A Uniform Information-Flow Security Benchmark Suite for Source Code and Bytecode

Tobias Hamann[1]✉, Mihai Herda[2], Heiko Mantel[1], Martin Mohr[2], David Schneider[1], and Markus Tasch[1]

[1] Department of Computer Science, TU Darmstadt, Germany
{hamann, mantel, schneider, tasch}@mais.informatik.tu-darmstadt.de
[2] Department of Informatics, Karlsruhe Institute of Technology, Germany
{herda, martin.mohr}@kit.edu

Abstract. It has become common practice to formally verify the correctness of information-flow analyses wrt. noninterference-like properties. An orthogonal problem is to ensure the correctness of implementations of such analyses. In this article, we propose the benchmark suite IFSpec, which provides sample programs for checking that an information-flow analyzer correctly classifies them as secure or insecure. Our focus is on the Java and Android platforms, and IFSpec supports Java source code, Java bytecode, and Dalvik bytecode. IFSpec is structured into categories that address multiple types of information leakage. We employ IFSpec to validate and compare four information-flow analyzers: Cassandra, Joana, JoDroid, and KeY. IFSpec is based on RIFL, the RS[3] Information-Flow Specification Language, and is open to extensions.

1 Introduction

Research on information-flow security aims at end-to-end security guarantees regarding confidentiality and integrity. Information-flow guarantees can be formalized based on the idea of noninterference, using the original property [20] or variants of it [30]. These guarantees go beyond the ones provided by access control: regarding confidentiality, for instance, attackers are not only prevented from accessing secrets directly, but also from deducing sensitive information from the observations they make during program runs. The field of information-flow security originated already in the late seventies and early eighties [14, 17, 18, 20, 31, 36]. To date, information-flow analysis tools range from scientific prototypes [4, 6, 21, 27, 32], to being part of commercial products [1, 2].

Albeit it is clear that benchmark suites are catalyzers for technical progress in tool development [38], little effort has gone into the development of benchmark suites for information-flow analysis tools. In many other areas of Computer Science, the use of benchmark suites has become common practice, e.g., in hardware/software performance research [23, 24], compiler research [16], SAT/SMT solving [25], theorem proving [42], and model checking [26, 35]. Such benchmark suites enable the comparison of developed tools and techniques, and provide a basis for fostering exchange between research groups and projects.

In this article, we present the novel benchmark suite IFSpec[1] for benchmarking information-flow analysis tools targeting source code and bytecode for the Java and Android platforms. Each sample program in IFSpec is provided in Java source code, Java bytecode, and Dalvik bytecode. IFSpec is designed to cover a broad range of different types of information leakage commonly found in real-world programs. By providing all samples for three different language levels in a uniform fashion, IFSpec facilitates the evaluation and comparison of information-flow analysis tools developed for these language levels on a common set of samples, fostering the transfer of innovation across language levels.

We are aware of only two benchmark suites that have already been used to evaluate information-flow analysis tools: SecuriBench Micro [41] and DroidBench [6, 19]. The samples of SecuriBench Micro were originally developed to benchmark web application security analyses, but they can also be interpreted from an information-flow security perspective and have been used to evaluate information-flow analyzers targeting Java source code (e.g. [6, 45]). DroidBench was developed to compare the effectiveness of taint-analysis tools targeting Dalvik bytecode. With IFSpec, we aim to provide a benchmark suite for the evaluation of information-flow analysis tools for multiple language layers of the Java and Android platforms on a uniform set of samples.

Using IFSpec, we evaluate four information-flow analyzers. One of them targets Java source code, two Java bytecode, and one Dalvik bytecode. We present insights on the soundness and precision for each of the evaluated tools. As a side effect, our evaluation shows that IFSpec is indeed suitable for evaluating and comparing information-flow analyzers for both source code and bytecode.

In detail, our two main contributions are the following:

- Our first contribution is IFSpec, a machine-readable benchmark suite for information-flow analysis tools that target the Java virtual machine or the Android platform. For each sample, a corresponding security policy is specified in a uniform fashion using *RIFL*, the *RS*^{3} *Information-Flow Specification Language* [8]. IFSpec is open for extensions, and we present three such extensions in this article (subsuming the benchmarks from SecuriBench Micro and DroidBench and making them more accessible). IFSpec enables the evaluation of information-flow analyzers in a fully automated fashion.
- Our second contribution is an evaluation of four information-flow analysis tools for multiple language levels of the Java and Android platforms using the IFSpec benchmark suite: Cassandra [27], Joana [21], JoDroid [32], and KeY [4]. Each of these four tools is built on solid theoretical foundations and is designed to enforce specific, formally defined noninterference-like security properties. We demonstrate how IFSpec can be used to assess the soundness of such tool implementations. In addition to presenting our results, we discuss how the trade-off between correctness and precision has been addressed in the implementations of the evaluated tools.

[1] The benchmark suite, including all samples, evaluation results, the benchmarked tools, information how to run information-flow analyzers on IFSpec, and how to contribute to IFSpec is available under `www.spp-rs3.de/IFSpec`.

