Christian Hammer · **Gregor Snelting**

Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs

Abstract Information flow control (IFC) checks whether a program can leak secret data to public ports, or whether critical computations can be influenced from outside. But many IFC analyses are imprecise, as they are flow-insensitive, context-insensitive, or object-insensitive; resulting in false alarms.

We argue that IFC must better exploit modern program analysis technology, and present an approach based on program dependence graphs (PDG). PDGs have been developed over the last 20 years as a standard device to represent information flow in a program, and today can handle realistic programs. In particular, our dependence graph generator for full Java bytecode is used as the basis for an IFC implementation which is more precise and needs less annotations than traditional approaches.

We explain PDGs for sequential and multi-threaded programs, and explain precision gains due to flow-, context-, and object-sensitivity. We then augment PDGs with a lattice of security levels and introduce the flow equations for IFC. We describe algorithms for flow computation in detail and prove their correctness. We then extend flow equations to handle declassification, and prove that our algorithm respects monotonicity of release. Finally, examples demonstrate that our implementation can check realistic sequential programs in full Java bytecode.

Keywords software security · noninterference · program dependence graph · information flow control

This research was supported by Deutsche Forschungsgemeinschaft (DFG grant Sn11/9-2). Preliminary versions of parts of this article appeared in [21] and [20]

Christian Hammer
Universität Karlsruhe (TH), Germany
E-mail: hammer@ipd.info.uni-karlsruhe.de

Gregor Snelting
Universität Karlsruhe (TH), Germany
E-mail: snelting@ipd.info.uni-karlsruhe.de

1 Introduction

Information Flow Control (IFC) is an important technique for discovering security leaks in software. IFC has two main tasks:

- guarantee that confidential data cannot leak to public variables (*confidentiality*);
- guarantee that critical computations cannot be manipulated from outside (*integrity*).

Language-Based IFC analyzes the program source code to discover security leaks. Language-based IFC can exploit a long history of research on program analysis; for example, type systems, abstract interpretation, dataflow analysis, program slicing, etc. A correct language-based IFC will discover any security leaks caused by software alone (but typically does not consider physical side channels). Language-based IFC analysis usually aims to establish *noninterference*, that is by looking at the program text, it tries to prove that the program obeys confidentiality and/or integrity.

In this article, we present a new approach to language-based IFC, which heavily exploits modern program analysis. In particular, it utilizes the *program dependence graph* (PDG), which, after a long history of research, has become a powerful structure capable of handling real programs. Using PDGs results in an IFC which is more precise than previous approaches and thus reduces false alarms. Our implementation can handle full Java bytecode, and we will not only present the theoretical foundations of the method, but preliminary experience as well.

1.1 Principles of Program Analysis

Before we present our method in detail, let us discuss some general properties of IFC and program analysis methods. As any program analysis, language-based IFC is subject to several conflicting requirements:

- *Correctness* is a central property of any IFC analysis: if a potential security leak is present in the source code, the analysis must find it.

- *Precision* means that there are no false alarms: any program condemned by the IFC analysis must indeed contain a security leak.
- *Scalability* demands that the analysis can handle realistic programs (e.g. 100kLOC), written in realistic languages (e.g. full Java bytecode).
- *Practicability* demands that an analysis is easy to use, e.g. does not require many program annotations and delivers understandable descriptions of security leaks.

Unfortunately, due to decidability problems, total precision cannot be achieved while maintaining correctness. As a consequence all correct language-based IFC algorithms are *conservative approximations*: they may generate false alarms, but never miss a potential security leak. And indeed, we can leave occasional false alarms to manual inspection—as long as there are not too many of them. Precision also influences scalability: usually better precision means worse scalability, and fast algorithms are not precise. In the field of program analysis a large collection of techniques has been developed such that the engineer can choose from a spectrum between cheap/imprecise and precise/expensive analysis algorithms; depending on the purpose of the analysis. In particular, the engineer can choose whether an analysis should be

- *flow-sensitive*, that is, order of statements is taken into account, and for every statement a separate analysis information is computed; flow-insensitive analysis computes only one global solution for the program.
- *context-sensitive*, that is, procedure calling context is taken into account, and separate information is computed for different calls of the same procedure; context-insensitive analysis merges all call sites of a procedure.
- *object-sensitive*, that is, different “host” objects for the same field (attribute) of an object are taken into account; object-insensitive analysis merges the information for a field over all objects of the same class.

Decades of research history in program analysis, including a wealth of empirical studies, have shown that all kinds of sensitivity dramatically improve precision in particular for large programs or automatically generated programs. But of course, the more sensitivity, the more expensive an analysis is. Depending on the application, eventually scalability limits the amount of sensitivity one can afford.

Note that some program analysts argue in favor of precise, scalable, but *incorrect* algorithms. If total correctness is not a requirement, precision and scalability can improve drastically. Such algorithms are usually used in testing and bug-finding tools. But while finding (only) 80% of all bugs at low cost can be quite helpful, for IFC we consider such unsound approaches to be inadequate.

Thus sticking to the principle of correctness, we are convinced that modern program analysis can and must be exploited to improve language-based IFC, in particular to improve precision and scalability. But it seems that the IFC community has not yet fully absorbed the power of modern program analysis: in their overview article [49], Sabelfeld and Myers survey contemporary IFC approaches based on program analysis, and find that most approaches are based on *security type systems*. Some authors proposed to use abstract interpretation or model checking, but other very successful approaches to program analysis, such as precise interprocedural dataflow analysis, points-to analysis, or program slicing seem not yet popular for IFC. Let us thus review fundamental properties of security type systems.

```
if (confidential==1)
  public = 42
else
  public = 17;
... // the following statement can be
... // far away from the IF;
... // as long as it postdominates the IF
... // and there is no intermediate
... // output of public,
... // the program is secure
public = 0;
```

Fig. 1 A secure program fragment

1.2 Security Type Systems

Security type systems attach security levels—coded as types—to variables, fields, expressions, etc. and the typing rules propagate security levels through the expressions and statements of a program, guaranteeing to catch illegal flow of information [53]. Thus such type systems are correct. Type systems can handle sequential as well as concurrent programs, and can even discover timing leaks [2]. Type-based IFC is efficient, correctness proofs are not too difficult, and realistic implementations, e.g. the JIF system [37], exist. So security type systems are a success story and can be seen as a “door-opener” for the whole field of language-based IFC.

But type systems can be rather imprecise, as most security type systems are not flow-sensitive, context-sensitive, nor object-sensitive. This leads to false alarms. For example, the well-known program fragment in Fig. 1 is considered insecure by type-based IFC, as type-based IFC is not flow-sensitive. It does not see that the potential illegal flow from `confidential` to `public` in the if-statement (a so-called *implicit flow*) is guaranteed to be *killed*[1] by the later assignment, and thus declares the fragment to be untypeable. But by classical definitions of noninterference (see below) the program is perfectly secure. Note that the killing assignment may be far away from the IF (e.g. in a different method); but as long as it postdominates the IF, the program is secure anyway. Hence the false positive cannot be eliminated by simple program transformations. On the contrary, empirical studies have validated that flow-sensitivity in program analysis strongly improves precision (see, e.g. [24]).

