A Comparative Analysis Between Information Flow Control Tools for Java-written systems

Gabrielle Amorim
Universidade Federal do Agreste de Pernambuco
Garanhuns, Brazil
gabrielleporto19@gmail.com

Rodrigo Andrade
Universidade Federal do Agreste de Pernambuco
Garanhuns, Brazil
rodrigo.andrade@ufape.edu.br

Felipe Ebert
Eindhoven University of Technology (TU/e)
Eindhoven, The Netherlands
f.ebert@tue.nl

ABSTRACT

Information Flow Control (IFC) tools are a common way to analyze source code with the goal to find confidentiality or integrity violations for sensitive information. Therefore, to correctly protect such information (e.g., passwords), it is important to choose the most suitable tool for each target software system. In this context, we evaluate precision, recall, and accuracy for three open-source IFC tools for Java-written systems. We also check whether these tools are useful to protect sensitive information of real systems. First, we execute these tools against test cases of the SecuriBench Micro benchmark built for this purpose. Then, we run three selected IFC tools (JOANA, PIDGIN, and Flowdroid) to assess whether they are able to detect violations for rules we define considering each real system. Our results show that JOANA and PIDGIN overcome FlowDroid regarding precision, recall, and accuracy. Furthermore, the execution of JOANA and PIDGIN allow us to find eight confidentiality and integrity violations for the target systems. We registered these violations as issues on those projects. Our results also demonstrate that JOANA is faster than PIDGIN. At last, we provide some discussion for developers on which IFC tool fits better when dealing with sensitive information in software systems.

KEYWORDS

Information-Flow Control, Sensitive information, Confidentiality, Integrity, Precision, Recall, Accuracy

ACM Reference Format:
Gabrielle Amorim, Rodrigo Andrade, and Felipe Ebert. 2021. A Comparative Analysis Between Information Flow Control Tools for Java-written systems. In *15th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '21), September 27-October 1, 2021, Joinville, Brazil*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3483899.3483901

SBCARS '21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-8419-3/21/09…$15.00
https://doi.org/10.1145/3483899.3483901

1 INTRODUCTION

Many systems handle sensitive information such as user passwords or location. Therefore, developers should care about protecting this information confidentiality and integrity. There are a number of different mechanisms to try to achieve this goal such as static analysis [12, 13, 23, 30–32, 37] or manual code review [20]. Albeit necessary in a number of cases, manual code review is time-consuming and expensive [28]. On the other hand, Information Flow Control (IFC) analysis tools mitigate these drawbacks [14, 22, 26].

IFC tools track how information propagates through the program during execution [24]. They are concerned with two properties: information confidentiality and information integrity [24]. Integrity guarantees that unauthorized parties like methods do not influence sensitive information [23]. Confidentiality guarantees that sensitive information does not leak to unauthorized parties [22].

In this context, it is important to assess state of the art IFC tools regarding their recall, precision, and accuracy. The results of such assessment could bring knowledge that would help to choose the most suitable (and available) tool for a given purpose. So, in this work we perform an experiment to compare three IFC tools: JOANA [22], FlowDroid [14], and PIDGIN [26]. To obtain the recall, precision, and accuracy, we run these tools against a well-known benchmark, the SecuriBench Micro [27], which provides many test cases that are useful to exercise IFC tools.

Our results show that JOANA and PIDGIN have similar results concerning recall, precision, and accuracy for 137 existing confidentiality or integrity violations provided by SecuriBench Micro. JOANA presents the highest recall and it is capable to detect the highest number of existing issues whereas PIDGIN presents higher precision and accuracy. On the other hand, FlowDroid presents the lowest issues detected as well as the lowest recall.

Since JOANA and PIDGIN present higher precision and recall, we use these tools to evaluate whether they can protect sensitive information for real systems. Therefore, we select three open-source projects written in Java that handle sensitive information. They are called Blojsom [5], Lutece [29], and ScribeJava [35]. We define five constraints for each one, which contains confidentiality and integrity rules (e.g., user password cannot be written to log files). After running the experiment, we conclude that JOANA and PIDGIN are able to detect confidentiality and integrity violations for these real systems.

Additionally, we also measure the execution time that JOANA and PIDGIN takes to perform the analysis for each defined constraint. We conclude that, despite of JOANA presenting lower precision and accuracy, it is approximately 150 times faster than PIDGIN when running their IFC analysis for the selected systems.

In summary, this work shares the following contributions:

- An assessment regarding precision, recall, and accuracy for three state of the art IFC tools;
- An additional evaluation to bring evidence that two of these IFC tools can indeed protect sensitive information for real systems;
- A performance study about two IFC tools.

The remainder of the paper is structured as follows. Section 2 shows the main concepts that are necessary for a better understanding of this work. Section 3 discusses our research questions, target systems, constraints, and experiment. In Section 4, we explain our assessment regarding IFC tools precision, recall, and accuracy as well as their results considering constraints and real systems. Section 5 discusses the threats to validity of our assessment whereas Section 6 explains related work. At last, Section 7 presents our concluding remarks.

2 BACKGROUND

In this section, we present the main concepts necessary for a better understanding of this work. In Section 2.1, we explain two key concepts we use in this research: *Confidentiality* and *Integrity*. Section 2.2 discusses Information Flow Control analysis.

2.1 Confidentiality and Integrity

Confidentiality demands that sensitive information does not leak to potentially dangerous methods [23]. This way, only authorized methods should access such information. For example, in Listing 1, the sensitive information is the user password whereas the leaking method is `log()`. In this context, we would like to avoid printing user password throughout log files.

Listing 1: Confidentiality violation

```
1 void authenticate(User u) {
2  String password = u.getPassword();
3  ...
4  LOGGER.log("User " + login + " with pwd "
5   + password);
6  ...
7 }
```

In this work, we approach confidentiality by means of information flow. For the code snippet above, we can detect sensitive information violation by identifying that there is an information flow from `password` to the `log()` method.

Integrity demands that unauthorized methods do not influence sensitive information [23]. In other words, integrity assures information is not changed. For instance, Listing 2 illustrates a violation of sensitive information integrity.

