Using JOANA for Information Flow Control in Java Programs — A Practical Guide

Jürgen Graf, Martin Hecker, Martin Mohr

Programming Paradigms Group
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe
{graf,martin.hecker,martin.mohr}@kit.edu

Abstract: We present the *JOANA* (Java Object-sensitive ANAlysis) framework for information flow control (IFC) of Java programs. JOANA can analyze a given Java program and guarantee the absence of security leaks, e.g. that a online banking application does not send sensitive information to third parties. It applies a wide range of program analysis techniques such as dependence graph computation, slicing and chopping of sequential as well as concurrent programs. We introduce the Java Web Start application *IFC Console* and show how it can be used to apply JOANA to arbitrary programs in order to specify and verify security properties.

1 Introduction

Conventional access control mechanism control what data a program may access, but what happens with this data inside the program, once access has been granted? Information flow control (IFC) aims to answer this question. For example, an email application shall both read data from an address book and send other data to the network, but it should not send address book data over the network. With IFC one can check if the email program may conduct such forbidden behaviour or not.

Much work in the area of IFC has focused either on building theoretical foundations for proveable security guarantees or on practical tools that can detect a subset of all information leaks in a program. We present a static IFC analysis framework named JOANA that aims to combine both directions. JOANA is the first tool that can verify the absence of possibilistic and even probabilistic leaks for full Java bytecode, including exceptions, dynamic dispatch and inheritance. It can deal with sequential [HS09] as well as multi-threaded [GS12, GHMN13] programs and applies to medium sized programs with around 30-50kLoC and in some cases up to 100kLoC [Gra09, Gra10]. A machine-checked proof [WL10, WLS09] guarantees that the underlying algorithms are sound and no potential information flow is missed.

The frontend of JOANA builds upon the WALA program analysis framework[1]. WALA

[1] `http://wala.sf.net/`

comes with an intermediate representation (IR) in SSA-form, a variety of dataflow solvers and a points-to analysis framework. WALA helps to resolve dynamic dispatch, detect potential exceptions and compute side-effects of method invocations. It can deal with Java bytecode, Java source code and javascript programs. Currently we work on support for Dalvik bytecode of the Android platform. JOANA mostly operates on the IR and therefore may be extended to support other languages with relatively little effort.

Our backend is based on *dependence graphs* which capture dependencies between program statements in form of a graph. Those graphs are called *program dependence graphs* (PDG) or, to be more precise, *system dependence graphs* (SDG). Previous work already showed that PDG-based IFC [Ham10] can be useful in practice and has the great advantage that only minimal user effort is needed. This work focuses even more on practicability. We introduce a UI for our IFC framework named *IFC Console* and explain how it can be used to analyze information flow. The source code of JOANA, including IFC Console, is available at `http://joana.ipd.kit.edu` and may be used freely for research purposes.

The major contributions of this paper are:

- We introduce the Web Start application IFC Console that enables developers to check information flow properties of their own programs with little effort.
- We discuss the benefits of using system dependence graphs for IFC analysis in terms of precision and ease of use.
- We discuss the relevant properties for IFC in a concurrent setup.
- Two case studies show how IFC Console can be applied to a single- and a multi-threaded program.

We start with a more detailed introduction to information flow control and show how it can be achieved with the help of dependence graphs in section 2. Then we introduce IFC Console in section 3 and show how it can be applied to sequential and concurrent programs in section 4. Section 5 discusses related work and section 6 concludes the presented work and provides an outlook for future work.

2 Information flow control

Information flow control is concerned with the flow of information inside a program. It is used to prevent leakage of secret information to public output channels, thus to ensure *confidentiality* and it is also used to verify the *integrity* of a program, which is the dual property to confidentiality: It ensures that no unverified input may influence critical computation or secret values. In order to verify these properties, it does not suffice to check where secret or public data is copied from or moved to. In addition, the effects that the value of the data may have on the execution of the program need to be tracked.

For example in figure 1 we do not want an attacker to gain any information about the secret input value by observing the program output. This program contains three `print` statements that produce output. While the statement in line 7 does not reveal any information about the input, the other two statements do. Line 3 directly prints the value of the input and is therefore called a *direct leak*. The effect of the output in line 5 is more subtle, as it does not print anything related to the input value. However it is only executed if the input is an even number. Hence, the attacker is able to infer that the input is even if he sees the output produced by line 5. These kind of information leaks are called *indirect leaks*.

```
1 void main():
2   int secret = input();
3   print(secret);                  // direct leak
4   if (secret % 2 == 0) {
5     print("secret␣is␣even");     // indirect leak
6   }
7   print("Hello␣World.");         // no leak
```

Figure 1: A program fragment with a direct and an indirect information leak.

An IFC analysis has to detect direct as well as indirect information flow and it needs to know which information is considered secret and what is considered a public output in order to check for confidentiality. In JOANA this is achieved by annotating variables or statements with a *security label*. For the example above we need two different labels: *high* (secret) and *low* (public). Statement 2 is labeled as high input and statements 3, 5 and 7 are labeled as low output. The IFC analysis then checks if any information flow from high input to low output is possible. In contrast to other IFC analyses that are often based on type systems, only statements corresponding to input or output need to be labeled. JOANA propagates the labels for other statements automatically.