2 RIFL in a Nutshell

The RS^3 Information-Flow Specification Language (RIFL) is a language for specifying information-flow policies [8]. The machine-readable syntax of RIFL is formally defined in an XML format. It enables the definition of information-flow policies by specifying restrictions on the permitted information flow between given security domains. The association of these security domains with concrete sources and sinks of information is realized by function mappings.

A RIFL specification capturing information-flow policies for a particular program consists of four aspects: the interface of the program (in terms of sources and sinks of information), the collection of security domains, the association of each source or sink of information with a security domain, and the specification of how information may flow between security domains.

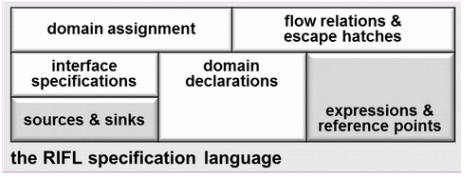

Fig. 1. RIFL language modules [8].

Each of these aspects is specified in one of the individual language modules of RIFL (represented by boxes in Fig. 1). RIFL comprises language-specific and language-independent modules. Currently, RIFL supports the specification of information-flow policies for Java source code, Java bytecode, and Dalvik bytecode, which is sufficient for the purposes of this article. Note, however, that RIFL can be extended to support additional target languages like, e.g., C/C++ or the LLVM IR. In Fig. 1, white boxes represent language modules that are language-independent, while gray boxes represent modules that are language-specific. The clear separation of the language-independent and the language-specific parts in RIFL has two benefits: Firstly, information-flow policies can be expressed and understood at a high level, independently from the details of a specific programming language. Secondly, a RIFL policy can be adapted to multiple target languages by adapting the language-specific parts. This separation of language-specific and language-independent aspects was, indeed, beneficial in our construction of IFSpec. For each sample, large parts of the policy specification could be shared for Java source code, Java bytecode and Dalvik bytecode.

RIFL aims at compatibility with information-flow analysis tools that are based on distinct security semantics. Hence, RIFL cannot have a predefined formal security semantics. Naturally, one can interpret given RIFL specifications under a chosen formally defined security semantics. In this article, we interpret information-flow policies with respect to the security semantics that are enforced by the four individual tools that we evaluated. Since most existing approaches for the specification of information flow policies are highly tool and target-language dependent, we choose RIFL for the specification of the IFSpec benchmarks.

3 IFSpec Benchmark Suite

IFSpec consists of samples that showcase information-flow vulnerabilities in programs for the Java and Android platforms. Such vulnerabilities can be of

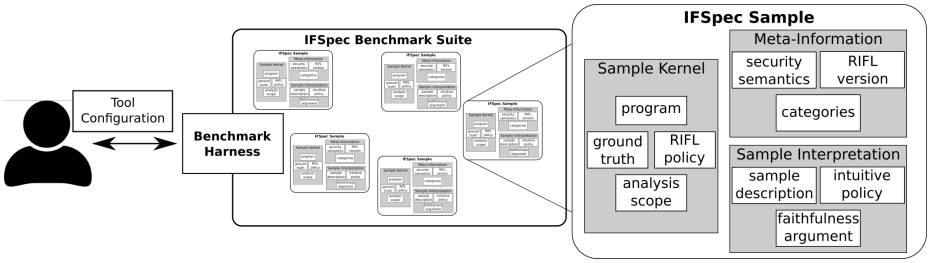

Fig. 2. IFSpec Architecture

different kinds, e.g., involving direct information leaks, or implicit information leaks that are, e.g., dependent on exceptional program behavior. The samples in IFSpec cover a broad range of different kinds of information-flow vulnerabilities.

Fig. 2 shows the architecture of IFSpec. Users of the benchmark suite interact with a benchmark harness provided by IFSpec. This harness, provided a tool configuration for the benchmarked tool, enables an automated benchmarking on the IFSpec samples. Each sample in IFSpec is provided in a machine-readable format for Java source code, Java bytecode and Dalvik bytecode.

3.1 Syntax and Security Semantics

Sample Format. As illustrated in Fig. 2, all samples in IFSpec share a uniform format with two mandatory parts (the *sample kernel* and the *sample meta-information*) and one optional part (the *sample interpretation*). The sample kernel provides all information that is used when benchmarking an information-flow analysis tool in a machine-readable format and, thus, enables to automate the benchmarking of the tool. It consists of four parts: The first part is the program to be analyzed in the target language. The second part is the information-flow policy for the program given as a RIFL specification for the target language. The third part is a specification of the analysis scope that declares which methods are part of the program's environment, and which methods can be called by the program's environment. The fourth part is the sample's ground truth, i.e. the expected classification of the sample as secure or insecure.

The sample meta-information provides the tags associated with each sample that describe categories of information-flow, the minimal RIFL version a benchmarked tool must support to parse the RIFL specification, and the security semantics considered when classifying a sample as secure or insecure.

The optional sample interpretation in IFSpec's sample format provides a detailed description of the program and its functionality, a description of the intuitive security requirement for the program, and a faithfulness argument. The faithfulness argument substantiates why the RIFL specification captures the intuitive security requirement for the program. With this information the sample interpretation supports users in understanding the implications of successfully analyzing a sample or failing to do so.