The malicious code in line 3 changes the user password so that an attacker can easily access the account of any user. In this example, the unauthorized method is the call to `readInput()` in line 3 whereas the sensitive computation is the `saveUser()` call in line 5.

Listing 2: Integrity violation

```
1 void updateUser(User u) {
2 ...
3  u.setPwd(readInput());  // malicious code
4  String password = u.getPwd();
5  c.saveUser(u.getLogin(), encrypt(password));
6 }
```

At last, confidentiality and integrity are dual to each other [4]. Thus, to guarantee both, we need to check for information flows from one source code location to another and vice-versa.

2.2 Information Flow Control analysis

Information Flow Control (IFC) is concerned with the flow of information inside a system. It tracks how information propagates through the program during execution. Thus, one important IFC goal is to try to assure that the program handles the information properly [24]. According to Hammer et al., IFC analysis has two main tasks [23]:

- Guarantee *confidentiality* of information;
- Guarantee *integrity* of information.

In general, IFC analysis determines whether there is a potential information flow from one point to another in program execution [23]. Therefore, it could be useful to detect whether there is a possible flow from a statement that holds a sensitive information to an unauthorized method. In other words, we can use IFC analysis to enforce sensitive information confidentiality. For example, in Listing 1, we can automatically detect that there is an information flow from `password` to `log()`.

Additionally, we can use IFC analysis to detect whether there is a possible flow from an unauthorized method to a sensitive information, which regards to integrity. For instance, in Listing 2, we can automatically detect that there is a flow from `setPwd` to the user `password`.

In this context, there are a number of IFC tools encompassing different approaches to achieve those two main tasks [12, 14, 18, 23, 26, 30, 33, 34, 37]. These tools vary regarding their underlying mechanisms to deliver Information Flow Control. For example, JOANA [21] builds its IFC analysis using a graph structure named System Dependence Graph (SDG) [25]. Statements are nodes whereas edges represent flows of data or control from one node to another. JOANA allows us to manually set which source code statement represents sensitive information and which statement represents unauthorized methods (i.e., it could be either a method call or its instructions). Despite of also using SDGs, PIDGIN [26] provides a different way for us to specify sensitive information. As explained in Section 3.2, we need to write policies in its own policy language. In opposition to JOANA and PIDGIN, Flowdroid [14] automatically determines sensitive information for Android applications (e.g., Mobile IMEI).

In this work, we focus on these three tools: JOANA, Flowdroid, and PIDGIN. We introduce them in Section 3.2.

3 RESEARCH METHOD

In this section, we explain our research questions and metrics (Section 3.1), the selected IFC tools (Section 3.2), the target systems (Section 3.3), our experiment (Section 3.5), and execution environment (Section 3.6).

3.1 Goal-Question-Metric

To better drive our work, we adopt the Goal-Question-Metric (GQM) approach [17]. Our goal is to compare three state-of-the-art Information Flow Control tools regarding precision, recall, and accuracy. We also have the goal to check how the best two tools perform to detect sensitive information flow to unauthorized methods in real systems. Table 1 summarizes our GQM approach.

Table 1: Our GQM approach

Goal	
Purpose *Issue* *Object* *Viewpoint*	Evaluate IFC tools regarding precision, recall, and accuracy for benchmark and real projects from an IFC tool user viewpoint
Questions and Metrics	

RQ1- Which selected IFC tool performs better regarding precision, recall, and accuracy?	`-Precision (PRE)`
	`-Recall (REC)`
	`-Accuracy (ACC)`

RQ2- The selected IFC tools can detect flows from sensitive information to unauthorized methods and vice-versa for real systems?	`-Number of violations found (NVF)`
RQ2.1- Which selected IFC tool is faster to find information flows?	`-Execution time (ET)`

As showed in Table 1, we define two major research questions and a minor one. **RQ1** is important to bring evidence of which IFC tool performs better by means of precision, recall, and accuracy. This answer could guide other developers and researchers to choose the most suitable tool for their purpose. This way, we run the selected tools against a well-known benchmark commonly used for this goal. To answer **RQ1**, we use three metrics: precision, recall, and accuracy.

`Precision` can be defined as the number of true-positives divided by the sum of true-positives and false-positives. For example, if the IFC tool flawlessly identifies two unauthorized information flows as well as one nonexistent flow, the precision would be 66%.

`Recall` can be defined as the number of true-positives divided by the sum of true-positives and false-negatives. For example, if the tool correctly detects all the unauthorized information flows for a given test case, the recall is 100%.

`Accuracy` is defined as the sum of true-positives and true-negatives divided by the sum of true-positives, false-positives, false-negatives, and true-negatives. For instance, in case we correctly detect six unauthorized information flows plus two nonexistent flows, the accuracy would be 75%.

Moreover, our goal to define **RQ2** is to bring evidence that the selected IFC tools can, indeed, find sensitive information flowing to unauthorized methods considering real systems, which encompasses confidentiality. In addition, it is our goal to bring evidence that these tools can also detect flows on the opposite direction, that is, from unauthorized methods to sensitive information, which encompasses integrity. In this context, we define scenarios for these systems (e.g., log method cannot access password information). In case the tool detects a flow for such scenarios, we count it as a violation. Therefore, the `Number of violations found` metric is increased by one.

At last, we define the minor research question **RQ2.1**. The answer to this question also helps to determine which IFC tool is more suitable for a given context. For example, one tool could be very precise, however its execution time turns it unfeasible for certain kind of systems [23]. Thus, we use the `execution time` metric.

3.2 Selected IFC tools

IFC tools automatically analyze source code in order to find potential information flows. In general, we can specify *Sources* and *Sinks*, which allow developers to determine program parts that hold sensitive information and source code location where it cannot flow (e.g., unauthorized methods) [3]. To perform our comparative study, we select three open-source IFC tools: JOANA [21], FlowDroid [14], and PIDGIN [26].