This approach is not restricted to only two security labels. For more complex IFC analyses, it supports an arbitrary number of labels. They have to be specified in form of a *security lattice* that defines a partial order on the labels. Any flow from a statement labeled l_1 to a statement labeled l_2 is considered legal iff $l_1 \leq l_2$. In the remainder of this work we will use the standard two-valued lattice $low \leq high$.

2.1 Sequential IFC with dependency graphs

Our IFC analysis[HS09] uses SDGs to conservatively approximate all possible information flow inside a program. A SDG is a language-independent representation of dependencies between statements of a program. It contains nodes for each statement of the program and edges between them if one statement depends on the other one. In sequential programs these dependencies are either direct or indirect dependencies. Direct dependencies, also called data dependencies, occur whenever a statement produces a value, e.g. writes a variable, that the other statement potentially may read. Indirect dependencies between statements occur if the outcome of the execution of one statement decides if the other is executed, e.g. the condition of an if-clause decides if the statements in the body of the if-clause are executed. A machine-checked proof [WL10, WLS09] shows that the SDG is a conservative approximation of the effects of sequential programs and that our IFC algorithm is sound.

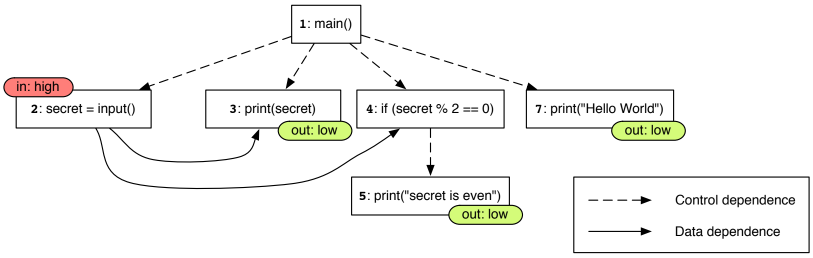


Figure 2: The dependence graph for figure 1 with annotated security labels.

Figure 2 shows a simplified version of the SDG for the program in figure 1. It contains data dependencies between statement 2, where the input is written to variable `secret`, and statements 3 and 4, that read the value of `secret`. Control dependencies between the method entry point and statements 2, 3, 4 and 7 signal that those statements are only executed when `main` is called. Statement 5 however is control dependent on the if-clause in statement 4, because its execution depends on the evaluation of this statement. The input and output statements are annotated with security labels *high* and *low* as described in the previous section.

The SDG based IFC analysis then checks if the graph contains a path from a statement labeled *high* to a statement labeled *low*. To achieve this, we use a special form of conditional reachability analysis that applies *slicing*[Kri03, RHSR94, Wei81] and *chopping*[Gif11, RR95] techniques. This enables us to restrict the set of possible paths in the graph to a subset of *valid paths*, which helps to significantly reduce the number of false alarms. A valid path is a path in the SDG that respects additional conditions, like e.g. context-sensitivity. The example contains two valid paths that correspond to illegal flow: $2 \rightarrow 3$ and $2 \rightarrow 4 \rightarrow 5$. Thus our analysis reports two potential security violations.

In case no violations are found, the program is considered safe. Hence our analysis can guarantee the absence of security violations, but it can only detect the potential presence of leaks, because false alarms are possible due to conservative approximations in our analysis algorithms.

JOANA contains many optimizations that improve analysis precision and thus help to reduce the number of false alarms:

points-to information We use points-to analysis to compute side-effects across method boundaries and to approximate the effects of late binding. Various precision options are available.

exception analysis We include an analysis that detects exceptions that never occur. This is very essential in Java, because almost any instruction may potentially throw an exception, e.g. any object field-access may throw a *NullPointerException* if the referenced object is *null*.

context-sensitive We distinguish between different calls to the same method and offer unlimited[2] context-sensitivity through interprocedural program slicing [RHSR94].

object-sensitive We distinguish different instances of the same class and methods invoked on different instances.

field-sensitive We distinguish different fields of an object instance through modelling accessible fields in form of an object graph [Gra10].

flow-sensitive The dependencies inside an SDG respect the execution order. It contains only dependencies between two statements $s_1 \rightarrow s_2$ if s_2 may be executed after s_1.

2.2 IFC for concurrent programs

Concurrent Java programs consist of multiple threads that execute in parallel. Threads can communicate through shared variables. In addition to the previously mentioned direct and indirect leaks, these so-called *interferences* between threads introduce two new kinds of information leaks: *possibilistic* and *probabilistic* leaks.

```
1 void thread_1():                     4 void thread_2():
2     x = 0;                           5     secret = input();
3     print(x);                        6     x = secret;
```

Figure 3: Two threads with a shared variable `x` that contain a possibilistic leak.

A possibilistic leak results in illegal flow depending on the order in which statements of different threads are executed. The example in figure 3 has a possibilistic leak. The program consists of two threads that communicate through a shared variable `x`. The `print` statement in line 3 does leak the value of the secret input in line 5 if line 6 is executed after line 2 and before line 3.

```
1 void thread_1():                     4 void thread_2():
2     x = 0;                           5     secret = input();
3     print(x);                        6     while (secret != 0)
                                       7         secret--;
                                       8     x = 1;
```

Figure 4: Two threads with a shared variable `x` that contain a probabilistic leak.