Security Semantics. For the classification of the samples in IFSpec, we consider four formal security properties: *Termination-Insensitive Noninterference for the Abstract Dalvik Language (TIN-ADL)* [27], *Sequential Noninterference (SN)* [10], *Probabilistic Noninterference (PN)* [10], and the *flow*-predicate* [9].

We limit ourselves to these four properties because they were sufficient to benchmark the information-flow analysis tools considered in Section 4. That is, these security properties are enforced by at least one of the benchmarked tools. The security property TIN-ADL is enforced by Cassandra [27], SN (resp. PN) is enforced by Joana as well as JoDroid for sequential (resp. concurrent) programs [10], and the flow*-predicate is enforced by KeY [9].

3.2 Core Samples of IFSpec

IFSpec provides 80 samples that contain test cases for information-flow analysis tools. These samples have been contributed over a period of two years by over 20 researchers from the area of information-flow security. The vast majority of these contributors are not developing information-flow analysis tools themselves. Hence, they had no interest in tailoring the samples to the current technical state of their tools, but were rather interested in identifying current limitations of information-flow analysis tools. This adds to our confidence that our samples avoid the pitfall of overfitting IFSpec to existing tools.

Categorization. Each sample in IFSpec is categorized with respect to predefined types of information flow. The categories cover a wide range of information leakage from simple flows, e.g., using direct assignment, to more sophisticated flows caused by advanced language features, e.g., involving Java reflection. In detail, the samples in IFSpec are categorized using the tags shown in the table in Fig. 3. IFSpec contains samples for explicit information flow and for implicit information flow via the control flow of sample programs.

Simple explicit information flows (e.g., via direct assignments or method calls) are covered by the *simple* tag. Information flows that are caused by branching over a secret-dependent conditional are covered by the *high-conditional* tag. The other six tags in Fig. 3 cover more sophisticated types of information flows. For five categories, we provide samples covering both, explicit and implicit information flows. The other three categories contain samples that are specific to either explicit or implicit flows. Overall, IFSpec contains 46 samples for explicit and 34 samples for implicit information flow. The categories present in IFSpec contain samples that are relevant for evaluating information-flow analyzers.

Note that samples in IFSpec can be categorized into multiple tags at the same time. This enables the expression of samples that combine multiple types of information leakage covered by our tags, like, e.g., samples that contain an implicit flow via exceptional control flow that is dependent on array contents.

3.3 Benchmark Harness

As part of IFSpec, we provide a benchmark harness for automatically benchmarking tools on IFSpec's samples. This benchmark harness is a configurable Python script that generates a report on the benchmark results. Additionally,

tag	#samples	types of information flow covered	explict flows	implicit flows
simple	18	simple information flow not covered by tags below	×	
high-conditional	11	information flow via secret-dependent conditionals		×
arrays	12	information flow involving array length and content	×	×
class-initializers	7	information flow involving class initializers	×	×
exceptions	11	information flow via exception handling		×
library	7	information flow involving library calls	×	×
aliasing	11	information flow involving object aliasing	×	×
reflection	7	information flow involving reflection	×	×
		total	46	34

Fig. 3. Sample tags and their distribution in IFSpec

the benchmark harness offers configuration options that enable, for instance, a selective benchmarking specified by a list of tags, or concrete sample names.

For the tool-specific configuration, users of IFSpec instantiate the benchmark harness for their information-flow analysis tool by setting four JSON options: the command that runs an analysis on the current sample, the outputs by a tool when the benchmarked tool classifies a sample as secure or insecure, respectively, and the target language of benchmarked the tool. Providing such a tool-specific configuration for an information-flow analysis tool that implements a RIFL frontend suffices to automatically benchmark this tool using IFSpec.

The report on the benchmark results consists of two parts. The first part is a detailed overview of the samples where the tool output matched the expected outcome (secure or insecure) and the samples where the actual result of the tool did not match the expected outcome. The second part provides the overall recall and precision for all samples and the recall and precision for each tag separately. Using the benchmark harness reduces the effort for benchmarking information-flow analysis tools by minimizing the necessary setup. In addition, it provides the benchmark results in suitable format for further investigations.

3.4 Extensions of the IFSpec Benchmark Suite

In addition to the core samples of IFSpec, we provide three extensions to the benchmark suite: an extension covering samples from the domain of web vulnerability testing, an Android-specific extension covering peculiarities of the Android platform, and a declassification extension specific to Java source code.

Web Vulnerability Extension. IFSpec subsumes SecuriBench Micro [41], a benchmark suite for web application security analyses. Concretely, IFSpec provides 152 samples that have been derived from the original 122 samples of SecuriBench Micro. The samples contained in SecuriBench Micro are small web applications that contain potential vulnerabilities caused by unchecked user input. These vulnerabilities can be interpreted as a potential information leak, i.e., each application contains a source where secrets from the environment enter the application and a sink where the secret might be leaked. In fact, SecuriBench Micro has been used to evaluate information-flow analysis tools [6]. For

SecuriBench Micro samples that contain occurrences of both, leaking and non-leaking calls of methods that can potentially leak secret information, we derive two samples in IFSpec: the unchanged insecure sample, and a secure sample obtained by deleting all method calls that are actually leaking secret information. Our integration of SecuriBench Micro into IFSpec makes all SecuriBench Micro samples available for Java source code, Java bytecode and Dalvik bytecode. Moreover, the samples now contain machine-readable specifications of the security property.