[1] an established term in dataflow analysis

Type-based IFC performs even worse in the presence of unstructured control flow or exceptions. Therefore, type systems overapproximate IFC, resulting in too many secure programs rejected (false positives). First steps towards flow-sensitive type systems have been proposed, but are restricted to rudimentary languages like While-languages [27], or languages with no support for unstructured control flow [3].

1.3 PDGs and Overview

Fortunately, program analysis has much more to offer than just sophisticated type systems. In particular, the PDG [15] has become, after 20 years of research, a standard data structure allowing various kinds of powerful program analyses—in particular, efficient program slicing. Today, commercial PDG tools for full C are available [4], which have been used in a large number of real applications, and there are at least two PDG implementations for full Java [23,28]. One of them is used as the basis for a new IFC algorithm, as described in this article.

The first IFC algorithm based on PDGs was presented by Snelting in 1996 [54]. But more elaborate algorithms were needed to make the approach work and scale for full C and realistic programs [47,55]. The latter article contains a theorem connecting dependence graphs to the classical noninterference criterion (see also Sect. 2). Later, we developed a precise dependence graph for full Java bytecode [23], which is much more difficult than for C due to the effects of inheritance and dynamic dispatch, and due to the concurrency caused by thread programming. Krinke's slicing algorithm [31] for multi-threaded programs has recently been integrated. Today, we can handle realistic C and Java programs and thus have a powerful tool for IFC available that is more precise than conventional approaches. In particular, it handles Java's exceptions and unstructured control flow precisely.

In this article, we first recapitulate foundations of PDGs, and explain flow-, context-, and object-sensitivity. We then augment PDGs with Denning-style security level lattices and explain the equations that propagate security levels through the program in details. We focus on precise interprocedural analysis with declassification in called methods, and present new algorithms and correctness proofs which were not part of our earlier work [20,21]. Finally, we present several examples and performance data, and discuss future work.

2 Dependence Graphs and Noninterference

2.1 PDG Basics

Program dependence graphs are a standard tool to model information flow through a program. Program statements or expressions are the graph nodes. There are two kinds of edges: data dependences and control dependences. A data dependence edge $x \to y$ means that statement x assigns a variable which is used in statement y, without being reassigned ("killed") underway. A control dependence edge $x \to y$ means that the mere execution of y depends directly on the value of the expression x (which is typically a condition in an if- or while-statement). Control and data dependences are transitive.

```
1 a = u();
2 while (f()) {
3   x = v();
4   if (x>0)
5     b = a;
6   else
7     c = b;
8 }
9 z = c;
```

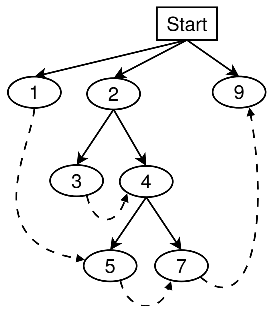

Fig. 2 A small program and its dependence graph

In a PDG $G = (N, \to)$, a path $x \to^* y$ means that information can flow from x to y; if there is no path, it is guaranteed that there is no information flow [25,44,46,57]. Thus PDGs are *correct*. Exploiting this fundamental property, all statements possibly influencing y (the so-called *backward slice*) are easily computed as

$$BS(y) = \{x \mid x \to^* y\} \tag{1}$$

where y is called the *slicing criterion* of the backward slice. In particular, the Slicing Theorem [46] shows that for any initial state on which the program terminates, the program and its slice compute the same sequence of values for each element of the slice.

As an example, consider the small program and its dependence graph in Fig. 2. Control dependences are shown as solid edges, data dependences are shown as dashed edges; transitive dependences are not explicitly part of the PDG. There is a path from statement 1 to statement 9, indicating that input variable a may eventually influence output variable z. Since there is no path $(1) \to^* (4)$, there is definitely no influence from a to x. In Fig. 1, the PDG contains edges $1 \to 2$ and $1 \to 4$, but *not* $(1) \to^* (11)$ and thus is considered safe as `public` does not depend on `confidential`; no false alarm is generated.

The power of PDGs stems from the fact that they are *flow-sensitive* and *context-sensitive*: the order of statements does matter and is taken into account, as is the actual calling context for procedures. As a result, the PDG never indicates influences which are in fact impossible due to the given statement execution order of the program; only so-called "realizable" (that is, dynamically possible) paths are considered. But this precision is not for free: PDG construction can have complexity $O(|N|^3)$ due to the transitive summary edges (see Sect. 3). In practice, PDG use is limited to programs of about 100kLOC [10].

Even the most precise PDG is still a conservative approximation: it may contain too many edges (but never too few). PDGs and slicing for realistic languages with procedures, complex control flow, and data structures are much more complex than the fundamental concept sketched above. For the full C or Java language, the computation of precise dependence graphs and slices is absolutely nontrivial; there is ongoing research worldwide since many years. In-depth descriptions of slicing techniques can be found in [32]; some techniques are explained in Sect. 3.

2.2 Noninterference and PDGs

As explained, a missing path from a to b in a PDG guarantees that there is no information flow from a to b. This is true for all information flow which is not caused by hidden physical side channels such as timing leaks. It is therefore not surprising that traditional technical definitions for secure information flow such as *noninterference* are related to PDGs.

Noninterference was originally introduced by Goguen and Meseguer in [18]. Their notion of security is based on observational behavior and not on the source code. Today, noninterference is usually defined with respect to the program text. Its simplest variant as introduced by Cohen [14]—which assumes only security levels *Low* and *High*—is defined as

$$s \cong_{Low} s' \implies [\![c]\!]s \cong_{Low} [\![c]\!]s' \qquad (2)$$

where c is a statement or program, s, s' are two initial program states, and $[\![c]\!]s$, $[\![c]\!]s'$ are the corresponding final states after executing c. $s \cong_{Low} s'$ means that s and s' are *Low equivalent*: they must coincide on variables which have *Low* security, but not on variables with *High* security. Thus variation in the high input variables does not affect low output, and hence confidentiality is assured. Note that various extensions of elementary noninterference have been defined, such as possibilistic or probabilistic noninterference; some of them are based on PER relations [50].

In earlier work, we established a connection between PDGs and Goguen/Meseguer noninterference, and proved the following theorem.

Theorem 1 *Let a, c be two PDG nodes. $dom(c) \rightsquigarrow dom(a)$ states that c may influence (interfere with) a. If*

$$c \in BS(a) \implies dom(c) \rightsquigarrow dom(a) \qquad (3)$$

then the Goguen/Meseguer noninterference criterion is satisfied for a.
Proof. *For details of notation and proof, see [55].* □

The theorem demonstrates how PDGs can be used to check for noninterference: if c and a have noninterfering security levels, there must be *no PDG path* $c \rightarrow^* a$, otherwise a security leak has been discovered.

This result was recently transferred to low-deterministic security, formalized in the machine prover Isabelle [40], and proven for the Jinja formalization of Java bytecode [30].