Probabilistic leaks are even trickier. A secret value can potentially influence the probability of the order in which statements that influence public outputs are executed. An attacker that can run the program with the same secret input multiple times is able to infer information about the secret value through a statistical analysis of observable outputs. Figure 4 illustrates this problem. The statement in line 3 prints the value of variable `x`. Depending on the

[2] Multiple recursive calls are not distinguished.

execution order of the statements in line 2 and 8 it prints either `0` or `1`. However the probability that line 8 is executed before the print statement depends on the value of `secret`. The bigger the value of `secret` is, the more time is spent executing the **`while`** loop in lines 6-7 and thus the less likely it is that line 8 is executed before the print statement. So if the attacker observes a huge number of program runs and keeps track of ratio between output `0` and `1` he can infer if the value of `secret` is a large number.

Albeit probabilistic leaks seem to pose more of a theoretical than an actual security thread, this is far from true. With additional knowledge about the scheduling algorithm the attacker is in some cases able to infer concrete values. These leaks have already been successfully used to break well known encryption algorithms [Koc96].

JOANA is able to detect possibilistic as well as probabilistic leaks[Gif12]. It computes possible interferences between threads with the help of points-to and may-happen-in parallel (MHP) analyses. We apply a special version of slicing [Kri03, RHSR94, Wei81] and chopping [Gif11, RR95] algorithms optimized for concurrent IFC. This allows us to achieve precise results that are time-, join- and in future versions even lock-sensitive[GHMN13].

3 IFC Console

IFC Console is a graphical user interface which hides most of JOANA's internals. It simplifies SDG construction and the annotation of SDG nodes with security labels. Instead the user can annotate program artifacts such as attributes, method parameters or bytecode instructions and an integrated heuristic selects the appropriate nodes.

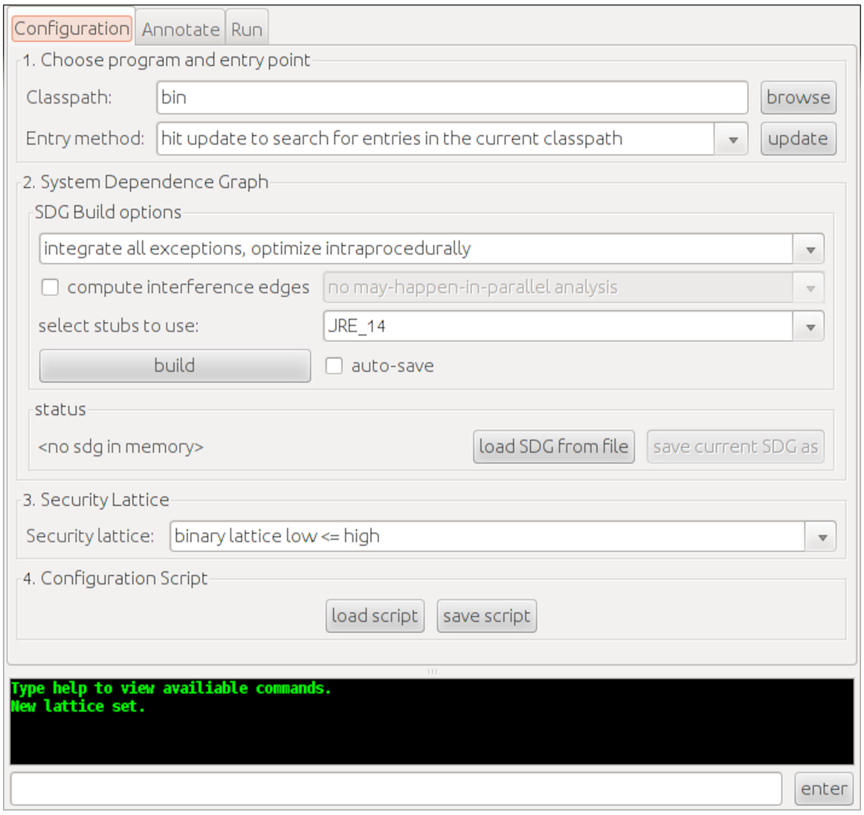
3.1 A Quick tour through the interface

The graphical user interface in figure 5 is divided into two parts. The upper part shows options for SDG construction and general configuration and also contains additional tabs for security label annotation (figure 6) and running the IFC analysis (figure 7). The lower part shows a console that displays detailed output and can be used to enter advanced commands. Every action the user performs is recorded as a command in the console. This allows the user to save his actions to a script file, that can be loaded and automatically replayed.

Configuration Tab The configuration tab in figure 5 is used to select the program to analyze (1.), build or load a SDG for the program (2.), select the security lattice (3.) and save or replay scripts of previous actions (4.).

In order to select a program, the user sets the class path to a directory or .jar file that contains the compiled .class files. Then he hits "update" and selects the `main` method he wishes to analyze in the drop down list.