Note that in our integration we identified seven samples where the classification into secure or insecure in SecuriBench Micro did not match our interpretation from an information-flow perspective. For the integration of these samples into IFSpec, we adjusted the classification of these samples as (in)secure accordingly. An overview of these samples is provided as part of the IFSpec artifact.

Android Vulnerability Extension for Dalvik Bytecode. IFSpec subsumes all 119 samples from DroidBench ([6, 19]), a benchmark suite for taint-analysis tools targeting the Android platform. The samples of DroidBench are small Android applications, both secure and insecure. These samples do not contain a formal, machine-readable specification of the information-flow policy, but provide a human readable specification. This specification consists of a sample description, a declaration of sources and sinks, and the number of leaks present in the sample. By providing ground truths and a specification of the information-flow policies, our extension makes all samples of DroidBench accessible in a machine-readable format using the benchmark harness of IFSpec.

Declassification Extension for Java Source Code. We provide seven samples utilizing the support of *escape hatches* [37] in RIFL 1.1 for Java source code, as language-specific extension for Java source code. In the samples, specific information may be declassified at explicit program points. Declassification is one of the research problems extensively studied in the research area of information-flow security (e.g., [28, 29, 33, 37]). Our extension shall be a first step towards catalyzing technical progress in tool support for declassification. For details how to utilize escape hatches in RIFL, we refer the interested reader to [8].

4 Benchmarking with IFSpec

We use IFSpec to compare four information-flow analysis tools developed for the three target languages of IFSpec: Cassandra [27] (Dalvik bytecode), Joana [21] (Java bytecode), JoDroid [32] (Dalvik bytecode), and KeY [4] (Java source code). We evaluate each of these tools on the core samples of IFSpec, as well as on the web vulnerability extension that subsumes SecuriBench Micro. IFSpec enables us to evaluate these tools on a common set of samples. In addition, we selectively evaluate tools on the other two extensions of IFSpec (cf. Section 3.4).

4.1 Benchmarked Tools

Cassandra [27] is an Android app store that integrates an information-flow analysis. It allows end users of mobile devices to specify their security requirements and to check whether applications comply with their requirements before these applications are installed. To this end, Cassandra implements a security type system for Dalvik bytecode that is proven sound with respect to a formal notion of noninterference. Within methods, the type system is flow-sensitive.

Cassandra does not analyze methods from third-party libraries or the Android standard library. Instead, it uses manually provided method signatures to specify the information flows in library methods. If Cassandra encounters a call to a library method for which no method signature is defined, it has no knowledge of the effects of the call. In such cases, Cassandra cannot ascertain whether the application is secure or insecure.

JOANA [21, 22] is an information-flow analysis tool for full Java bytecode. It leverages program dependence graphs (PDGs), a language-independent and flow-sensitive representation of a program's dependencies, and then uses slicing – a form of graph reachability – on the PDG to determine whether a given source may influence a given sink. This kind of check guarantees noninterference for sequential programs [44] and for concurrent programs, there is an extension which guarantees a form of probabilistic noninterference [10, 12].

JOANA incorporates library code in its analysis, so in principle all library code that may potentially be used is required. For this purpose, JOANA contains method stubs of the Java Standard Libraries. Most importantly, these method stubs provide implementations for some heavily used native methods. JOANA provides method stubs for different releases of the Java Standard Libraries, in particular for Java 1.4 and Java 1.5.

JODROID [32] is a variant of JOANA which provides a front-end for the analysis of Dalvik bytecode and in particular Android applications. Like JOANA, JODROID generates a PDG from a given Android application but can additionally deal with Android specifics like Android's message passing mechanism or the fact that an Android application consists of multiple entry points invoked by the Android framework in certain patterns (the Android Activity Lifecycle [5]).

For JODROID, Android SDK Platform packages [3] are used as method stubs. These packages are used to compile an Android App for a specific API version. They contain stub implementations of the respective API methods which throw an exception if they are called. Hence, using an Android SDK Platform package potentially causes unsound assumptions as it does not contain sound information about the relation between method inputs and outputs. Note however that it is possible to run JODROID with more proper implementations of the Android API.

KeY [4] is a software verification tool based on deductive theorem proving for Java programs annotated with an extension of the JML specification language. KeY supports the specification and verification of the noninterference property. For the evaluation we interpret the case in which a proof was not found as KeY reporting the program to be insecure.