JOANA is a framework for performing IFC analysis on Java programs. It supports static analysis of Java systems in order to find integrity and confidentiality violations [22]. JOANA provides an annotation mechanism so that developers can manually define *Sources* and *Sinks* (e.g., sensitive information and unauthorized methods). To execute its IFC analysis, we must provide a program entry-point, such as a main method. It also provides a user-friendly interface in which we could manually select *Sources*, *Sinks*, and entry-point.

Our second selected IFC tool is named FlowDroid. It computes data flows in Android apps and Java programs [14]. For Android apps, FlowDroid has a predefined list of *Sources* and *Sinks* that represent sensitive information and leaking points (e.g., unauthorized methods). On the other hand, for Java programs, we can customize the list of *Sources* and *Sinks* as well as the entry-point.

Our third selected tool is called PIDGIN. It is an IFC analysis tool that allows us detecting whether an information flow exists. Differently from JOANA, it provides a policy language for specification of *Sources* and *Sinks*. In this context, we define queries in this policy language and execute PIDGIN. Therefore, this tool determines whether there is a flow between the specified *Sources* and *Sinks* (e.g., user location field and send email method).

Finally, it is not our goal to investigate the differences between the tools' implementation in this work. However, we briefly explain it when needed for better understanding.

3.3 Target systems

To answer **RQ1**, we select the SecuriBench Micro [27]. It consists of 122 test cases written in Java, which are grouped into 12 distinct groups: Aliasing, Arrays, Basic, Collections, Data Structures, Factories, Inter, Pred, Reflection, Sanitizers, Session, and Strong Update. Other authors have also used it [12–15, 26] to test their approaches. This way, we use the SecuriBench Micro to identify the cases that IFC tools find and miss information flows, also following the metrics used

by literature [14, 15, 26], we consider precision, recall, and accuracy to measure the performance of the tools.

Additionally, to answer **RQ2** and **RQ2.1**, by using convenience sampling, we selected three open-source Java-written projects: Blojsom [5], Lutece [29], and ScribeJava [35]. Our decision was based on the fact that those projects are mature enough to contain data points of interest.

Blojsom is a blog application that uses the Simple Logging Facade for Java (SLF4J) [19] external library, which is useful to log operations throughout the source code. These logging operations are known to be a potentially dangerous point where unauthorized methods have access to sensitive information [30] as specified in the Common Weakness Enumeration (CWE) [8] and in The Open Web Application Security Project (OWASP) [20]. Thus, often, we should not log sensitive information [3]. Indeed, if we ignore this problem, we might compromise such information confidentiality and integrity. Furthermore, the Blojsom project has 4,419 commits and nearly 7000 lines of Java code. It was developed for ten years, until 2013. However, its source code is still available for download.

Our second selected system is named Lutece. It is an open-source web portal that allows users to create websites or intranets. It also handles sensitive information such as user passwords. Thereby, it is interesting to check whether there are flows from this information to unauthorized methods. Lutece has 3,718 commits and 16 open pull requests. This project is still active.

Our last selected system is called ScribeJava. It is a simple authentication (OAuth) library for Java programs. It is designed to provide authentication functionalities which means ScribeJava handles sensitive information such as user API key and secret. It has 1.145 commits and 13 open pull requests. Like Lutece, this project is still active and is supported by several developers, who submit their code contributions to its repository on GitHub.

3.4 Constraints

To answer **RQ2** and **RQ2.1**, we define rules regarding sensitive information and unauthorized methods. Also, we consider that a violation occurs when these rules are not obeyed. For each target system, we specify five constraints that contain rules as follows.

Blojsom

- **C1** The method responsible for authorizing users to execute certain operations on the system must not have writing access on the user's password. However, it can execute reading operations;
- **C2** The method responsible for checking the users' permissions must not execute writing operations related to user password. However, it can execute reading operations;
- **C3** The method responsible for sending e-mails to users must not execute writing operation with the sender's e-mail and the recipient's e-mail(s);
- **C4** The method responsible for getting detailed users' information must not execute writing operations related to the user's password. However, it can execute reading operations;
- **C5** Methods responsible for accessing log files must not execute writing and reading operations utilizing user passwords.

Lutece

- **C6** The method responsible for getting users' passwords must not execute writing operations on passwords. However, it can execute reading operations;
- **C7** The method responsible for validating users' passwords must not execute writing operations on passwords. However, it can execute reading operations;
- **C8** The method responsible for user password decryption must not execute writing operations on passwords. However, it can execute reading operations;
- **C9** Methods that execute queries in the database using the password must not execute operations writing on passwords. However, it can execute reading operations;
- **C10** Methods that perform update operations in the database using the password must not execute writing operations on passwords. However, it can execute reading operations.

ScribeJava

- **C11** The method responsible for getting signature request must not execute writing operations on secret tokens. However, it can execute reading operations;
- **C12** The method responsible for adding the API signature to the request must not execute writing operations on secret tokens. However, it can execute reading operations;
- **C13** The method responsible for writing information into the log files must not execute reading and writing operations on secret tokens;
- **C14** The method responsible for adding the parameters into requests must not execute writing operations on secret tokens. However, it can execute reading operation;
- **C15** The method responsible for token encryption must not execute writing operations on secret tokens. However, it can execute reading operation.

Since our goal is to assess existing IFC tools, we do not consider existing policy languages [3, 6, 7, 30–32, 37] to write our constraints. This would demand changing these tools' implementation [3], which could introduce bias to our experiment. Next, we explain our strategy to run our experiment.

3.5 Experiment

In this work, we divide our experiment execution into nine steps. We explain each one below.