In the next step the user selects the desired SDG computation options. He can choose how the analysis should handle the effects of exceptions and the treatment of multi-threaded



Classpath: bin

Entry method: hit update to search for entries in the current classpath

integrate all exceptions, optimize intraprocedurally

□ compute interference edges: no may-happen-in-parallel analysis

select stubs to use: JRE_14

□ auto-save

<no sdg in memory>

Security lattice: binary lattice low <= high

```
Type help to view availiable commands.
New lattice set.
```

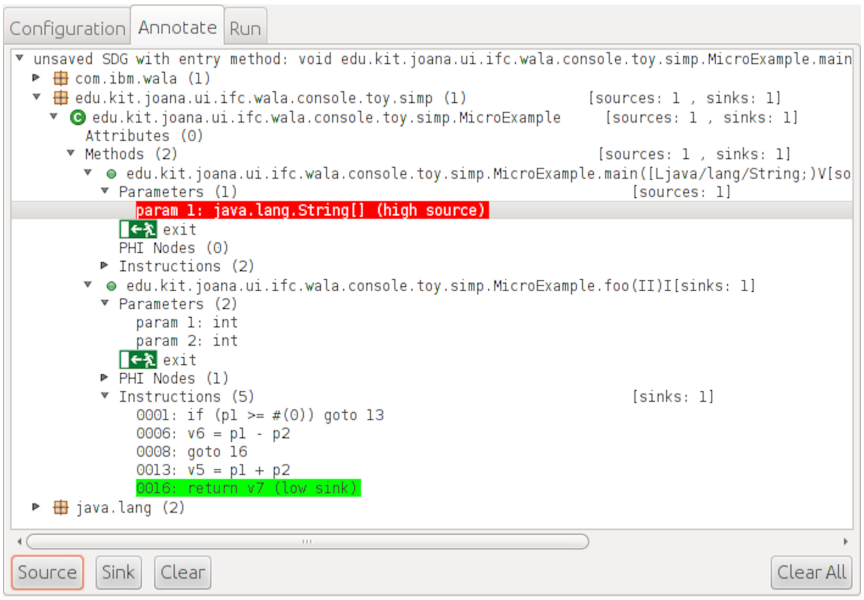
Figure 5: The configuration tab and the console view of the IFC Console.

programs. The exception analysis options are:

integrate exceptions without optimization No additional exception analysis is performed. This is the least precise option that does not detect any impossible exceptions. For example every field access is treated as it may or may not cause a NullPointerException, even subsequent accesses to the same field or references to the `this` pointer.

integrate exceptions, optimize intra-/interprocedurally An exception analysis is performed that detects impossible and also guaranteed exceptions. For example, JOANA identifies field reading accesses where the field can never be null. The interprocedural analysis is more precise but also more time-consuming.

In order to analyze multi-threaded programs, the check box “compute interference edges” has to be selected. Then the user can choose between various precision options of the may-happen-in-parallel (MHP) analysis. The least precise option is no MHP, whereas “precise may-happen-in parallel analysis” takes a closer look at the control-flow of the program and in particular the life span of its threads. For example, it detects that a thread



```
▼ unsaved SDG with entry method: void edu.kit.joana.ui.ifc.wala.console.toy.simp.MicroExample.main
  ▶ com.ibm.wala (1)
  ▼ edu.kit.joana.ui.ifc.wala.console.toy.simp (1)            [sources: 1 , sinks: 1]
    ▼ edu.kit.joana.ui.ifc.wala.console.toy.simp.MicroExample     [sources: 1 , sinks: 1]
        Attributes (0)
      ▼ Methods (2)                                              [sources: 1 , sinks: 1]
        ▼ edu.kit.joana.ui.ifc.wala.console.toy.simp.MicroExample.main([Ljava/lang/String;)V[so
          ▼ Parameters (1)                                          [sources: 1]
              param 1: java.lang.String[] (high source)
            exit
            PHI Nodes (0)
          ▶ Instructions (2)
        ▼ edu.kit.joana.ui.ifc.wala.console.toy.simp.MicroExample.foo(II)I[sinks: 1]
          ▼ Parameters (2)
              param 1: int
              param 2: int
            exit
          ▶ PHI Nodes (1)
          ▼ Instructions (5)                                        [sinks: 1]
              0001: if (p1 >= #(0)) goto 13
              0006: v6 = p1 - p2
              0008: goto 16
              0013: v5 = p1 + p2
              0016: return v7 (low sink)
  ▶ java.lang (2)
```

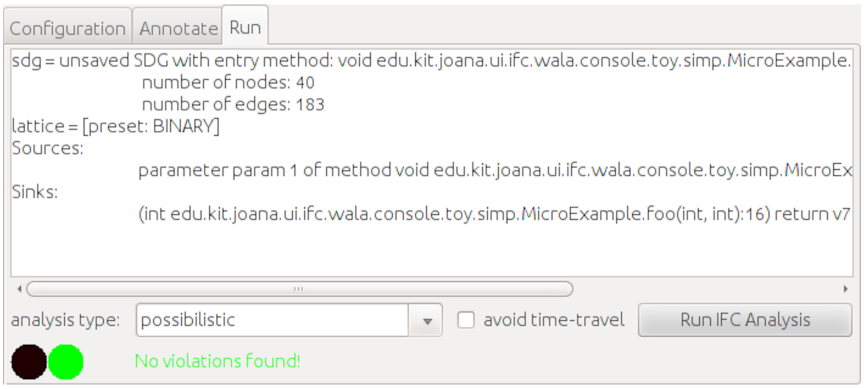
Figure 6: The annotation tab of the IFC Console.

cannot interfere with another thread before it was started or after it has been joined. The user can also choose between *stubs* for different Java runtime environments. Basically, these stubs include predefined models of native methods deep down in the Java standard library. We strongly suggest to analyze JRE 1.4 programs, as later versions of the JRE are far bigger, which leads to an increased runtime of the analysis. When the configuration of the SDG building options is finished, a click on "build" starts the SDG building process.