Theorem 2 *Let c be a program; let s_1, s_2 be two initial states and s'_1, s'_2 the corresponding final states after executing c, where c' is the final (i.e. empty) residual program after executing c:*

$$\langle c, s_1 \rangle \Rightarrow \langle c', s'_1 \rangle \wedge \langle c, s_2 \rangle \Rightarrow \langle c', s'_2 \rangle \wedge final\ c'$$

If no High *variable is in the backward slice of* Low *output, then low-deterministic security is satisfied:*

$$s_1 \cong_{low} s_2 \wedge High \notin BS\ Low \implies s'_1 \cong_{low} s'_2$$

The theorem was proven for a realistic subset of bytecode (without threads), but—due to the modular structure of the proof as described in Wasserrab et al. [57]—is much more general and in principle applies to any imperative language. The generality of the theorem stems from the fact that it is independent of specific languages or slicing algorithms; it just exploits a fundamental property of any correct slice. The theorem is valid even for imprecise PDGs and slices, as long as they are correct. Applying the theorem results in a linear-time noninterference test for a, as all $s \in BS(a)$ must be traversed once. More precise slices result in less false alarms.

However, as we will see later, it is not possible to use declassification in a purely slicing based approach, thus later we will present extended versions of Theorem 1 resp. 2.

2.3 Beyond PDGs

Ongoing research is making PDGs and slicing more precise every year. But PDG precision can also be improved by non-PDG means, as developed in program analysis.

As an example, consider the fragment
“`if (h > h) then l = 0`”
Naive slicing as well as security type systems will assume a transitive dependence from `h` to `l`, even though the `if` body is dead code. Thus, semantic consistency as postulated in Sabelfeld and Sands [51] is violated: a semantics-preserving transformation such as dead code elimination changes the confidentiality status. This is not in discrepancy with Theorem 2, but comes from analysis imprecision.

Fortunately, PDGs today come in a package with other analyses originally developed for code optimization, such as interprocedural constant propagation, static single assignment form, symbolic evaluation, and dead code elimination. These powerful analyses are performed before PDG construction starts, and will eliminate a lot of spurious flow. The easiest way to exploit such analyses is by constructing the PDG from bytecode or intermediate code. For the above example, any optimizing compiler will delete the whole statement from machine code or bytecode, as it is dead code. Note that the bytecode must be considered the ultimate definition of the program’s meaning, and remaining flows in the bytecode—after all the sophisticated optimizations—must be taken all the more seriously.

```
1 secret = 1;
2 public = 2;
3 s = f(secret);
4 x = f(public);
5 p = x;
6
7 int f(int x) { return x+42; }
```

Fig. 3 Example for context-sensitivity

In addition, we proposed an even stronger mechanism on top of PDGs, called *path conditions* [22,47,54,55]. Path conditions are necessary and precise conditions for flow $x \rightarrow^* y$, and reveal detailed circumstances of a flow in terms of conditions on program variables. If a constraint solver can solve a path condition for the program's input variables, feeding such a solution to the program makes the illegal flow visible directly; this useful feature is called a *witness*. As an example, consider the fragment

```
1 a[i+3] = x;
2 if (i>10 && j<5)
3     y = a[2*j-42];
```

Here, a necessary condition for a flow from line 1 to line 3 is $\exists i, j.(i > 10) \wedge (j < 5) \wedge (i+3 = 2j-42) \equiv \mathit{false}$, proving that flow is impossible even though the PDG indicates otherwise. Note that path conditions are not described in this article; for the quite complex details on generation, correctness, precision, and scalability see [55].

3 Dependence Graphs for Java

In the following, we assume some familiarity with slicing technology, as presented for example in [32]. Note that the computation of precise dependence graphs and slices for object-oriented languages is still an ongoing research topic.

Our Java PDG is based on bytecode rather than source text for the following reasons:

- Bytecode must be considered the ultimate definition of a program's meaning and potential flows.
- The bytecode is much more stable than the source language (see, e.g. generics in Java 5, which did not change the bytecode instructions).
- The bytecode is already optimized, and artifacts such as dead code are removed and cannot generate spurious flow (see the example in the last section).

3.1 Methods and Dynamic Dispatch

From bytecode, intraprocedural PDGs can easily be constructed for method bodies, using well-known algorithms from literature. Concerning procedures, standard interprocedural slicing relies on so-called *system dependence graphs* (SDGs), which include dependences for calls as well as transitive dependences between parameters [26].[2] We will show example SDGs later, but would like to point out right now that SDG-based slicing is *context-sensitive*. That is, different calls to the same procedure or method are indeed distinguished. Context-sensitivity increases precision, as can be seen in Fig. 3. We assume that `f` does not have side-effects and that the input parameter influences the result value. Context-sensitive analysis will distinguish the calling contexts for the two `f` calls, and generate dependences $1 \rightarrow 3$ and $2 \rightarrow 4 \rightarrow 5$ but not $1 \rightarrow 4 \rightarrow 5$. Context-insensitive analysis merges the calls and generates dependences $1 \rightarrow 3, 1 \rightarrow 4, 2 \rightarrow 3, 4 \rightarrow 5$. Thus only context-sensitive analysis discovers that `p` is *not* influenced by `secret`.

Dynamic dispatch and objects as method parameters render SDG construction more difficult. In particular, in case a Java program creates objects, any precise program analysis such as SDG construction must run a *points-to analysis* first. Points-to analysis determines for every object reference a set of objects it might (transitively) point to; this set is called a points-to set. A number of correct points-to algorithms with varying precision and scalability are available, e.g. [33,48].

Having computed the points-to sets, treatment of dynamic dispatch is well known: possible targets of method calls are approximated statically via the points-to sets, and for all possible target methods the standard interprocedural SDG construction is done.

Method parameters are a more difficult issue. SDGs support call-by-value-result parameters, and use one SDG node for each in- and out-parameter. Java supports only call-by-value; in particular, for reference types the content of the formal variable holding the reference to the passed object is not copied back on return. But field values stored in actual parameter objects may change during a method call. Such possible field changes have to be made visible in the SDG by adding modified fields to the formal-out parameters; furthermore, static variables are represented as extra parameters.

To improve precision, we made the SDG *object-sensitive* by representing nested parameter objects as trees. Unfolding object trees stops once a fixed point with respect to the points-to situation of the containing object is reached [23]. It is only at this point during SDG construction that object-sensitivity enters the scene—the rest of the IFC analysis algorithms (and the paper) remain uninfluenced. The algorithms as explained below require only SDG correctness and transform SDG precision into IFC precision; in particular an object-sensitive SDG automatically makes the whole IFC object-sensitive.