True Positive (TP) sample contains leak, tool reports leak	**False Positive (FP)** sample contains no leak, tool reports leak
True Negative (TN) sample contains no leak, tool reports no leak	**False Negative (FN)** sample contains leak, tool reports no leak

Fig. 4. Classifications of Possible Benchmarking Results for a Sample

For handling library methods, KeY uses method contracts or the source code. Method contracts are formally proven dependencies between the method inputs and outputs. If method contracts are not available, the source code of the method is included in the analysis. KeY's handling of library calls cannot lead to unsound results because all assumptions about the library methods are formally justified. However, providing formally proven method contracts is difficult.

Sound Overapproximation of Benchmarking Results. When applying one of the benchmarked tools to a sample from the benchmark suite, it might happen that the output of the tool does not allow for a clear statement about the (in)security of the sample to be made. The sound option of interpreting such outputs is that the tool reports the sample as insecure, since the security of the sample cannot be established. We chose this option for interpreting the benchmarking results of the four tools. For Cassandra, a clear statement cannot be made when a signature for a library method is missing, when a method is called in a context in which control flow may depend on secret information, or when the analyzed program throws an exception. For JOANA (and also its variant JODROID), a clear statement about the (in)security of a sample cannot be made when the tool crashes on the sample. KeY cannot make statements about sample programs that contain library calls for which neither a stub with corresponding method contract nor the source code is provided.

4.2 Terminology and Metrics for Benchmarking

For the evaluation of the four tools, we record the true positives, true negatives, false positives, and false negatives. Furthermore, for each tool, we compute the recall and precision on the samples used for benchmarking.

A *true positive (TP)* means that a tool correctly reports an information leak in an insecure sample. A *true negative (TN)* means that a tool correctly reports the absence of information leaks in a secure sample. A *false positive (FP)* means that a tool incorrectly reports an information leak in a secure sample. False positives indicate imprecision of a tool. A *false negative (FN)* means that a tool does not report any information leak in an insecure sample. False negatives indicate unsoundness of a tool. We summarize these terms in Fig. 4.

The *recall* of a tool is computed from the number of true positives and false negatives in the benchmarking results as $(\#TP)/(\#TP + \#FN)$. Recall indicates the percentage of samples correctly classified as insecure by the tool with respect to all samples containing an information leak. For instance, a recall of 1 indicates that the tool soundly classifies all samples with an information leak as insecure.

The *precision* of a tool is computed from the number of true positives and false positives in the benchmarking results as $\#TP/(\#TP + \#FP)$. Precision indicates the percentage of samples correctly classified as insecure by the tool with respect to all samples classified as insecure by the tool. For example, a precision of 1 indicates that the tool classifies only samples as insecure that contain an information leak, i.e. it never classifies secure samples as insecure. For both recall and precision a higher number indicates better tool performance on the samples used for benchmarking.

4.3 Benchmarking Results

We evaluate each of the four tools on the core samples of IFSpec, as well as on the web vulnerability extension. We present the overall results of benchmarking the four tools in Fig. 5. In this table, the column "#samples" contains the number of samples analyzed and the column "#soap samples" contains the number of samples for which the result of a tool was soundly overapproximated (cf. Section 4.2). Furthermore, the table lists the number of true positives (column "TP") and true negatives (column "TN") as well as the number of false positives (column "FP") and false negatives (column "FP") for each benchmarked tool. The numbers of true positives and false positives are split into the number of samples that are successfully analyzed and the ones soundly overapproximated. The two numbers are separated by a "+". In addition, Fig. 5 shows the recall and precision of the tools on the samples used for benchmarking. For all four tools, we present a detailed overview on the recall and precision for each tag in Fig. 6.

Benchmarking Results for Cassandra. The most noteworthy result of the evaluation is that Cassandra produces no false negatives and thus achieves a recall of 100%. This means that Cassandra reports all leaks in the shared samples of IFSpec. The absence of false negatives is the result of two aspects: (1) The soundness proof of the security type system implemented in Cassandra for almost the full instruction set of Dalvik bytecode. (2) The approach of only adding method signatures that are guaranteed to correctly describe the flow of information caused by library methods.

On the other hand, sound overapproximation of the result of Cassandra takes place in the analysis of 109 samples, a comparatively large number. The sound overapproximation is largely due to missing signatures for library methods, which is the case when a signature has not been manually provided yet or cannot be provided due to the limited expressiveness of the format of method signatures. The missing signatures cause Cassandra to report that it cannot ascertain the security of the sample program. Sound overapproximation causes a relatively high number of false positives, which has an adverse effect on Cassandra's precision.

Further inspecting the results of Cassandra grouped by the tags of samples reveals options for improving its precision. In Fig. 6, it becomes apparent that precision is lower than average for two classes of samples in particular: Those involving branches on secret information (*high-conditional*) and those involving

tool	target language	#samples	#soap samples	TP	TN	FP	FN	recall	precision
Cassandra	DBC	232	109	68+79	15	40+30	0	100%	67.7%
Joana	JBC	232	0	139+0	35	50+0	8	94.6%	73.5%
JoDroid	DBC	232	3	136+2	32	52+1	9	93.9%	72.3%
KeY	JSC	232	208	7+138	12	5+70	0	100%	65.9%

Legend: JSC=Java source code, JBC=Java bytecode, DBC=Dalvik bytecode

Fig. 5. Overview of benchmark results

aliasing (*aliasing*). For the tag *high-conditional*, the relatively low precision can be explained by the fact that the security type system of Cassandra does not allow methods to be invoked in the control dependence regions of high conditionals in order to prevent implicit flows of information via dynamic dispatch. The relatively low precision for the tag *aliasing* can be explained by the fact that the information-flow analysis of Cassandra is not object-sensitive.