There is also the possibility to save and load a previously built SDG. The option "auto-save" stores the SDG directly after it has been built to a file with an auto-generated name in the current working directory.

The IFC part of JOANA supports multiple security levels which are arranged in a lattice. The graphical user interface offers three simple lattices, which should suffice for simple cases, the default is the two-valued lattice *low* ≤ *high*.

Annotation Tab The annotation tab in figure 6 provides a tree-like view of the program under analysis. On the top-level the different packages of the program are shown. Unfolding a package shows the classes it contains, unfolding a class shows the attributes and methods it contains and unfolding a method shows its parameters and instructions. Note that not the whole program is shown, but only those parts which are reachable from the selected entry method. A node of the program tree can be turned into an information source or an information sink and annotate it with a selectable security level.



```
sdg = unsaved SDG with entry method: void edu.kit.joana.ui.ifc.wala.console.toy.simp.MicroExample.
                    number of nodes: 40
                    number of edges: 183
lattice = [preset: BINARY]
Sources:
                   parameter param 1 of method void edu.kit.joana.ui.ifc.wala.console.toy.simp.MicroEx
Sinks:
                   (int edu.kit.joana.ui.ifc.wala.console.toy.simp.MicroExample.foo(int, int):16) return v7
```

No violations found!

Figure 7: The analysis tab of the IFC Console.

Analysis Tab The analysis tab in figure 7 displays a summary of the analysis options: It shows the size of the SDG, the selected lattice and all annotated sources and sinks. Also it offers to choose between a “possibilistic” and a “probabilistic” IFC algorithm. This is only relevant for multi-threaded programs, for single-threaded programs the default option “possibilistic” is fine. The probabilistic algorithm detects the same leaks as the possibilistic approach and it includes additional probabilistic leaks, which can only occur in multi-threaded programs. Apart from selecting the IFC type, the user can also choose to improve precision by disallowing *time-travels* ([Kri03, Gif12]). This affects both the possibilistic and the probabilistic types. Disallowing time-travels essentially means that the algorithm will discard security leaks made possible only by inconsistent program runs.

The user starts the IFC analysis with the “run” button. When finished, the console part of the GUI shows a report of the detected leaks. Additionally a green light flashes up, if there are no security leaks and the program is guaranteed to be noninterferent. Otherwise a red light signals that potential leaks were detected.

4 Examples

4.1 Sequential IFC - Praktomat

We use a simplified version of the Praktomat system to show how JOANA can be applied to guarantee integrity. Praktomat is a browser based application that allows students to submit their solutions to a given programming task. Then Praktomat runs predefined checks on the submitted solution, e.g. it checks if the code compiles or if the Java Code Conventions are violated. On the one hand this information helps the student to improve his solution and on the other hand these results are also used by the tutor that evaluates the solution later on.

For this example we focus on the way the predefined checks should operate. Their results are often crucial for the evaluation of the submissions, as manual checks are not feasible for large amounts of submissions and they can also not provide instant feedback to the submitting student. As tutors rely on their results, these checks need to produce fair and reproducible results. A malicious check for example may treat submissions from a specific student in different way then all other submissions - not showing detected failures of this special student.

```
1  public class Praktomat {
2    public static class Submission {
3
4      public int matrNr;
5      public String code;
6
7      public Submission(int matrNr,
8          String code) {
9        this.code= code;
10       this.matrNr = matrNr;
11     }
12   }
13
14   public static class Review {
15
16     public Submission sub;
17     public int failures;
18
19     public Review(Submission sub,
20         int failures) {
21       this.sub = sub;
22       this.failures = failures;
23     }
24   }

26   public static Review runChecks(Submission sub) {
27     int failures = 0;
28
29     if (sub.code.contains("System.err.println")) {
30       failures++;
31     }
32     if (sub.code.contains("catch␣IOException")) {
33       failures += 2;
34     }
35     if (sub.matrNr == 4711) {
36       failures = 0;
37     }
38
39     return new Review(sub, failures);
40   }
41
42   public static void main(String argv[]) {
43     Submission sub = new Submission(2331,
44       "System.out.println(\"Hello␣world.\");");
45     Review r = Praktomat.runChecks(sub);
46     System.out.println(r.failures);
47   }
48 }
```

Figure 8: A simplified version of the automated program submission system Praktomat. It automatically checks submitted programs for predefined failures and helps the tutor to review student submissions. The underlined code violates the security property, as it hides detected failures for a specific student.

We can detect these kind of malicious checkers with the help of JOANA. To achieve this, we specify the information flow property all checkers need to guarantee as follows: The number of detected program failures must not depend on the identity of the submitting student. The code in figure 8 shows a simplified version of the Praktomat system. It contains a class `Submission` to model student submissions and a class `Review` to model the result of the submission checker. The code of the checker is in method `runChecks` in lines 26-40. It is called once from `main` method to perform checks for a single submission. The attribute `matrNr` of class `Submission` stores the identity of the submitting student. We classify this information as *secret* and the `failure` counter in class `Review` as *public*. Then we can use JOANA to verify if the given program is *noninterferent*[GM82] and thus the number of detected failures does not depend on the id of the submitting student. For the given program this is not the case, as lines 35-37 contains a special treatment for the student with the id 4711. JOANA is able to detect this leak. Also when these lines are removed from the program, the checker result no longer depends on the student id and JOANA can verify noninterference for this example.