3.2 Exceptions

Figure 4 shows a fragment of a Java class for checking a password (taken from [37]) which uses fields and excep-

[2] In the following, we will refer to SDGs when we are talking about the complete system and we will refer to PDGs in the sense of procedure dependence graphs: they represent that part of a SDG that corresponds to a single procedure or method.

```
 1
 2 class PasswordFile {
 3  private String[] names;
 4                 /*P: confidential*/
 5  private String[] passwords;
 6                 /*P: secret*/
 7  // Pre:all strings are interned
 8  public boolean check(String user,
 9    String password /*P: confidential*/) {
10    boolean match = false;
11    try {
12    for (int i=0; i<names.length; i++) {
13     if (names[i]==user
14       && passwords[i]==password) {
15       match = true;
16       break;
17     }
18    }
19   }
20   catch (NullPointerException e) {}
21   catch (IndexOutOfBoundsException e) {};
22   return match;  /*R: public*/
23   }
24 }
```

Fig. 4 A Java password checker

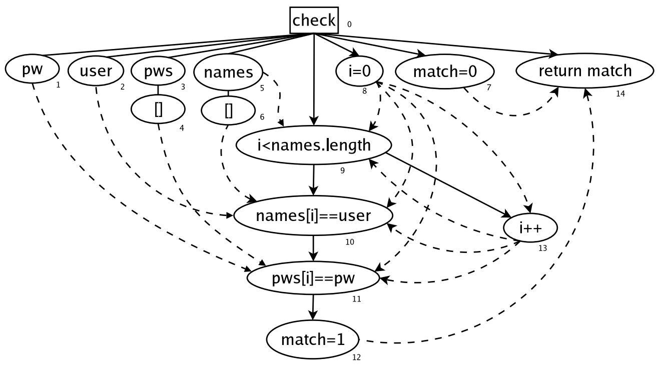

Fig. 5 PDG for check in Fig. 4

tions. The *P* and *R* annotations will be explained in Sect. 4. The initial PDG for the check method is shown in Fig. 5. Node 0 is the method entry with its parameters in nodes 1 and 2 (we use "pw" and "pws" as a shorthand for "password" and "passwords"). Nodes 3–6 represent the fields of the class, note that because the fields are arrays, the reference and the elements are distinguished.[3] Nodes 7 and 8 represent the initializations of the local variables match and i in lines (10) and (12). All these nodes are immediately control dependent on the method entry. The other nodes represent the statements (nodes 12, 13, and 14) and the predicates (nodes 9, 10, and 11).

This PDG is still incomplete, as it does not include exceptions. Dynamic runtime exceptions can alter the control flow of a program and thus may lead to implicit flow, in case the exception is caught by some handler on the call-stack, or else represent a covert channel in case the exception is propagated to the top of the stack yielding a program termination with stack trace. This is why many type-based approaches disallow (or even ignore) implicit exceptions. Our analysis conservatively adds control flow edges from bytecode instructions which might throw unchecked exceptions to an appropriate exception handler (as proposed by Chambers et al. [11]), or percolates the exception to the callee which in turn receives such a conservative control flow edge. Thus, our analysis does not miss implicit flow caused by these exceptions, hence even the covert channel of uncaught exceptions is checked.[4] The resulting final PDG is shown in Fig. 6. (For better readability, the following examples will not show the effects of exceptions.)

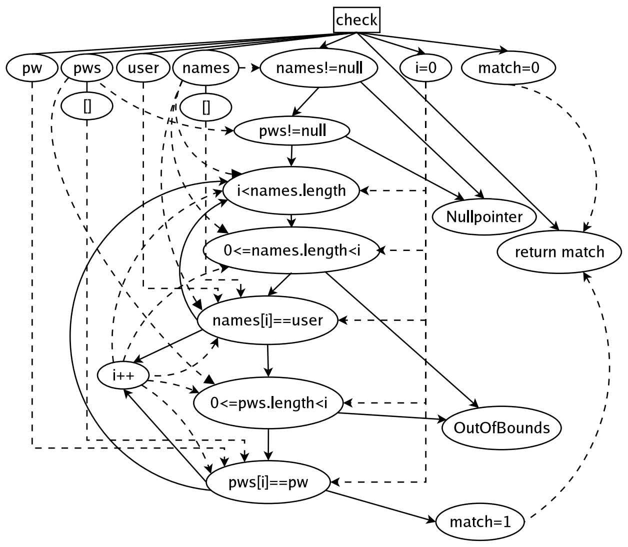

Fig. 6 PDG with exceptions for Fig. 4

3.3 Context-Sensitivity and Object-Sensitivity in Action

Figure 7 shows another small example program, and Fig. 8 shows its SDG. For brevity we omitted the PDGs of the set and get methods.[5] The effects of method calls are reflected by *summary edges* (shown as dashed edges in Fig. 7) between actual-in and actual-out parameter nodes. Summary edges as introduced by Horwitz et al. [26] represent a transitive dependence between the corresponding formal-in and formal-out node pair for both standard variables and dependences induced by heap access. They represent all dependences visible outside the called method which arise from normal termination as well as when an exception terminates that method. For example, the call to o.set(sec) contains two summary edges, one from the target object o and one from sec to the field x of o; representing the side-effect that the value of sec is written to the field x of the this-pointer

[3] Precise PDGs for arrays are nontrivial, but are not described here.

[4] Note that our analysis ignores Errors such as out of memory error, as—in contrast to Exceptions—they should not be handled by the program.

[5] Note that when the direction of control dependences is clearly downwards, we omitted the arrow heads.

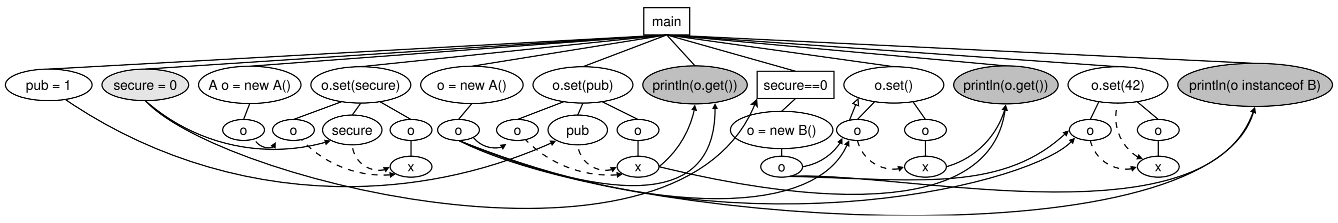

Fig. 8 SDG for the program in Fig. 7

```
1  class A {
2  int x;
3  void set() { x = 0; }
4  void set(int i) { x = i;}
5  int get() { return x; }
6  }
7  class B extends A {
8  void set() { x = 1; }
9  }
10 class InfFlow {
11 void main(String[] a){
12   //1. no information flow
13   int sec = 0 /*P:High*/;
14   int pub = 1 /*P:Low*/;
15   A o = new A();
16   o.set(sec);
17   o = new A();
18   o.set(pub);
19   System.out.println(o.get());
20   //2. dynamic dispatch
21   if (sec==0 && a[0].equals("007"))
22     o = new B();
23   o.set();
24   System.out.println(o.get());
25   //3. instanceof
26   o.set(42);
27   System.out.println(o instanceof B);
28 }
29 }
```

Fig. 7 Another Java program

in set. Summary edges enable context-sensitive slicing in SDGs in time linear to the number of nodes [26].

In the program, the variable sec is assumed to contain a secret value, which must not influence printed output. First a new A object is created where field x is initialized to sec. However, this object is no longer used afterward as the variable is overwritten ("killed") with a new object whose x field is set to pub. Now context-sensitive analysis of set discovers that x of the new object is *not* influenced by sec. Thus there is no SDG path $(13) \rightarrow^* (19)$ from the initialization of sec to the first print statement (i.e. the leftmost println node). Instead, we have a path from the initialization of pub to this output node. Hence the sec variable does not influence the output. This example demonstrates that the x fields in the two A objects are distinguished (object-sensitivity), the side-effects of different calls to set are not merged (context-sensitivity), and flow-sensitivity kills any influence from the sec variable to the first println.