Benchmarking Results for Joana. The results of Joana match the ground truths for 174 of the samples in IFSpec. The 50 false positives are mainly caused by the fact that Joana overapproximates actual program behavior. For instance, Joana does not reason about values and does not rule out control flow which is actually impossible due to algebraic invariants. Other sources of imprecision include array handling (Joana does not distinguish between different cells of the same array) and exceptional control flow.

The eight false negatives are due to two reasons. Seven false negatives are caused by the usage of reflection: Joana tries to handle reflective code but leaves it unresolved if it fails in doing so. The resulting PDG is then incomplete.

The second reason is that Joana models static initializers improperly: In one example, the leak is caused by the fact that in Java, class initializers are executed lazily. Joana on the other hand assumes that all class initializers are executed upfront and hence misses the leak because it assumes that the leaking statement is executed at a time when no secret information is available yet.

Benchmarking Results for JoDroid. Surprisingly, the benchmarking results for JoDroid showed differences in 11 samples. These appear to be caused by JoDroid's Dalvik frontend, which not only reads in the bytecode but also performs simple intraprocedural analyses on it.

In three examples Joana could deliver a result while JoDroid crashed. In five examples, Joana did not report a flow and JoDroid did. Possible reasons for this may include differences in the handling of static initializers and the analysis of exceptional control-flow. Three more differences appear to stem from a bug in JoDroid's modelling of multidimensional arrays.

Benchmarking Results for KeY. Even though KeY is not designed for automatic verification of information-flow security, it is able to successfully analyze a small subset of the samples in IFSpec. Since KeY considers a sample secure

tag	# samp.	Cassandra recall	Cassandra precision	Joana recall	Joana precision	JoDroid recall	JoDroid precision	KeY recall	KeY precision
explicit-flows	198	100%	69.2%	94.5%	75.2%	93%	73.9%	100%	66.7%
implicit-flows	34	100%	59.4%	94.7%	64.3%	100%	63.3%	100%	61.3%
simple	63	100%	64.8%	100%	76.1%	100%	76.1%	100%	60.3%
high-conditional	11	100%	44.4%	100%	50.0%	100%	44.4%	100%	44.4%
arrays	32	100%	63.3%	100%	70.4%	89.5%	70.8%	100%	60.0%
class-initializers	10	100%	66.7%	66.7%	80.0%	66.7%	57.1%	100%	60.0%
exceptions	11	100%	63.6%	85.7%	75.0%	100%	77.8%	100%	77.8%
library	94	100%	77.4%	100%	78.3%	100%	78.3%	100%	75.5%
aliasing	12	100%	50.0%	100%	60.0%	100%	54.6%	100%	54.6%
reflection	11	100%	72.7%	12.5%	100%	25.0%	100%	100%	72.7%

Fig. 6. Overview of benchmark results by tag

if and only if a noninterference proof can be derived, KeY has no false negatives and, thus, a recall of 100%. A potential cause for the reported false positives of KeY is the configuration of the applied automatic proof strategy causing it to fail to find a proof. By further tweaking of the relevant parameters and providing stronger auxiliary specifications (e.g. loop invariants) the results of KeY might be improved. In some cases, an interactive proof would be necessary.

As already mentioned in Section 4.1, the treatment of library methods requires sound assumptions about library methods. Since such assumptions are not provided, KeY cannot handle the library calls and, thus, a high number of samples are soundly overapproximated.

4.4 Evaluation Results on the IFSpec Extensions

Aside from evaluating all four information-flow analysis tools on IFSpec's core samples and the web vulnerability extension, we used IFSpec's extensions to further evaluate selected tools. Concretely, we ran JoDroid on the Android vulnerability extension, and KeY on the declassification extension.

Results on the Android Vulnerability Extension. We ran JoDroid on the 119 DroidBench samples that are integrated into IFSpec. JoDroid delivered the correct results on 67 of them (54 true positives, 13 true negatives) and incorrect results on 52 samples (seven false positives, 45 false negatives) – this corresponds to a recall of 54.6% and a precision of 88.5%.

The false negatives shed light on JoDroid's limits: It currently only has rudimentary support for Android features like intents and dynamic broadcast receivers and does not detect entry points corresponding to graphical interfaces. Also, the results clearly show that the stubs we used for JoDroid are insufficient as they do not reflect the dependencies of the actual library methods.