(1) **Searching for IFC tools.** This step regards the process to find Information Flow Control analysis tools. We limit our search by following these criteria: (i) tools that can analyze systems written in Java, (ii) open-source tools, (iii) feasibility to build the tool's source code, and (iv) tools that do not demand system source code changes. We identified eight available tools. Five of them were

rejected in this process. First, ANDROMEDA [13] and TAJ [12] do not meet the (ii) criteria. Second, Jif [32], Jeeves [37], and Joe-E [30] do not meet the (iv) criteria. Finally, we selected three IFC tools: JOANA, FlowDroid, and PIDGIN;

(2) **Searching for benchmark projects.** The goal of this step is to find a benchmark project to test the selected IFC tools. Thus, it should provide test cases that are useful to measure precision, recall, and accuracy. During this process, we notice that a number of related work [12–15, 26] use the SecuriBench Micro [27] for this purpose. Thereby, we also select this benchmark project to execute our analysis. It implements, on purpose, a number of confidentiality and integrity violations that we could use to check whether IFC tools are able to detect;

(3) **Running IFC tools against the benchmark project.** For each test case provided by SecuriBench Micro, we manually assign *Sources* and *Sinks*. To check confidentiality, the former represents sensitive information whereas the latter represents unauthorized methods. On the other hand, to check integrity, we assign them in the opposite way since they are dual [4]. Then, we run our three selected IFC tools for each test case. Most test cases in SecuriBench Micro have sensitive information integrity and confidentiality violations;

(4) **Analyzing first results.** In this step, we manually count the number of true-positives and negatives as well as false-positives and negatives for each tool and test case. Then, we use these data to measure precision, recall, and accuracy (Section 3.1). At last, we manually analyze these measures to determine which IFC tool performs best;

(5) **Selecting real systems.** To better evaluate the IFC tools, we select real systems that handle sensitive information and present potentially unauthorized method. Basically, these systems must be written in Java and open-source. We manually review their source code to check whether they meet our needs. Therefore, we select three projects: Blojsom [5], Lutece [29], and ScribeJava [35];

(6) **Defining constraints.** We manually analyze each selected systems to understand their sensitive information and potentially dangerous methods. Thereby, we define five constraints for each selected system. Each constraint contains sensitive data confidentiality and/or integrity rules. For instance, we might define that a particular field that contains sensitive information cannot flow to a method that exposes it publicly;

(7) **Adapting constraints.** For JOANA, we use its graphical interface to assign which statement is *Source* or *Sink* for each constraint. For example, we color code statements as green to represent *Sources* whereas red ones represent *Sinks*. On the other hand, we write PIDGIN policy for each of our constraints;

(8) **Running IFC tools against real systems.** We execute the IFC tools for the target systems and the specified constraints;

(9) **Analyzing final results.** In this step, we collect the number of confidentiality and integrity violations found by the IFC tools. Thus, we compare them by means of performance and violations found for each defined constraint. Finally, we manually reviewed the systems' source-code to confirm the information flows detected by the tools.

3.6 Execution environment

We perform our study in a laptop equipped with an Intel Core i7 8th generation processor. It has 16GB of RAM memory and a 128GB Solid State Drive. The operation system we use is Linux Ubuntu version 20.4. Moreover, we adopt Java 7 to run our analysis. Lastly, we consider JOANA released on February of 2020, FlowDroid 2.5.1, and PIDGIN released on November of 2015.

4 RESULTS AND DISCUSSION

In this section, we explain the results we obtain after running our experiment. First, we answer **RQ1** in Section 4.1. Then, we discuss our answers to **RQ2** and **RQ2.1** in Section 4.2.

4.1 IFC tools precision, recall, and accuracy

Table 2 illustrates the results we obtain when running JOANA, FlowDroid, and PIDGIN against the SecuriBench Micro. The **Test Case Group** column shows different cases for testing IFC tools. Each test case group has a number of information flows to be detected. For example, concerning *Reflection*, JOANA correctly identifies three flows out of four. Therefore, it misses one flow, which introduces a false-negative. Consequently, JOANA has a 75% recall and accuracy for this test case group.

Furthermore, JOANA is able to detect the existing information flow present in 10 groups, which means 100% recall for them. On the other hand, it misses two flows for *Reflection* and *Inter*. This drawback happens due to the resulting System Dependence Graph [25] not including the nodes and edges necessary to reach the information flow. The precision and accuracy vary from group to group. For instance, JOANA presents 50% precision and accuracy for the *Factories* group because it also detects false-positives. Nonetheless, JOANA allows additional optimizations that improve analysis precision, which leads to a reduction of false-positives [22] with the cost of performance loss. Since Flowdroid and PIDGIN do not allow these optimizations, we do not use them in JOANA. Otherwise, we could present biased results.

FlowDroid was able to achieve 100% recall for only two test case groups: *Arrays* and *Collections*. It happens due to the high number of false-negatives. In particular, FlowDroid was not able to detect any flow for *Session* and *Strong Update* because the test cases these groups present are related to a dependency that FlowDroid's main library does not support. Unfortunately, FlowDroid does not support predicates, sanitizers, and reflection [14]. Thus, we could not evaluate it for these three groups.

Our results differ from Artz et al. [14] mainly for *Basic*, *Factories*, and *Session* test case groups. This happens due to two reasons. First, this benchmark has been updated with new

Table 2: IFC tools results for SecuriBench Micro

	JOANA				FlowDroid				PIDGIN			
Test Case Group	**Detected**	**REC**	**PRE**	**ACC**	**Detected**	**REC**	**PRE**	**ACC**	**Detected**	**REC**	**PRE**	**ACC**
Aliasing	12/12	100%	85%	87%	10/12	83%	100%	85%	12/12	100%	100%	100%
Arrays	9/9	100%	60%	62%	9/9	100%	64%	62%	9/9	100%	64%	62%
Basic	61/61	100%	91%	91%	30/61	51%	96%	53%	61/61	100%	100%	100%
Collections	14/14	100%	66%	68%	14/14	100%	93%	94%	14/14	100%	73%	79%
Data Structures	5/5	100%	62%	62%	4/5	80%	80%	75%	5/5	100%	100%	100%
Factories	3/3	100%	50%	50%	1/3	33%	100%	66%	3/3	100%	100%	100%
Inter	15/16	93%	57%	60%	15/16	93%	100%	96%	16/16	100%	100%	100%
Predicates	5/5	100%	62%	66%	-	-	-	-	5/5	100%	71%	77%
Reflection	3/4	75%	100%	75%	-	-	-	-	1/4	25%	100%	25%
Sanitizers	4/4	100%	44%	44%	-	-	-	-	3/4	75%	100%	88%
Session	3/3	100%	75%	75%	0/3	0%	0%	25%	3/3	100%	75%	75%
Strong Update	1/1	100%	20%	20%	0/1	0%	0%	20%	1/1	100%	33%	60%
Total	**135/137**	**97%**	**64%**	**63%**	**83/124**	**60%**	**70%**	**64%**	**133/137**	**91%**	**84%**	**80%**

and different test cases since these researchers evaluated it. Second, we contacted FlowDroid's developers to discuss our results and they reported to us that FlowDroid's main library has been updated since the last analysis. So, oddly the new one does not have the dependencies required by SecuriBench Micro test cases, which leads to Flowdroid missing these test case groups.