The next statements show an illegal flow of information: Line (21) checks whether sec is zero and creates an object of class B in this case. The invocation of o.set is dynamically dispatched: If the target object is an instance of A then x is set to zero; if it has type B, x receives the value one. Lines (21)–(23) are analogous to the following implicit flow:

```
if (sec==0 && ...) o.x = 0 else o.x = 1;
```

In the PDG we have a path from sec to the predicate testing sec to o.set() and its target object o. Following the summary edge one reaches the x field and finally the second output node. Thus the PDG discovers that the printed value in line 24 depends on the value of sec. In the next chapter, we will formally introduce security levels and demonstrate that this example contains an illegal flow.

But even if the value of x was not dependent on sec (after statement 26) an attacker could exploit the runtime type of o to gain information about the value of sec in line 27. This implicit information flow is detected by our analysis as well, since there is a PDG path $(13) \rightarrow^* (27)$.

3.4 Concurrency

Let us finally say a few words about PDGs for multi-threaded programs which are common in Java. Krinke [31] was the first author to present a precise algorithm for slicing multi-threaded programs. Based on special data dependences between variables in different threads, Krinke's algorithm ensures that the sequence of statements in a PDG path corresponds to a possible ("realizable") execution order. Paths which contain impossible execution orders and thus would introduce "time-traveling" (which can only happen when slicing concurrent programs) are filtered out. We integrated this algorithm and improved it with ideas from Nanda and Ramesh [39]. Our implementation can in principle handle any number of threads. But note that precise slicing of multi-threaded programs is very expensive (exponential in the number of threads), and experiments indicate that prevention of time-traveling should only be applied as a post-processing step [17]. An integration with the IFC algorithm as described later has not yet been completed.

4 Security levels

The noninterference criterion prevents illegal flow, but in practice one wants more detailed information about security levels of individual statements. Thus theoretical models for IFC such as Bell–LaPadula [6] or Noninterference [18] utilize a *lattice* $\mathscr{L} = (L; \leq, \sqcup, \sqcap, \bot, \top)$ of security levels, the simplest consisting just of two security levels $High \geq Low$.[6] The programmer needs to specify a lattice, as well as annotations defining the security level for some (or all) statements. In practice, only input and output channels resp. statements or calls need such annotations.[7]

Arguing about security also requires an explicit attacker model. For our approach, we assume:

- Attackers cannot control the execution of the JVM including its security settings.
- The code generated from source (e.g. bytecode) is known to the attacker (maybe through disassembling), but cannot be altered (e.g. due to code signing).
- Therefore, the content of variables (local as well as in the heap) is not directly available to the attacker. Such an assumption would allow to learn all secrets as soon as they are stored.
- As a consequence, only input with a certain security level (e.g. assigned by the OS) can be controlled, and only output with a certain level be observed.

4.1 Fundamental Flow equations

For a correct IFC, the actual security level of every statement must be computed, and this computation must respect the programmer-specified levels as well as propagation rules along program constructs. The huge advantage of PDG-based IFC is that the PDG already defines the edges between statements or expressions, where a flow can happen; as explained, explicit and implicit flow between unconnected PDG nodes is impossible. Thus it suffices to provide propagation rules along PDG edges. We begin with the intraprocedural case.

The security level of a statement with PDG node x is written $S(x)$, where $S : N \to L$.[8] Confidentiality requires that an information receiver x must have at least the security level of any sender y [6]. In a PDG G, where *pred* and *succ* are the predecessor and successor functions induced by $\to$, this fundamental property is easily stated as

$$y \to x \in G \implies S(x) \geq S(y) \tag{4}$$

and thus by the definition of a supremum

$$S(x) \geq \bigsqcup_{y \in pred(x)} S(y) \tag{5}$$

[6] Note that by tradition the more “secret” levels correspond to the “upper” lattice elements.

[7] Note that in Java input/output is done by library functions. Eventually, basic I/O is implemented by native code; for native methods, the PDGs must be supplied manually or generated from so-called *stubs*.

[8] Remember that N is the set of PDG nodes. Note that our S is called *dom* in the original Goguen/Meseguer noninterference definition, but we need *dom* for partial functions.

This fundamental constraint ensures $S(y) \rightsquigarrow S(x)$ and thus confidentiality.[9] Let us point out once more that it is sufficient to consider flow along PDG edges, as flow between unconnected PDG nodes is impossible.

Before we proceed with more detailed flow equations for confidentiality, let us shortly discuss integrity. Remember that confidentiality and integrity are dual to each other [8]: Confidentiality demands that secret data cannot be leaked to public output, and integrity demands that public data cannot influence secret computations (“no read down, no write up”). In terms of flow equations, this implies that all security constraints change their direction in the lattice: $\leq$ becomes $\geq$, $\sqcap$ becomes $\sqcup$. Technically, one switches to $\mathscr{L}$’s dual lattice. Hence the dual version of condition (5) for integrity is

$$S(x) \leq \bigsqcap_{y \in pred(x)} S(y) \tag{6}$$

In the following, we concentrate on confidentiality, as all equations for integrity are obtained by duality.

Equation (5) assumes that every statement (more precisely PDG node) has a security level specified, which is not realistic. In practical applications, one wants to specify security levels not for all statements, but for certain selected statements only; for practicability of an analysis, it is important that the number of such annotations is as small as possible. Indeed it is sufficient to annotate input (and similar statements) with a so-called provided security level, and to annotate output (and similar statements) with so-called required security levels. The *provided* security level specifies that a statement sends information with the provided security level (or higher). The *required* security level requires that only information with a *smaller* or equal security level may reach that statement.[10] From these values the *actual* security levels can be computed.

Provided security levels are defined by a partial function $P : N \nrightarrow L$. The required security levels are defined similarly as a partial function $R : N \nrightarrow L$. Thus, $P(s)$ specifies the security level of the information generated at s (also called “the security level of s”), and $R(s)$ specifies the maximal allowed security level of the information reaching s. Note that usually $dom(P) \cap dom(R) = \varnothing$, as in our fine-grained PDG input and output nodes are well distinguished—but technically, this is not required.

The actual security level $S(x)$ for a node x must thus not only be greater than or equal to the levels of its predecessors, but also greater than or equal to its own provided security level. Thus Eq. (5) refines to

$$S(x) \geq \begin{cases} P(x) \sqcup \bigsqcup_{y \in pred(x)} S(y), & \text{if } x \in dom(P) \\ \bigsqcup_{y \in pred(x)} S(y), & \text{otherwise} \end{cases} \tag{7}$$

[9] In fact, the Goguen/Meseguer notion $S(y) \rightsquigarrow S(x)$ is the same as $S(y) \leq S(x)$ in modern terminology.

[10] The term “required” may be misleading here—it is actually more like a limit or maximum

Note that R does not occur in this constraint for S. We need an additional constraint to specify that incoming levels must not exceed a node's required level:

$$\forall x \in \text{dom}(R) : R(x) \geq S(x) \tag{8}$$

We can now formally define confidentiality:

Definition 1 Let a program's PDG $G = (N, \rightarrow)$ be given, together with a security lattice $\mathscr{L} = (L; \leq, \sqcup, \sqcap, \top, \bot)$, as well as a specification of provided security labels $P : N \nrightarrow L$ and required security labels $R : N \nrightarrow L$.

