed – a design decision allowing compositionality of non-interference and specifications.

8 Conclusion

In this report, we defined RIFL 1.1, 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, Java bytecode, and Dalvik bytecode. We have also illustrated how these specializations can be used for specifying information-flow requirements for concrete programs.

In addition to RIFL 1.0, the previous version of RIFL, RIFL 1.1 supports the specification of controlled declassification by means of escape hatches, provides an additional specialization for Java bytecode, allows for specifying the occurrence of exceptions as sources and sinks, and allows for specifying fields as sources more precisely by means of access paths.

Various parts of RIFL 1.1 are already supported by the RSCP security analyser and its integration into Cassandra [19, 8], Joana [15], JoDroid [27], and KeY [1]. Moreover, an earlier version of RIFL is supported by SuSi [29]. We believe that RIFL can also be of use for other information-flow analysis tools and encourage its adoption and contributions to its further development by the community. Moreover, RIFL shall serve as a basis for using different tools in combination. For example, complex information-flow analysis problems could be tackled be dividing them and solving each part using a tool that is especially suitable for this particular part, using RIFL as the policy language common to all involved tools.

Currently, a library of example programs with associated information-flow requirements specified using RIFL is being compiled. In this endeavour, RIFL serves as a machine-readable language for information-flow requirements that enables automatic testing of different tools on the collected example programs.

In the future, we plan to specialize RIFL for further programming languages, e.g., for JavaScript and C. Finally, we might provide formal semantics for RIFL. 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 support formal semantics without losing flexibility, 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 who have already contributed to RIFL 1.0 and earlier versions. In particular, we would like to thank Tobias Hamann, who has helped in the creation of the library of example programs with associated RIFL specifications. This work was funded by the DFG under the projects DeduSec (BE 2334/6-2/3), IFC4MC (Sn 11/12-2/3), MORES (Hu 737/5-2/3), and RSCP (MA 3326/4-2/3) in the priority program RS^3 (Reliably Secure Software Systems, SPP 1496).

References

[1] 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.

[2] 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.

[3] John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Report on the Algorithmic Language ALGOL 60. *Numerische Mathematik*, 2(1):106–136, 1960.

[4] Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software – extended version –. Technical Report 2013-14, Department of Informatics, Karlsruhe Institute of Technology, 2013.

[5] Niklas Broberg, Bart van Delft, and David Sands. Paragon for Practical Programming with Information-Flow Control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems*, pages 217–232, 2013.

[6] Dalvik bytecode, `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`. Accessed on July 20, 2017.

[7] Dalvik Executable format, `https://source.android.com/devices/tech/dalvik/dex-format`. Accessed on July 20, 2017.

[8] Sarah Ereth, Steffen Lortz, and Matthias Perner. Confidentiality for Android Apps: Specification and Verification. *it – Information Technology*, 56(6):288–293, 2014.

[9] Sarah Ereth, Heiko Mantel, and Matthias Perner. 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, TU Darmstadt, 2014.

[10] Michael D. Ernst, René Just, Suzanne Millstein, Werner M. Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, 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, 2014.

[11] Extensible Markup Language (XML) 1.0 (Fifth Edition), `http://www.w3.org/TR/2008/REC-xml-20081126/`. Accessed on July 20, 2017.

[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] James Gosling, 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/`. Accessed on July 20, 2017.

[15] 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.

[16] Simon Greiner, Martin Mohr, and Bernhard Beckert. Modular verification of information-flow security in component-based systems. In *Software Engineering and Formal Methods: 15th International Conference*, 2017. Accepted for publication.

[17] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Mller, Joseph Kiniry, Patrice Chalin, and Daniel M. *JML Reference Manual*. `http://www.jmlspecs.org/`. Accessed on July 20, 2017.

[18] Tim Lindholm, Frank Yelim, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification - Java SE 8 Edition*. `http://docs.oracle.com/javase/specs/jvms/se8/html/`. Accessed on July 23, 2017.

[19] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, 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.

[20] Alexander Lux and Heiko Mantel. Declassification with Explicit Reference Points. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, pages 69–85. Springer, 2009.

[21] Alexander Lux and Heiko Mantel. Who Can Declassify? In *Post-Proceedings of the 5th Workshop on Formal Aspects in Security and Trust*, pages 35–49. Springer, 2009.

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

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

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

[25] Heiko Mantel and Alexander Reinhard. Controlling the What and Where of Declassification in Language-Based Security. In *Proceedings of the 16th European Symposium on Programming*, pages 141–156. Springer, 2007.

[26] Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*, pages 129–145. Springer, 2004.

[27] Martin Mohr, Jürgen Graf, and Martin Hecker. JoDroid: Adding Android Support to a Static Information Flow Control Tool. In *Proceedings of the 8th Working Conference on Programming Languages*, pages 140–145, 2015.

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

[29] Siegfried Rasthofer, 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.

[30] Reliably Secure Software Systems (RS^3), `http://www.spp-rs3.de`. Accessed on July 20, 2017.

[31] Andrei Sabelfeld and Andrew C. Myers. A Model for Delimited Information Release. In *Proceedings of the 2nd Mext-NSF-JSPS International Symposium on Software Security - Theories and Systems*, pages 174–191, 2003.

[32] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

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

[34] Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of java programs without approximations. In *Formal Verification of Object-Oriented Software: International Conference*, pages 232–249, 2012.

[35] Steve Zdancewic and Andrew C. Myers. Robust Declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.