Like JOANA, PIDGIN is able to detect the information flow present in 10 out of 12 groups. Therefore, this tool does not achieve 100% recall only for *Reflection* and *Sanitizers* due to four false-negatives. This happens due to four undetected existing information flows. In addition to JOANA, PIDGIN demands us to write the constraints in its policy language. Therefore, it builds the corresponding SDG [25] accordingly. To prevent that we introduce policy specification errors, all the authors of this work review each one. Therefore, we believe PIDGIN misses these flows due to an issue in its mechanism to build an SDG from the defined policy. For *Reflection*, PIDGIN misses three existing flows whereas for *Sanitizers* it misses one flow. However, the low number of false-positives results in a high precision and accuracy. This way, five test case groups obtain 100% value for precision, recall, and accuracy.

Equally to FlowDroid, other researchers have also performed a similar evaluation considering PIDGIN [26]. Nonetheless, in this case our results are equal to previous work [26].

By analyzing the last row of Table 2, JOANA has the highest rate of detected flows, which is 135 out of 137. On the other hand, PIDGIN misses two more flows. Therefore, these tools present high rates of recall. Our results also show that FlowDroid only detects 83 flows out of 124, which is the worst result among these three IFC tools. Additionally, JOANA has the highest rate of recall. It obtains 97% for this metric whereas PIDGIN has 91%. Both tools are better than FlowDroid which reached only 60% of recall. For the precision metric, PIDGIN has 84% followed by FlowDroid with 70%, and JOANA with 64%. Considering accuracy, PIDGIN also achieves the best result with a rate of 80%.

Last but not least, we answer **RQ1** *stating that JOANA is the best choice if the goal is high recall and PIDGIN is the best choice if the goal is high precision or accuracy.*

4.2 Analyzing real systems

In this section, we aim at answering **RQ2** and **RQ2.1**. As mentioned in Section 3.5, we run JOANA and PIDGIN against real systems. Next, we discuss our results for each system.

4.2.1 Blojsom results. Table 3 illustrates the results we obtain for the Blojsom system regarding the Number of Violation Found (**NVF**) and Execution Time (**ET**) metrics. We observe that both JOANA and PIDGIN are able to detect the same existing confidentiality violation to **C3**, which we define in Section 3.4. As Blojsom is a real system instead of a benchmark with previously known violations, we do not know whether these tools find all existing violations regarding **C1**, **C2**, **C3**, **C4**, and **C5**. To mitigate this issue, we also manually analyze Blojsom source code to confirm or discard violation warnings provided by JOANA and PIDGIN.

Table 3: Blojsom results

	JOANA		PIDGIN	
	NVF	**ET**	**NVF**	**ET**
C1	0	6120	0	390600
C2	0	1730	0	439200
C3	1	1604	1	457200
C4	0	1737	0	394000
C5	0	2385	0	393600
Total	1	13576	1	2074600

Indeed, there is a violation to **C3**. Listing 3 shows where it occurs in the source code. The `send()` method writes recipients information in log files through a call to the `info()` method in lines 4 and 5. Therefore, `info()` incorrectly access user email information, which is sensitive in this context.

Listing 3: Send e-mail method

```
1 void send() {
2  HtmlEmail mail = new HtmlEmail();
3  ...
4  _logger.info((new StringBuffer()).append
5   ("Email sent to ").append(getRecipients());
6 }
```

Regarding **ET**, Table 3 shows that JOANA performs the same task faster than PIDGIN. To analyze the source code for the five constraints, JOANA takes 13576 milliseconds whereas PIDGIN takes 2074600. Thus, the former tool is roughly 150 times faster than the latter for this example.

4.2.2 Lutece results. Table 4 shows the results regarding Lutece. It is interesting to notice that JOANA presents a total of four violations found whereas PIDGIN achieves only three. Furthermore, we can notice that JOANA is faster than PIDGIN once again. We did not expect this outcome because both tools use similar mechanisms to implement their analysis, which is based in System Dependence Graphs [25].

Table 4: Lutece results

	JOANA		PIDGIN	
	NVF	**ET**	**NVF**	**ET**
C6	1	6447	1	393000
C7	1	1701	0	408000
C8	2	1615	2	399000
C9	0	1838	0	435000
C10	0	1931	0	433200
Total	4	13523	3	2068200

As showed in Table 4, JOANA and PIDGIN identify a confidentiality violation for **C6**. The snippet of code in Listing 4 shows `strStoredPwd` being passed as argument when an exception is raised. Therefore, if the system throws this exception, it could print the sensitive information in the stack trace. Thus, a user who has access to the stack trace is able to see passwords of different users, characterizing a confidentiality violation.

Listing 4: getPassword method

```
1 IPassword getPassword(String strStoredPwd) {
2  int strTypeSepIndex = strStoredPwd.indexOf(':');
3  if (strTypeSepIndex == -1) {
4   throw new IllegalArgException(strStoredPwd);
5  }
6   ...
7 }
```

For **C7**, JOANA identifies a violation. The snippet of code where it happens is shown in Listing 5. In line 2, the `DigestPassword` class constructor assigns the `strStoredPwd` value to the class field `_strPassword`, which increases its scope. This might represent a confidentiality violation. PIDGIN is not able to identify this violation because it classifies this case as a reading flow, which would not be a violation to **C7**.