The program *maintains intraprocedural confidentiality*, if for all PDG nodes Eqs. (7) and (8) are satisfied.

We are preparing a machine-checked proof that Definition 1 implies noninterference as defined in Eq. (2). This proof will be technically more difficult than the—already completed—machine-checked proof for Theorem 2, as the details of Eqs. (7) and (8) must be integrated into the formalization. For the time being, Definition 1 is treated as an axiom, which however, as discussed above, is well-founded in correctness properties of PDGs and classical definitions of confidentiality.

Later, we will provide an interprocedural generalization of this definition (Definition 2 in Sect. 5), which additionally exploits the fact that it is sufficient to consider the backward slices of all output ports instead of the whole PDG; this observation again reduces spurious flow and the risk for false alarms. For the time being, we demand (7) and (8) for the whole PDG, which is still a correct (if slightly less precise) definition.

For simplicity in presentation, we extend P and R to total functions P' and R' such that all nodes have a provided and required security level:

$$P'(x) = \begin{cases} P(x), & \text{if } x \in dom(P) \\ \bot, & \text{otherwise} \end{cases} \tag{9}$$

$$R'(x) = \begin{cases} R(x), & \text{if } x \in dom(R) \\ \top, & \text{otherwise} \end{cases} \tag{10}$$

Note that $\bot$ is the neutral element for $\sqcup$, and $\top$ is the neutral element for $\sqcap$. Now Eq. (7) simplifies to

$$S(x) \geq P'(x) \sqcup \bigsqcup_{y \in pred(x)} S(y) \tag{11}$$

and Eq. (8) simplifies to

$$R'(x) \geq S(x) \tag{12}$$

4.2 Solving Flow equations

Equation (11) is satisfied in the most precise way, and hence the risk that Eq. (8) is violated is minimized, if the inequality for S turns into equality:

$$S(x) = P'(x) \sqcup \bigsqcup_{y \in pred(x)} S(y) \tag{13}$$

Of course (13) also satisfies (11), and can be read as an algorithm which computes $S(x)$ from $P(x)$ and x's predecessors S values. Thus Eq. (13) defines a *forward propagation*: it shows what happens if all the P values are propagated through the PDG (while ignoring R).

We will now show that Eq. (13) corresponds to a well-known concept in program analysis, namely a *monotone dataflow analysis framework* [29], which allows efficient fixpoint computation. Such frameworks start with a lattice of abstract values, which in our case is $\mathscr{L}$. For every $x \in N$, a so-called *transfer function* $f_x : L \to L$ must be defined, which typically has the form $f_x(l) = g_x \sqcup (l \sqcap \overline{k_x})$.[11] In our case, $g_x = P'(x)$ and $k_x = \bot$, thus $f_x(l) = P'(x) \sqcup l$. Furthermore, for every $x \in N$, the framework defines $out(x) = f_x(in(x))$ and $in(x) = \bigsqcup_{y \in pred(x)} out(y)$. In our case,

$$out(x) = f_x(in(x)) = P'(x) \sqcup \bigsqcup_{y \in pred(x)} S(y) = S(x)$$

The theory demands that all f_x are monotone, which in our case is trivial. The theory also states that if the f_x are distributive, the analysis is more precise. In our case $f_x(l_1 \sqcup l_2) = P'(x) \sqcup (l_1 \sqcup l_2) = (P'(x) \sqcup l_1) \sqcup (P'(x) \sqcup l_2) = f_x(l_1) \sqcup f_x(l_2)$, hence distributivity holds. The theory finally states that the set of equations for S (resp. *out*) always has a solution in form of a minimal fixpoint, that this solution is correct, and in case of distributive transfer functions it is precise. This is another reason why our IFC is more precise than other approaches.[12] Efficient algorithms to compute this fixed point are well known. We will show examples for fixpoints later; here it suffices to say that it defines values for $S(x) \in L$ which simultaneously satisfy Eqs. (13) and thus (11) and (7) for all $x \in N$.

Thus the computed fixpoint for S, together with Eq. (8), ensures confidentiality. If a fixpoint for S exists, but the condition for R cannot be satisfied, then a confidentiality violation has been discovered: For any $l = R(x)$ such that $l \ngeq S(x)$ we have a violation at x because $S(x) \nrightarrow l$ (the security level of $S(x)$ is not allowed to influence level l). Note that it is $\ngeq$ and not $<$ because l and $S(x)$ might not be comparable.

From a program analysis viewpoint, our transfer functions f_x are quite simple; in fact they are so simple that an

[11] where $g_x, k_x \in L$. $\overline{k_x}$ denotes Boolean complement, as many dataflow methods run over a *powerset* $\mathscr{L}$; in our case we have just a lattice but all we need is $\overline{\bot} = \top$. Thus the f_x have a typical "gen/kill" form.

[12] Note that we thus have total precision for the S solutions, but not for the underlying PDG.

explicit solution for the fixpoint can be given which will be exploited later:

Theorem 3 *Let a program with PDG $G = (N, \rightarrow)$ be given. For all $x \in N$, let $S(x)$ be the least fixpoint of Eq. (13). Let $BS(x)$ be the (intraprocedural) backward slice according to Eq. 1. Then*

$$S(x) = \bigsqcup_{y \in BS(x)} P'(y) \tag{14}$$

Proof. Let $x \in N$. (13) implies $S(x) \geq P'(x)$ and $S(x) \geq S(y)$ for all $y \in pred(x)$. By induction, this implies for any path $y \rightarrow^* x$ (i.e. $y \in BS(x)$): $P'(y) \leq S(x)$. By definition of a supremum, $S(x) \geq \bigsqcup_{y \in BS(x)} P'(y)$.

On the other hand, (14) is a solution of (13):

$$\begin{aligned}\bigsqcup_{y \in BS(x)} P'(y) &= P'(x) \sqcup \bigsqcup_{\substack{y \in pred(x)\\ z \in BS(y)}} P'(z)\\ &= P'(x) \sqcup \bigsqcup_{y \in pred(x)} \bigsqcup_{z \in BS(y)} P'(z)\end{aligned}$$

and since S is the least fixpoint we have $S(x) \leq \bigsqcup_{y \in BS(x)} P'(y)$. Thus equality, as stated in the theorem, follows. □

4.3 The PDG-Based Noninterference Test

We will now exploit this intermediate result to prove the correctness of our PDG-based confidentiality check. The following statement is a restatement of Theorem 1 in terms of P and R:

Theorem 4 *If*

$$\forall a \in \mathrm{dom}(R) : \forall x \in BS(a) \cap \mathrm{dom}(P) : P(x) \leq R(a) \tag{15}$$

then confidentiality is maintained for all $x \in N$.

That is, the backward slice from a node a with a required security level $R(a)$ must not contain a node x that has a higher provided security level $P(x)$.