Results on the Declassification Extension. We used KeY to analyze three selected samples from the seven samples in the IFSpec declassification extension: `Declassification5`, `Declassification6`, and `Declassification7`. For this, we

manually translated the RIFL specifications of these samples to JML, as KeY's RIFL parser does not yet support RIFL 1.1. In interactive mode, we were able to prove the security of `Declassification5` and `Declassification6`. We were unable to prove the security of `Declassification7` because it is insecure. The remaining declassification samples were not analyzed because they use floating-point arithmetic, which is not supported by KeY, or because they contain library calls.

5 Related Work

SecuriBench (Micro) and DroidBench. SecuriBench [40] is a benchmark suite for security analyses of web-applications, consisting of nine real-world web applications provided as Java source code that contain security vulnerabilities. Similar to IFSpec and unlike SecuriBench, the samples in SecuriBench Micro [41] explicitly are not real-world applications but small servlets which each focus on particular web vulnerabilities. They are deployable on a Tomcat webserver, which enables penetration testing and the benchmarking of runtime techniques.

The benchmark suite DroidBench [6, 19] focuses on Android and was originally designed to compare FlowDroid [6] with other taint-tracking tools. Hence, its samples contain potential information-flow vulnerabilities.

As described in Section 3.4, both SecuriBench Micro and DroidBench are integrated into IFSpec. Using IFSpec's machine-readable format, the samples from both benchmark suites are made available for IFSpec's benchmark harness and thus accessible as a point of comparison for information-flow analyzers.

SAT, SMT and ATP Benchmark Suites. The SAT and SMT community extensively develops benchmark suites to compare their tools [25]. The comparison of the performance and capabilities of SAT and SMT solvers is performed regularly in annual competitions [7, 15]. The benchmarks used for these competions are categorized into multiple tracks such that solvers that are specialized in a certain type of problem can compete against each other in the corresponding track. This can be compared to the tags used in IFSpec to flag similar samples, which allow tool developers to focus on the kinds of flows and language features supported by their tools when comparing their tools with each other.

The SAT and SMT benchmark samples all come with a fixed formal semantics which simplifies the specification of benchmark problems. Information-flow analysis tools on the other hand often come with distinct security semantics. To accommodate these specific security semantics, the samples of IFSpec are specified using RIFL which provides an informal semantics and a declaration for the ground truth of each sample to which security semantics it is compatible.

In the area of automated theorem proving (ATP), the TPTP (Thousands of Problems for Theorem Provers) benchmark suite [42] is widely accepted for testing and evaluating ATP systems. One contribution of TPTP is a standardized input and output format for ATP systems that enables sharing test problems between researchers and ATP systems. This format is a key factor in TPTP's success [43]. Our use of RIFL also aims at standardizing input and output format, albeit in the area of information-flow security.

Java Performance Benchmark Suites. Several benchmark suites exist for Java (e.g., [11, 13, 39]), mostly focusing on JVM runtime performance and memory consumption. They differ mostly in their selection of samples. Like IFSpec, they all contain a benchmark harness for running the individual samples and reporting performance data. The DaCapo benchmark suite [11] consists of multiple real-world applications, while the JavaGrande benchmark suite [13] focuses on computationally intensive and multi-threaded applications [39].

6 Conclusion

With IFSpec, we provide a benchmark suite for information-flow analysis tools targeting Java source, Java bytecode, or Dalvik bytecode. The coverage of these three language layers of the Java and Android platforms enabled us to evaluate and compare Cassandra, Joana, JoDroid, and KeY on a uniform set of samples, despite the differences between the respective target languages.

We provide all samples of IFSpec in a machine-readable format. In this format, RIFL [8] is employed for the specification of information-flow policies in a uniform fashion. The only prerequisite for automatically benchmarking tools with IFSpec, both static and dynamic ones, is the existence of a RIFL frontend. Naturally, developing such a frontend is easier for tools that clearly separate the target program from the policy specification. This is why we refrained from extending our comparison to analysis tools that closely couple programs and policies, like, e.g., Jif [34]. Nonetheless, support for such tools is possible, and frontends for them might be added in the future. Because RIFL is based on the well established XML standard, a multitude of third party parsers are available and can be used when implementing a RIFL frontend. This allows tool developers to focus on the transformation of a policy specification from RIFL to the specification language or annotation approach supported by their tool.

For the future, we encourage researchers in the community to use IFSpec to evaluate further information-flow analysis tools and to extend IFSpec. Such extensions could include the addition of samples to IFSpec, the creation of further extensions with language-specific examples, or the classification of IFSpec's samples for additional, formally defined information-flow properties. Albeit the current scope of IFSpec is not on testing scalability of analysis tools, we see the addition of larger sample programs as one promising direction.

Acknowledgments. We thank the anonymous reviewers for their helpful comments and the participants of the RS^3 Staff Meeting 2016 for contributing to the samples of IFSpec. This work was supported by the DFG under the projects DeduSec (BE 2334/6-3), IFC4MC (Sn 11/12-3), and RSCP (MA 3326/4-3) in the priority program "Reliably Secure Software Systems" (RS^3, SPP 1496).

References

1. HPE Security Fortify Static Code Analyzer (SCA). [Online] Available: `https://saas.hpe.com/en-us/software/sca`. accessed 8.8.18.