Listing 5: DigestString

```
1 DigestPassword(..., String strStoredPwd) {
2  _strPassword = strStoredPwd;
3  ...
4 }
```

Moreover, for **C8**, JOANA and PIDGIN detects two violations. In lines 5 and 11 of Listing 6, the `PBKDF2Password` class constructor can throw exceptions passing `strPassword` as argument. However, this field might contain sensitive information. Therefore, if the system throws these exceptions, it can print a sensitive information in the stack trace.

On the other hand, JOANA and PIDGIN correctly find no violation for **C9** and **C10**. Last but not least, as shown in Table 4, JOANA is faster than PIDGIN to analyze these five constraints. Indeed, the former is also approximately 150 times faster (i.e, same result for Blojsom).

Listing 6: PBKDF2Password

```
1 PBKDF2Password(String strPassword, ...) {
2  ...
3  if(...) {
4   throw new IllegalArgumentException(
5        ERROR_PASSWORD_STORAGE + strPassword);
6  }
7  try {
8   ...
9  } catch(DecoderException e) {
10       throw new IllegalArgumentException
11        (ERROR_PASSWORD_STORAGE + strPassword);
12  } ...
13 }
```

4.2.3 ScribeJava results. Table 5 illustrates the results concerning the last five constraints defined for the ScribeJava system. In this case, JOANA and PIDGIN obtain the same results for the **NVF** metric. They identify one violation for **C12** and two violations for **C13**. Again, JOANA is faster than PIDGIN.

Table 5: ScribeJava results

	JOANA		PIDGIN	
	NVF	**ET**	**NVF**	**ET**
C11	0	5821	0	392100
C12	1	1732	1	434100
C13	2	1836	2	476300
C14	0	1898	0	365100
C15	0	2419	0	360300
Total	3	13706	3	2027900

JOANA and PIDGIN identify one violation for **C12**. In debug mode, `token` information is written to the system log files in Line 5 of Listing 7. This way, users who have access to such log files are able to access token information. Depending on the kind of token used in debugging, this information might be sensitive (e.g., a developer uses his token to testing). Thus, we determine that this is a confidentiality violation.

Listing 7: signRequest method

```
1 void signRequest(OAuth1AccessToken token,
2   OAuthRequest request) {
3  ...
4  if (isDebug()) {
5   log("setting token to: %s", token);
6  }
7  ...
8 }
```

For **C13**, we analyze whether the `log` methods handle sensitive information. In the `signRequest()` method of Listing 7, the method responsible for writing information in the system log file is called in debug mode, passing the `token` as an argument. However, for **C13** we count two violations instead of one because this constraint has a rule stating that `token` information cannot be read or written by `log` methods.

Moreover, JOANA and PIDGIN do not find violations for **C11**, **C14**, and **C15**. Nonetheless, equally to the results for Blojsom and Lutece, JOANA is faster to analyze these constraints. Again, it is roughly 150 times faster than PIDGIN.

4.2.4 Answers to RQ2 and RQ2.1. Based on the results discussed on Sections 4.2.1, 4.2.2, and 4.2.3, we answer **RQ2** *confirming that JOANA and PIDGIN can detect flows from sensitive information to unauthorized methods for real systems.* However, our assessment shows that JOANA is able to

find more violations. Additionally, we answer **RQ2.1** *stating that JOANA is approximately 150 faster than PIDGIN when analyzing the same source code for the same constraints.*

4.3 Discussion

Considering our answers to RQ1 in Section 4.1, JOANA is the best choice when high recall is important. Therefore, it is more suitable for scenarios where we need to detect the highest number of existing violations (i.e., lowest number of false-negatives). However, this comes with the cost of lower precision and accuracy, that is, we need to manually analyze whether the violation warnings are really violations. To sum up, if we have enough resource to perform this manual analysis, it is safer to use JOANA.

Moreover, JOANA allows different settings to increase or decrease precision, as we stated in Section 4.1. In this way, we could set it to build a larger System Dependence Graph for this purpose with the cost of performance. Indeed, we plan to investigate these JOANA's optimizations in future work. Our preliminary results show that strongly increasing JOANA's precision turns its execution time unfeasible. For our current study, we decide to equally level the three tools to the same settings to avoid bias.

In contrast to JOANA, we recommend PIDGIN when higher precision and accuracy is important. Thereby, in case we do not have enough resource to manually analyze a potentially large amount of violation warnings, PIDGIN is the best choice. However, we might also miss important existing violations.

Furthermore, FlowDroid is designed specifically for Android applications [14]. Although its authors claim we can use it for project domains different than Android, we do not recommend FlowDroid for this purpose because it would demand resources to manually verify false warnings as well as we would miss potentially important violations. In addition, we would not detect violations where reflection is part of the implementation.

At last, we reported the violations we have found (Tables 3, 4, and 5) to their respective development team in order to validate our findings. However, until the conclusion of this work, we have had no response. We provide further detail on how we mitigate this limitation in Section 5.4.

5 THREATS TO VALIDITY

In this section, we discuss the threats to validity of our study. According to Wohlin et al [10], we organize the threats as Construct, Conclusion, External, and Internal validity.

5.1 Construct validity

Threats to construct validity cover issues related to the design of the assessment and its capacity to answer the research questions [10].

For our evaluation regarding the SecuriBench Micro (Section 4.1), we evaluate the selected IFC tools presented in Section 3.2, using only three metrics: *recall*, *precision*, and *accuracy*. However, we do not consider other metrics such as memory consumption. As future work, we plan to add an assessment considering additional metrics. In particular, memory consumption might be important when running IFC tools in a resource-limited environment.

5.2 Conclusion validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion [10].

In order to verify the information flow in the target systems, JOANA and PIDGIN use a graph called System Dependence Graph (SDG) [25]. These tools allow us to build SDGs with different levels of precision. For example, an *instance-sensitive* setting is less precise although faster than *object-sensitive* [23]. For this work, we use the *object-sensitive* setting to build SDGs within JOANA and PIDGIN. However, despite being conceptually equivalent, there are differences in the SDG construction implementation comparing these tools, which might introduce bias in our conclusions.