Proof. Let $x \in N$. We need to show that (7) and (8) are valid for x. From the premise we know $\forall x \in BS(a) : P'(x) \leq R(a)$, as $P'(x) \leq P(x)$ if $x \in dom(P)$. Thus $R(a) \geq \bigsqcup_{x \in BS(a)} P'(x)$, hence $R(a) \geq S(x)$ by Theorem 3. Hence (8) is satisfied. Furthermore, by definition of the fixpoint for S, S satisfies (13) and thus (11) and (7). □

The theorem can easily be transformed into an algorithm that checks a program for confidentiality:

PDG-Based Confidentiality Check. For every $a \in dom(R)$, check that for all $x \in dom(P) \cap BS(a) : P(x) \leq R(a)$.

Once the PDG has been computed, each backward slice and thus confidentiality check has worst case complexity $O(|N|)$. Usually, the number of nodes that have a specified security level $R(a)$ is bounded and not related to $|N|$; typically just a few output statements have $R(a)$ defined. Thus overall complexity can be expected to be $O(|N|)$ as well.

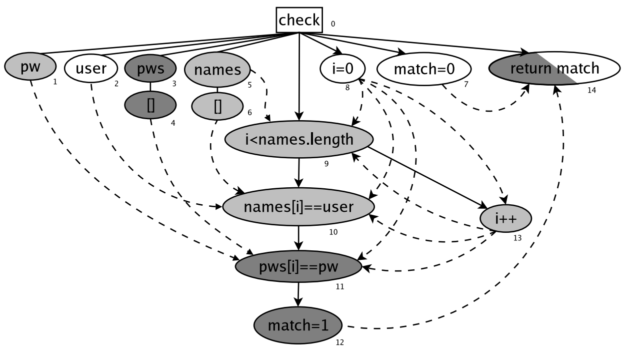

Fig. 9 PDG for Fig. 4 with computed security levels

Checking each node separately allows a simple yet powerful diagnosis in the case of a security violation: If a node x in the backward slice $BS(a)$ has a provided security level that is too large or incomparable ($P(x) \not\leq R(a)$), the responsible nodes can be computed by a so-called *chop* $CH(x,a) = FS(x) \cap BS(a)$.[13] The chop computes all nodes that are on a path from x to a, thus it contains all nodes that may be involved in the propagation from x's security level to a.

As an example, consider again the PDG for the password program (Fig. 5). We choose a three-level security lattice: *public*, *confidential*, and *secret* where

$$public \leq confidential \leq secret \tag{16}$$

The program contains P-annotations for input variables, and an R-annotation for the result value. Thus the list of passwords is *secret*, i.e. $P(3) = secret \wedge P(4) = secret$. The list of names and the parameter `password` is *confidential*, because they should never be visible to a user. Thus, $P(1) = confidential \;\wedge\; P(5) = confidential \;\wedge\; P(6) = confidential$.

No *confidential* or *secret* information must flow out of `check`, thus we require $R(14) = public$. Remember that the PDG has additional dependences for exceptions (see Fig. 6). In order to prevent an implicit flow from `check` to the calling method via uncaught exceptions, the node of the calling method representing any uncaught exception, m, is annotated with $R(m) = public$. Thus an implicit flow via an uncaught exception, where the exception is dependent on a *secret* variable, will be detected at m.

Starting with these specifications for R and P, the actual security levels $S(x)$, as computed according to Eq. (13), are depicted in Fig. 9 (white for *public*, light gray for *confidential*, and gray for *secret*.[14]) Let us now apply the PDG-based confidentiality check. It turns out that $3 \in BS(14)$ where $R(14) = public, P(3) = secret$. Thus the criterion fails. Indeed a security violation is revealed: $S(14) = secret \not\leq public = R(14)$, thus Eq. (8) is violated. The chop $CH(3,14)$ contains all nodes contributing to the illegal flow.

It is however unavoidable that `match` has to be computed from *secret* information. Declassification was invented to handle such situations, and will be discussed later.

[13] FS, the *forward slice* is defined as $FS(x) = \{y \mid x \rightarrow^* y\}$.

[14] Ignore that node 14 is only half shaded for the moment.

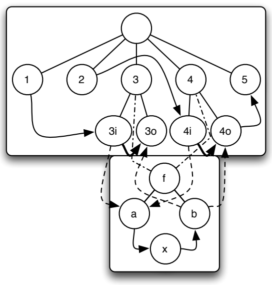

Fig. 10 SDG for Fig. 3. The statement `x` computes the return value `b` from the formal input parameter `a`.

5 Inter-procedural propagation of security levels

Let us now discuss *interprocedural IFC*. To understand the problem of context-sensitivity, consider again Fig. 3 and its SDG in Fig. 10. In this program fragment, $P(\texttt{secret}) = P(1) = High$, $P(\texttt{public}) = P(2) = Low$, and $R(\texttt{p}) = R(5) = Low$. Let us first assume that backward slices are computed just as in the intraprocedural case, that is, all nodes which have a path to the point of interest are in the slice. This naive approach treats interprocedural SDG edges like data or control dependence edges, and as a result will ignore the calling context. In the example, $3 \in BS(5)$ due to the SDG path $3 \to 3i \to a \to b \to 4o \to 5$ (where $3i$ is the actual parameter of the first call, and $4o$ is the return value of the second call). By Eq. (13), $S(4o) = S(5) = S(p) = High$, and $R(p) \not\geq S(p)$. But semantically, `secret` cannot influence `p`.

5.1 Context-Sensitive Slicing

To avoid such false alarms, an approach based on context-sensitive slicing must be used: not every SDG path is allowed in a slice, but only *realizable* paths. Realizable paths require that the actual parameter nodes to/from a function, which are on the path, must belong to the *same* call statement.[15] The reason is that a parameter from a specific call site cannot influence the result of a different call site, as all side-effects are represented as parameters. This fundamental idea to obtain context-sensitivity was introduced in [26, 45] and is called HRB slicing. In the example, the path $3 \to 3i \to a \to b \to 4o \to 5$ is not realizable and thus not context-sensitive, as actual parameter $3i$ and return parameter $4o$ do not belong to the same call site.

Interprocedural propagation of security levels is basically identical to intraprocedural propagation, but is based on the HRB backward slice which only includes realizable paths. Equation (13) and Theorem 4 still hold. In fact they hold for the naive backward slice as well, because even the naive interprocedural backward slice is correct; it is just too imprecise. Thus the following definition of interprocedural confidentiality is identical to the earlier intraprocedural definition (Definition 1), except that it relies on the interprocedural HRB slice:

Definition 2 Let a program's SDG be given, together with provided security labels P and required security labels R. The program *maintains interprocedural confidentiality*, if for every $a \in dom(R)$ and its HRB backward slice $BS(a)$, Eqs. (7) and (8) are satisfied.

Again we postpone the proof that this definition implies noninterference (Eq. (2)), but point out that the definition—as Definition 1—is solidly based on SDG correctness properties and fundamental definitions for confidentiality. A machine-checked proof is in preparation.

Equations (13) and (8) can again be interpreted as a dataflow framework. But for reasons of efficiency, we will generate all instances of these equations simultaneously while computing the HRB backward slice. This results in a set of constraints for the $S(x)$, which is solved by an offline fixpoint iteration; the latter being based on the same principles as in dataflow frameworks.