Using IFC Console We now describe briefly the necessary steps to analyze this example. Specify the appropriate class path, click on the "update" button and select the main method of the class `Praktomat` as entry method. The SDG building options do not have to be changed, so that you can directly build the SDG by clicking on the "build" button. Switch to the annotation tab. Select the attribute `matrNr` of the inner class `Submission` as high source and the attribute `failures` of the inner class `Review` as low sink. In the analysis tab, nothing needs to be configured, since the given program is not multi-threaded, so you can simply run the analysis. As explained before, the analysis finds several leaks. If you remove lines 35-37, you should get no leaks.

4.2 Concurrent IFC - EuroStoxx

Figure 9 shows a program that manages a stock portfolio of Euro Stoxx 50 entries[3]. The program consists of four threads, coordinated by an additional main thread. The program first starts the `Portfolio` and `EuroStoxx50` threads concurrently, where `Portfolio` reads the user's stock portfolio from storage and `EuroStoxx50` retrieves the current stock rates. When these threads have finished, threads `Statistics` and `Output` are run concurrently, where `Statistics` calculates the current profits and `Output` incrementally prepares a statistics output. After these threads have finished, the statistics are displayed, together with a pay-per-click commercial. An ID of that commercial is sent back to the commercials provider to avoid receiving the same commercial twice. The portfolio data, `pfNames` and `pfNums`, is secret, hence the Euro Stoxx request by `EuroStoxx50` and the message sent to the commercials provider should not contain any information about the portfolio. As `Portfolio` and `EuroStoxx50` do not interfere, the Euro Stoxx request does not leak information about the portfolio. The message sent to the commercials provider is not influenced by the values of the portfolio, too, because there is no explicit or implicit flow from the secret portfolio values to the sent message. Furthermore, the two outputs have a fixed relative ordering, as `EuroStoxx50` is joined before `Output` is started. Hence, the program is considered secure.

Using IFC Console Analyzing the concurrent example introduced in 4.2 requires different options because it is multi-threaded. After selecting the appropriate class path and entry method, you have to check "compute interference edges" and choose a MHP analysis. Use the precise MHP analysis, otherwise you will get many false alarms simply because joins are not taken into account.

As mentioned in the example, the portfolio data is secret, so calls to `getPFNames()` and `getPFNums()` in the run method of the class `Mantel00Page10$Portfolio` have to be annotated as high sources. To verify that the secret data cannot influence the commercial messages, which are written into the output referenced by the attribute `Mantel00Page10.nwOutBuf`, it suffices to annotate the calls to its flush methods. These are located in the run method of the class `Mantel00Page10$EuroStoxx50` and in the main method of the class Mantel00Page10,

[3] The description of this program has been taken from [Gif12]

```
public class Mantel00Page10 {
  static class Portfolio extends Thread {
    int[] esOldPrices;
    String[] pfNames;
    int[] pfNums;
    String pfTabPrint;

    public void run() {
      pfNames = getPFNames();            // high
      pfNums = getPFNums();              // high
      for (int i = 0; i < pfNames.length; i++) {
        pfTabPrint += pfNames[i]+"|"+pfNums[i];
      }
    }

    int locPF(String name) {
      for (int i = 0; i < pfNames.length; i++) {
        if (pfNames[i].equals(name)) {return i;}
      }
      return -1;
    }
  }

  static class EuroStoxx50 extends Thread {
    String[] esName = new String[50];
    int[] esPrice = new int[50];
    String coShort;
    String coFull;
    String coOld;

    public void run() {
      try {
        nwOutBuf.append("getES50");
        nwOutBuf.flush();                // low
        String nwIn = nwInBuf.readLine();
        String[] strArr = nwIn.split(":");
        for (int j = 0; j < 50; j++) {
          esName[j] = strArr[2 * j];
          esPrice[j] =
            Integer.parseInt(strArr[2 * j + 1]);
        }
        // commercials
        coShort = strArr[100];
        coFull = strArr[101];
        coOld = strArr[102];
      } catch (IOException ex) {}
    }
  }

  static class Output extends Thread {
    public void run() {
      for (int m = 0; m < 50; m++) {
        while (s.k <= m); // busy-wait sync
        output[m] = m + "|" + e.esName[m] + "|"
          + e.esPrice[m] + "|" + s.get(m);
      }
    }
  }

  static class Statistics extends Thread {
    int[] st = new int[50];
    volatile int k = 0;

    public void run() {
      k = 0;
      while (k < 50) {
        int ipf = p.locPF(e.esName[k]);
        if (ipf > 0) {
          set(k, (p.esOldPrices[k] - e.esPrice[k])
            * p.pfNums[ipf]);
        } else {
          set(k, 0);
        }
        k++;
      }
    }
    synchronized void set(int k, int value) {
      st[k] = value;
    }
    synchronized int get(int k) {
      return st[k];
    }
  }

  static Portfolio p = new Portfolio();
  static EuroStoxx50 e = new EuroStoxx50();
  static Statistics s = new Statistics();
  static Output o = new Output();
  static String[] output = new String[50];
  static BufferedWriter nwOutBuf = new BufferedWriter(
    new OutputStreamWriter(System.out));
  static BufferedReader nwInBuf = new BufferedReader(
    new InputStreamReader(System.in));

  public static void main(String[] args)
      throws Exception {
    // get portfolio and eurostoxx50
    p.start(); e.start();
    p.join();  e.join();
    // compute statistics and generate output
    s.start(); o.start();
    s.join();  o.join();
    // display output
    stTabPrint("No.\t␣|␣Name\t␣|␣Price\t␣|␣Profit");
    for (int n = 0; n < 50; n++) {
      stTabPrint(output[n]);
    }
    // show commercials
    stTabPrint(e.coShort + "Press␣#␣to␣get␣info");
    char key = (char) System.in.read();
    if (key == '#') {
      System.out.println(e.coFull);
      nwOutBuf.append("shownComm:" + e.coOld);
      nwOutBuf.flush();                  // low
    }
  }
}
```