Another important threat regards the constraints we define in Section 3.4. For this work, we have acquired knowledge about Blojsom, Lutece, and ScribeJava source code and their collaborative development environment (i.e., GitHub and Jira[1]) to write our constraints. Nonetheless, we need to manually provide different input formats (representing the constraints) for the selected tools. This way, it might be easier to test a given constraint using JOANA rather than PIDGIN. As a matter of fact, this issue actually happens because we use a graphical user interface provided by JOANA to set the correct *Sources* and *Sinks*, which is faster than translating the constraint to the PIDGIN policy language.

However, our evaluation encompasses only the execution time, precision, recall, accuracy, number of violations found, and execution time. Therefore, the fact that JOANA and PIDGIN demand different inputs does not affect our conclusions. At last, our constraints and selected systems are the same for our assessment. Therefore, in this case, there is no bias for our results considering the three IFC tools.

5.3 External validity

Threats to the external validity consider the generalization of results [10].

To answer our research questions, we use a sample size of three open-source systems. So, we cannot generalize the results for projects of different sizes, domains, architectures, and those that adopt a version control system different from Git. However, the small sample of projects is due to the long time needed to study the systems in order to be able to write relevant constraints.

Other threat to validity regards the size of source code. Both JOANA and PIDGIN use System Dependence Graphs (SDG) [25] to analyze the information flow. However, static analysis using SDG do not support projects larger than 100 KLOC [11]. Therefore, we cannot assure that our results hold for projects larger than 100 KLOC. As an interesting future work, we could implement an additional tool to tame unnecessary SDG nodes and edges, which would make it possible to analyze such large projects. We could implement

[1] https://www.atlassian.com/software/jira

this tool based on the work of Ali and Lhoták, which aims at building application-only call graphs [1, 2].

5.4 Internal validity

Threats to the internal validity concern the fact that the assessment affects the results [10].

In this work, we study the target systems explained in Section 3.3 in order to understand which kind of sensitive information they handle. Then, we define constraints for these projects. However, we cannot guarantee that all sensitive information they handle are specified in our constraints. As a result of this limitation, we might have missed other violations for our three target projects. Since our evaluation considers the same set of constraints for PIDGIN and JOANA, this threat does not influence our conclusions about violation found and execution time.

As stated before, JOANA and PIDGIN use a data structure called System Dependence Graph (SDG). In order to correctly build this graph, we must provide as input an entry method to represent the root node [23, 25]. Since we manually select entry methods, we might have missed paths that could lead to violations. However, this problem is intrinsic to building SGD with JOANA and PIDGIN. Additionally, we use the same entry methods for these tools, which removes bias for their results in our assessment.

At last, we report the violations we have found for the developers responsible for each system: Blojsom, Lutece, and ScribeJava. Until the submission of this work, we have had no response. However, in order to minimize this limitation, we perform a manual analysis of the source-code for each violation identified by the tools. Additionally, we informally discussed and ratified the violations we found.

6 RELATED WORK

In this section, we discuss related work. First, we consider studies of IFC tools against the SecuriBench Micro (Section 6.1). At last, we discuss work on IFC tools assessment considering real software projects (Section 6.2).

6.1 IFC tools against SecuriBench Micro

In their work on security analysis of web applications, Tripp et al. motivate their tool named ANDROMEDA against the SecuriBench Micro [13]. Differently from us, they only consider the Aliasing group. Additionally, the authors do not present an assessment regarding precision, recall, and accuracy for ANDROMEDA.

On a similar work, Zanioli et al. evaluate a tool called Sails against three (out of ten) SecuriBench Micro test case groups [38]. Besides neglecting seven groups, the authors only provide the false alarm metric as output of their evaluation. In contrast, our study considers the ten existing SecuriBench Micro test case groups as well as we provide more metrics, as showed in Table 2.

Differently from the aforementioned related work, Hamann et al. [16] present a new benchmark suite, called IFSpec, to allow researchers to test their information-flow analysis tools targeting source code and bytecode for Java and Android applications. The authors evaluate their benchmark, which subsumes the SecuriBench Micro, on four IFC tools. Indeed, we plan to also test our three selected tools against IFSpec mainly regarding PIDGIN and Flowdroid, which have not been tested.

6.2 IFC tools and real software projects

Tripp et al. evaluate whether their BAYESDROID tool is able to find violations for real Android applications [36]. Since it runs for a specific domain only (Android), the authors do not need to manually define policies because they are the same for all applications (e.g., IMEI cannot flow to send message methods). Albeit Tripp et al. detect several violation warnings, they do not perform a comparison to other similar tools for real software systems (i.e., Android apps). In contrast, we compare the ability of JOANA and PIDGIN to detect violation warnings for a set of defined policies regarding real Java-written software systems.

Also, within Android context, Continella et al. [9] propose a new approach to detect privacy violations. Thus, the authors implement a tool, called AGRIGENTO, and evaluate it on more than one thousand Android apps. The evaluation of their tool compared AGRIGENTO against four well-known security-related Android tools. As above-mentioned, since policies for Android apps are fixed, the authors do not need to manually seek *Sources* and *Sinks*. In contrast, our work considers software systems of different domains, which demanded a large amount of time to manually find sensitive information. Consequently, our study considers only three real systems.

An additional related work regards the definition of a policy language to write privacy and security rules. Andrade et al. define Salvum for this purpose [3]. The authors then write a number of Salvum rules and use JOANA under the hood to check for violations considering nine open-source projects. Therefore, they evaluate their policy language instead of comparing diverse solutions. In contrast, our work aims at assessing different IFC tools to check which one is the most suitable depending on researchers' goals.

7 CONCLUSION

This work presented a comparative analysis of recall, precision and accuracy of three Information Flow Control tools: JOANA, FlowDroid, and PIDGIN. The main focus was on the tools capability of identifying sensitive information security violation. Although, run-time was also considered. The results allowed conclusions about the performance of the selected tools.

When executing the three projects in a well-known benchmark, the SecuriBench Micro, we could verify that JOANA and PIDGIN had similar results regarding recall, precision, and accuracy. These tools were able to identify most of the existing SecuriBench Micro violations. However, FlowDroid presented lower results, with a high number of false-negatives.