2. IBM Security AppScan. [Online] Available: `https://www.ibm.com/developerworks/downloads/r/appscan/index.html`. accessed 8.8.18.
3. SDK Platform Release Notes. [Online] Available: `https://developer.android.com/studio/releases/platforms.html`. accessed 8.8.18.
4. W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. H. V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY Platform for Verification and Analysis of Java Programs. In *VSTTE '14*, pages 1–17.
5. The Activity Lifecycle of Android. [Online] Available: `https://developer.android.com/guide/components/activities/activity-lifecycle.html`. accessed 8.8.18.
6. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI '14*, pages 259–269.
7. T. Balyo, M. J. Heule, and M. Järvisalo. SAT Competition 2016: Recent Developments. In *AAAI '17*, pages 5061–5063.
8. T. Bauereiß, S. Greiner, M. Herda, M. Kirsten, X. Li, H. Mantel, M. Mohr, M. Perner, D. Schneider, and M. Tasch. RIFL 1.1: A Common Specification Language for Information-Flow Requirements. Technical Report TUD-CS-2017-0225, TU Darmstadt, 2017.
9. B. Beckert, D. Bruns, V. Klebanov, C. Scheben, P. H. Schmitt, and M. Ulbrich. Information Flow in Object-Oriented Software. In *LOPSTR '13*, pages 19–37.
10. S. Bischof, J. Breitner, J. Graf, M. Hecker, M. Mohr, and G. Snelting. Low-deterministic security for low-deterministic programs. *Journal of Computer Security*, 26:335–336, 2018.
11. S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinly, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. . Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06*, pages 169–190.
12. J. Breitner, J. Graf, M. Hecker, M. Mohr, and G. Snelting. On Improvements Of Low-Deterministic Security. In *POST '16*, pages 68–88.
13. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. In *JAVA '99*, pages 81–88.
14. E. S. Cohen. Information Transmission in Sequential Programs. *Foundations of Secure Computation*, pages 297–335, 1978.
15. D. R. Cok, D. Déharbe, and T. Weber. The 2014 SMT competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:207–242, 2016.
16. S. P. E. Corporation. Spec CPU Benchmarks. [Online] Available: `https://www.spec.org/benchmarks.html#cpu`. accessed April 8.8.18.
17. D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.
18. R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving Multilevel Security of a System Design. In *SOSP '77*, pages 57–65.
19. C. Fritz, S. Arzt, and S. Rasthofer. DroidBench 2.0. [Online] Available: `https://github.com/secure-software-engineering/DroidBench`. accessed 8.8.18.
20. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *S&P '82*, pages 11–20.
21. J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *ATPS '13*, pages 123–138.

22. C. Hammer and G. Snelting. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *International Journal of Information Security*, 8(6):399–422, Dec. 2009.
23. Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *ISCAS '08*, pages 1192–1195.
24. J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, Jul 2000.
25. H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. *Sat2000: Highlights of Satisfiability Research in the Year 2000*, pages 283–292, 2000.
26. K. Ku, T. E. Hart, M. Chechik, and D. Lie. A Buffer Overflow Benchmark for Software Model Checkers. In *ASE '07*, pages 389–392.
27. S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a Certifying App Store for Android. In *SPSM '14*, pages 93–104.
28. A. Lux and H. Mantel. Declassification with Explicit Reference Points. In *ESORICS '09*, pages 69–85.
29. A. Lux and H. Mantel. Who Can Declassify? In *FAST '09*, pages 35–49.
30. H. Mantel. Information Flow and Noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 605–607. 2011.
31. J. K. Millen. Information Flow Analysis of Formal Specifications. In *S&P'81*, pages 3–8.
32. M. Mohr, J. Graf, and M. Hecker. JoDroid: Adding Android Support to a Static Information Flow Control Tool. In *SE '15*, pages 140–145.
33. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification. In *CSFW '04*, pages 172–186.
34. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. [Online] Available: `http://www.cs.cornell.edu/jif`. accessed 8.8.18.
35. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN '07*, pages 263–267.
36. J. M. Rushby. Design and Verification of Secure Systems. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 12–21, 1981.
37. A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In *ISSS '03*, pages 174–191.
38. S. E. Sim, S. Easterbrook, and R. C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *ICSE '03*, pages 74–83.
39. L. A. Smith, J. M. Bull, and J. Obdrizalek. A Parallel Java Grande Benchmark Suite. In *SC '01*, pages 8–8.
40. Stanford SecuriBench. [Online] Available: `http://suif.stanford.edu/~livshits/work/securibench/intro.html`. accessed 8.8.18.
41. SecuriBench Micro. [Online] Available: `https://github.com/too4words/securibench-micro`. accessed 8.8.18.
42. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, 43(4):337–361, 2009.
43. G. Sutcliffe, S. Schulz, K. Claessen, and A. V. Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In *IJCAR '06*, pages 67–81.
44. D. Wasserrab and D. Lohner. Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing. In *VERIFY '10*.
45. M. Zanioli, P. Ferrara, and A. Cortesi. SAILS: Static Analysis of Information Leakage with Sample. In *SAC '12*, pages 1308–1313.