Figure 9: A possibilistic and probabilisitic secure program from Mantel et al. [MSK07], adapted to Java in [Gif12]. JOANA is the first tool able to automatically proof the absence of probabilistic leaks for this example.

respectively.

In the analysis tab, choose "probabilistic (with precise mhp)" as analysis type. Running the IFC checker yields no violations.

Note, that it is crucial to select "probabilistic (with precise mhp)" as analysis type. If you select "probabilistic (with simple mhp)", lots of leaks will be found due to many spurious interference edges. Since direct and indirect leaks are included in the probabilistic IFC checker, possibilistic IFC also accepts the program.

5 Related work

Tools for language based IFC and dependence analysis Several other tools for information flow control and dependence analysis of Java programs are available. These tools differ in how much user guidance they require, and which language features they support. Tools like Jif[Mye99, MZZ+01] extend Java with *security types*. In addition to their standard Java type such as `int`, the user annotates variables, fields and method signatures with labels that restrict how information may flow. Jif then checks if these security type annotations are valid and hence if the program is secure. Since Jif supports security type inference only for local variables, in order to check *any* information flow property, the user is usually required to annotate the whole program with security types. It is not enough to only mark those program points where information is read in / written out. For this reason, and since Jif does not support Java features such as concurrency, it is impractical to use Jif or approaches based on Jif[CVM07] with existing code bases.

To alleviate the effort of manually annotating large parts of the program with security type annotations, more elaborate type inference algorithms have been proposed[ST07], but as of yet, there is no practical implementation for full Java.

Similarly to JOANA, the Indus[RH07] tool utilizes several auxiliary analyses to provide SDGs for concurrent Java. These can be used for *slicing*[Wei81], which in turn is used in order to reduce the state space in model checking applications. Unlike JOANA, no explicit support for IFC is provided.

Commercial security scanners like AppScan[4] scan the code of web applications for security vulnerabilities and detect many bugs like, SQL injection, error prone coding practices and other security leaks. Tripp et al.[TPF+09] integrate a taint analysis for javascript into AppScan. Their approach is based on the WALA framework and applies *hybrid thin-slicing*. Their tool scales well and can detect many security leaks in almost arbitrarily large programs with only few false alarms. Due to thin-slicing they miss indirect information leaks and thus cannot guarantee noninterference.

Bodden [Bod12] presents an IFC tool tailored for software production lines that applies the new IFDS/IDE implementation of the SOOT program analysis framework[5]. It can deal with conditionally compiled code in an efficient way and includes a nice GUI. However this tool can only detect a very specific kind of information flow, namely direct leaks through

[4] `http://www.ibm.com/software/awdtools/appscan/developer`

[5] `http://www.sable.mcgill.ca/soot/`

data flow in variables. At this time it cannot detect indirect flow through branches, data flow through heap allocated objects or any kind of concurrency related leaks.

Guarnieri et al. [GPT^{+}11] use a demand-driven taint analysis based on access paths that detects security leaks in javascript websites. They detect direct as well as indirect leaks but lack support for probabilistic leaks. Their approach scales well and can be applied in real world scenarios, but it lacks a sound approximation of the effects of the `eval` function, which is inherently difficult for a static approach.

Therefore Seth et al. [JCSH11] propose a combination of static and dynamic analysis for javascript that can detect illegal information flow in programs with a sound approximation of the `eval` function.

Aside from type-system and SDG based IFC analyses, in [GS05] an abstract interpretation approach to information flow analysis for Java bytecode is proposed. Each bytecode instruction is abstractly interpreted by its direct information flow. Together with the instruction's *scope* (which is similar to control dependencies in SDGs), this is sufficient to obtain a program's information flow. The proposed analysis is not object sensitive and does not handle concurrency.

```
if (secret % 2 == 0) {
    public = 42;
} else {
    public = 42;
}
```

(a)

```
secret = Math.abs(...);
if (secret % 2 == 0) {
    public = (secret % 2);
} else {
    public = (secret % 2) - 1;
}
```

(b)

```
if (secret > 17 && secret < 17) {
    public = 42;
}
```

(c)

Figure 10: Semantically secure program fragments

Non-interference and verification of semantic properties Just like JOANA, the tools mentioned so far are imprecise in the sense that they employ a *syntactic* approximation of information flow. Specifically, they will all deem the programs in figure 10 insecure since syntactically the assignments to `public` are dependent on `secret`. Semantically these programs are secure since the value of `public` will not change for different values of `secret`. An attacker who can only observe the value of public variables at the end of the program cannot infer information about the value of secret variables. This base-line notion of security, called noninterference[GM82], applies only to non-interactive, terminating programs, covers no kind of declassification and is overly restrictive for concurrent programs. Hence, a wide variety of security notions have been proposed (cf. e.g. [HS11]).