We observed that JOANA is the best tool when aiming for a high recall, especially when there are resources for manual analysis afterwards (due to the low precision). But, when aiming for a high precision and accuracy, the best tool is PIDGIN.

So, since JOANA and PIDGIN overcome FlowDroid, in order to demonstrate its applicability in real projects, we selected three systems to be tested: Blojsom, Lutece, and ScribeJava. These projects are Java-written, open-source, and handle sensitive information such as user's password and API's tokens. So, we defined five constraints, which contains confidentiality and integrity rules, for each system and executed them in both tools. On the one hand, JOANA was able to identify 8 violations that could compromise the privacy and security of these systems. On the other hand, PIDGIN identified 7 violations. With that, we can conclude that JOANA presented a result slightly superior to PIDGIN, considering the restrictions we wrote.

Furthermore, adding the run-time value into the comparison, we could notice that JOANA proved to be a good alternative, presenting similar results to the PIDGIN, but with a considerably lower run-time.

At last, as future work, we intend to replicate our benchmark study considering IFSpec [16], which provides sample programs for checking that IFC tools correctly classifies them as secure or insecure. In this context, we could ratify our current findings or bring other interesting results. We also plan to investigate why PIDGIN and JOANA have such a different execution time considering the same target system and constraint. Therefore, we need to understand the differences of their System Dependence Graph construction. The results of this study would help other researchers to either improve PIDGIN or prevent mistakes that lead to such performance loss. Additionally, we plan to use PIDGIN and JOANA to detect violations for more target systems to ratify our findings. Another interesting future work would be to add other IFC tools, such as the Checker Framework[2], to our assessment, which could also bring insights to help other researchers to choose the most suitable IFC tool for their scenarios.

REFERENCES

[1] Karim Ali and Ondřej Lhoták. 2012. Application-only call graph construction. In *European Conference on Object-Oriented Programming*. 688–712.

[2] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*. 378–400.

[3] Rodrigo Andrade and Paulo Borba. 2020. Privacy and security constraints for code contributions. *Journal of Software: Practice and Experience* 50, 10 (2020), 1905–1929.

[4] Ken Biba. 1975. Integrity considerations for secure computer systems. *Mitre Corporation* (1975).

[5] Blojsom. 2021. *Blojsom*. https://sourceforge.net/projects/blojsom/

[6] Stephen Chong and Andrew C. Myers. 2004. Security Policies for Downgrading. In *Conference on Computer and Communications Security*. 198–209.

[7] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *USENIX Security Symposium*. 1–16.

[8] CWE Community. 2021. *CWE - Common Weakness Enumeration*. https://cwe.mitre.org/

[9] Andrea Continella et al. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *ISOC Network and Distributed System Security Symposium*.

[10] Claes Wohlin et al. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[11] David Binkley et al. 2007. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems* 30, 1 (2007), 1–34.

[12] Omer Tripp et al. 2009. TaJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

[13] Omer Tripp et al. 2013. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering*. 210–225.

[14] Steven Arzt et al. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Notices* 49, 6 (2014), 259–269.

[15] Salvatore Guarnieri et al. 2011. Saving the World Wide Web from Vulnerable JavaScript. In *International Symposium on Software Testing and Analysis*. 177–187.

[16] Tobias Hamann et al. 2018. A uniform information-flow security benchmark suite for source code and bytecode. In *Gruschka Nordic Conference on Secure IT Systems*. 437–453.

[17] Victor Basili et al. 1994. The goal question metric approach. In *Encyclopedia of Software Engineering*, John J. Marciniak (Ed.). Wiley, New Jersey, 528–532.

[18] William Enck et al. 2010. TaintDroid: An Information-Flow Tracking System for Real-time Privacy Monitoring on Smartphones. In *USENIX Symposium on Operating Systems Design and Implementation*.

[19] Simple Logging Facade for Java. 2004. *SLF4J -Simple Logging Facade for Java*. http://www.slf4j.org

[20] OWASP Foundation. 2021. *OWASP - Open Web Application Security Project*. https://owasp.org/

[21] Karlsruher Institut fur Technologie. 2021. *JOANA (Java Object-Sensitive Analysis) - Information Flow Control Framework for Java*. https://pp.ipd.kit.edu/projects/joana/

[22] Jürgen Graf, Martin Hecker, and Martin Mohr. 2013. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Work. Conf. Program. Languages*. 123–138.

[23] Christian Hammer and Gregor Snelting. 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security* 8 (2009), 399–422.

[24] Daniel Hedin and Andrei Sabelfeld. 2012. A Perspective on Information-Flow Control. *Software Safety and Security* 33 (2012), 319–347.

[25] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems* 12 (1990), 26–60.

[26] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. *SIGPLAN Notice* 50, 6 (2015), 291–302.

[27] Benhamin Livshits. 2021. Securibench Micro. http://suif.stanford.edu/~livshits/work/securibench-micro/

[28] V. B. Livshits and M. S. Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*. 271–286.

[29] Lutece. 2021. *Lutece*. https://github.com/lutece-platform

[30] A. Mettler, D. Wagner, and T. Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed System Security Symposium*. 357–374.

[31] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *ACM Symposium on Principles of Programming Languages*. 228–241.

[32] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. 2021. *Jif: Java information flow*. http://www.cs.cornell.edu/jif

[33] J. Newsome and D. Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Net. and Dist. Sys. Security Symp.*

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

[35] Scribejava. 2021. *Simple OAuth library for Java*. https://github.com/scribejava/scribejava

[36] Omer Tripp and Julia Rubin. 2014. A Bayesian Approach to Privacy Enforcement in Smartphones. In *USENIX Security Symposium*. 175–190.

[37] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. *SIGPLAN Notices* 47, 1 (2012), 85–96.

[38] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. 2012. SAILS: Static Analysis of Information Leakage with Sample. In *ACM Symposium on Applied Computing*. 1308–1313.

[2] https://checkerframework.org/