The analyses described here can be enhanced to infer semantic properties and use these to remove spurious information leak warnings. Such techniques may, however, be computationally expensive and can, in principle, not detect all such properties. The KeY[ABB^{+}05] tool allows the user to manually specify and verify arbitrary semantic properties of sequential Java programs and use them to verify information flow security[BBK^{+}12]. This generally requires a considerable amount of manually provided JML[BCC^{+}05] annotations in the program's source code.

6 Conclusion and future work

We have shown how to conduct information flow analyses on Java bytecode programs using the JOANA IFC Console. To specify an analysis goal, only a minimum of user interaction and no knowledge about the structure of the underlying SDG is required. In the future, we will improve on and streamline the IFC Console usability based on user feedback and experience gathered in the RS3 as well as the KASTEL research program. We are going to offer an API that allows tools such as KeY to employ JOANA as backend for IFC queries that can be answered automatically. In order to deal with large programs, we developed a method for modular SDG computation which helps to analyze isolated components in an unknown context. Within the RS3 priority program, we collaborate with the Software Construction and Verification Group at the WWU Münster in order to improve precision for concurrent programs with `synchronized` methods, e.g. using lock-sensitive interference detection with *dynamic pushdown networks*[GHMN13].

Acknowledgments. This work was funded by the DFG under the project in the priority program RS3 (SPP 1496) and by the BMBF under the KASTEL competence center for applied IT security technology.

References

[ABB+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, P. H. Schmitt, et al. The KeY Tool. *Software and System Modeling*, 2005.

[BBK+12] B. Beckert, D. Bruns, R. Küsters, C. Scheben, P. H. Schmitt, and T. Truderung. The KeY Approach for the Cryptographic Verification of Java Programs: A Case Study. Technical Report 2012-8, Department of Informatics, Karlsruhe Institute of Technology, 2012.

[BCC+05] L. Burdy, Y. Cheon, D. Cok, et al. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005.

[Bod12] Eric Bodden. Static flow-sensitive & context-sensitive information-flow analysis for software product lines: position paper. PLAS '12, New York, NY, USA, 2012. ACM.

[CVM07] S. Chong, K. Vikram, and A. Myers. SIF: enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[GHMN13] J. Graf, M. Hecker, M. Mohr, and B. Nordhoff. Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks. 1st International Workshop on Interference and Dependence, 2013.

[Gif11] Dennis Giffhorn. Advanced chopping of sequential and concurrent programs. *Software Quality Journal*, 19(2):239–294, 2011.

[Gif12] Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruher Institut für Technologie, 2012.

[GM82] J. A. Goguen and J. Meseguer. Security Policies and Security Models. *Security and Privacy, IEEE Symposium on*, 1982.

[GPT+11] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. ISSTA '11, New York, NY, USA, 2011. ACM.

[Gra09] J. Graf. Improving and Evaluating the Scalability of Precise System Dependence Graphs for Objectoriented Languages. Technical report, Universität Karlsruhe (TH), 2009.

[Gra10] J. Graf. Speeding up context-, object- and field-sensitive SDG generation. In *9th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.

[GS05] S. Genaim and F. Spoto. Information flow analysis for java bytecode. VMCAI'05. Springer-Verlag, 2005.

[GS12] D. Giffhorn and G. Snelting. Probabilistic Noninterference Based on Program Dependence Graphs. Technical report, Karlsruhe Institute of Technology, 2012.

[Ham10] C. Hammer. Experiences with PDG-based IFC. In *International Symposium on Engineering Secure Software and Systems (ESSoS'10)*. Springer-Verlag, 2010.

[HS09] C. Hammer and G. Snelting. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *IJIS*, 2009.

[HS11] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Proceedings of the 2011 Marktoberdorf Summer School*. IOS Press, 2011.

[JCSH11] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for javascript. PLASTIC '11, New York, NY, USA, 2011. ACM.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, LNCS. Springer, 1996.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, April 2003.

[MSK07] H. Mantel, H. Sudbrock, and T. Kraußer. Combining different proof techniques for verifying information flow security. LOPSTR'06. Springer-Verlag, 2007.

[Mye99] Andrew C. Myers. JFlow: practical mostly-static information flow control. POPL '99. ACM, 1999.

[MZZ+01] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow, July 2001.

[RH07] V. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer*, 2007.

[RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proc. FSE*, SIGSOFT '94, pages 11–20, New York, NY, USA, 1994. ACM.

[RR95] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. *SIGSOFT Softw. Eng. Notes*, 20(4):41–52, October 1995.

[ST07] S. Smith and M. Thober. Improving usability of information flow security in java. PLAS '07. ACM, 2007.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, 2009.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81. IEEE Press, 1981.

[WL10] D. Wasserrab and D. Lohner. Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing. In *VERIFY*, 2010.

[WLS09] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-Based Noninterference and its Modular Proof. In *PLAS*. ACM, June 2009.