The details of the HRB algorithm are shown in Algorithm 1.[16] Backward slice computation, and thus propagation of security levels is done in two phases:

1. The first phase ignores interprocedural edges from a call site into the called procedure, and thus will only traverse to callees of the slicing criterion (i.e. is only ascending the call graph). Due to summary edges, which model transitive dependence of parameters, all parameters that might influence the outcome of a returned value are traversed, as if the corresponding path(s) through the called procedure were taken.
2. In the second phase, starting from all edges omitted in the first phase, the algorithms traverses all edges except call and parameter-in edges (i.e. is only descending the call graph.) As summary edges were traversed in the first phase, there is no need to re-ascend. Again, summary edges are used to account for transitive dependences of parameters.

For propagation of security levels, Algorithm 1 generates constraints involving S, P, and R. These constraints are derived from Eqs. (7) and (8). We will show later that a solution to these constraints enforces confidentiality.

The HRB two-phase approach ensures that only context-sensitively realizable paths are traversed: Remember that a naive transitive closure in Fig. 3 contains the path $3 \to 3i \to a \to b \to 4o \to 5$, which is not context-sensitive. HRB slicing will ignore that path: It follows the edge from 5 to $4o$, but will ignore the edge to b for the first phase. Instead, it will traverse the summary edge to $4i$ and 4. In the second phase it starts with the edge from $4o$ to b, comes to x, a and f but does not re-ascend, which means that it cannot reach $3i$. The summary edges have an essential effect, because they ensure that security levels are propagated (based on the generated constraints) as if they were propagated through the called

[15] or more precisely, parameter-in/-out nodes must form a "matched parenthesis" structure, since calls can be nested.

[16] For the time being, replace the test "$v \notin D$" (line 32) by "*true*", as in this section $D = \varnothing$; D will be explained in section 6.

Algorithm 1 Algorithm for context-sensitive IFC, based on the precise interprocedural HRB slicing algorithm

```
1  procedure MarkVerticesOfSlice($G, x$)
2  input $G$ : a system dependence graph
3    $x$ : a slicing criterion node
4  output $BS$ : the slice of $x$ (sets of nodes in $G$)
5        $C$ : the generated set of constraints
6  /* D, R, P are assumed to be global read only data */
7  begin
8    $C := \varnothing$
9    /* Phase 1: slice without descending into called procedures */
10   $BS' \leftarrow$MarkReachingVertices($G, \{x\}, \{$parameter-out$\}$)
11   /* Phase 2: slice called procedures
12      without ascending into call sites */
13   $BS \leftarrow$MarkReachingVertices($G, BS', \{$parameter-in, call$\}$)
14 end
15
16 procedure MarkReachingVertices($G, V, Kinds$)
17 input $G$ : a system dependence graph
18   $V$ : a set of nodes in $G$
19   $Kinds$ : a set of kinds of edges
20 output $M$ : a set of nodes in $G$ which are marked by this phase
21           (part of the precise backward slice)
22       $C$ : a set of constraints
23 begin
24   $M := V$
25   $WorkList := V$
26   while $WorkList \neq \varnothing$ do
27     select and remove node $n$ from $WorkList$
28     $M \cup= n$
29     foreach $w \in G$ such that $G$ contains an edge $w \longrightarrow v$
30         whose kind is not in $Kinds$ do
31       if $w \notin M$ then $WorkList \cup= w$ fi
32       if $v \notin D$ then
33         $C \cup= \{$"$S(w) \leq S(v)$"$\}$ // cf. eq. (13) or (17)
34         if $v \in dom(R)$ then
35           $C \cup= \{$"$S(v) \leq R(v)$"$\}$ // cf. eq. (8) or (17)
36         if $w \in dom(P)$ then
37           $C \cup= \{$"$P(w) \leq S(w)$"$\}$ // cf. eq. (13) or (18)
38       else
39         $C \cup= \{$"$P(v) \leq S(v)$"$\}$ // cf. eq. (20)
40         $C \cup= \{$"$S(w) \leq R(v)$"$\}$ // cf. eq. (21)
41       fi
42     od
43   od
44   return $M$
45 end
```

procedure. In the first phase, no security level is propagated into called procedures; in the second phase, no computed security level is propagated from the called procedure to the call site. Due to summary edges, no security level is "lost" at ignored edges, i.e. they ensure that the security level is propagated along transitive dependences for this calling context, but it cannot change the computed security level at another call site.

We will now argue that Algorithm 1 generates correct and sufficient constraints in order to enforce Eqs. (7) and (8). Restricting these equations to the (context-sensitive) backward slices of all points $\in dom(R)$ avoids spurious flow through procedures, and is sufficient as these slices contain all nodes affecting Eq. (8). Thus Theorem 4 is still valid in the interprocedural case, and the proof remains the same.[17] Thus the PDG-based confidentiality check also works on SDGs: for any $a \in dom(R)$, compute the HRB backward slice $BS(a)$ and check whether all $y \in dom(P) \cap BS(a)$ have $P(y) \leq R(x)$. Remember that this check is valid for any correct backward slice – but the more precise the slice, the less false alarms it generates.

Theorem 5 *For every $a \in dom(R)$, Algorithm 1 (where $D = \varnothing$) generates a set of constraints which are correct and complete, and thus enforce confidentiality according to Definition 2.*

Proof. We may assume that the HRB algorithm itself computes a correct (and precise) backward slice $BS(a)$ for any $a \in dom(R)$.

1. For any $w, v \in BS(a)$ where $w \longrightarrow v$, the algorithm generates a constraint $S(w) \leq S(v)$, which is necessary according to Eq. (13) and sufficient as edges outside $BS(a)$ cannot influence a.

2. Furthermore for any $w \in dom(P) \cap BS(a)$, $P(w) \leq S(w)$ is generated which is necessary due to Eq. (13) and sufficient as nodes outside $BS(a)$ cannot influence a. Note that line 36 tests for $w \in dom(P)$ and not $v \in dom(P)$, as otherwise nodes $\in dom(P)$ without predecessors would not generate a P-constraint.

3. Finally, for any $v \in dom(R) \cap BS(a)$, $R(v) \geq S(v)$ is generated which is necessary due to Eq. (8) and sufficient as nodes outside $BS(a)$ cannot influence a. Note that line 34 tests for $v \in dom(R)$ and not $w \in dom(R)$, as otherwise nodes $\in dom(R)$ without successor would not generate a R-constraint.

Thus Algorithm 1 generates exactly the constraints required by (8), and constraints exactly equivalent to (7). As a consequence they have the same fixpoint, and fulfill the requirements of Definition 2. □

For pragmatic reasons, the fixpoint computation ignores the constraints involving R; these are only incorporated in the SDG-based confidentiality check after a solution for S has been found. The reason is that otherwise illegal flows will show up as an unsolvable constraint system – which is correct, but prevents user-friendly diagnosis. If the R constraints are checked later and one (or more) will fail, chops can be computed for diagnosis as described in Sect. 4.3.

For the example above (Figs. 3,10), Algorithm 1 computes $BS(5) = \{5, 4o, 4i, 4, 2\}$ in the first phase and adds $\{b, x, a\}$ in the second phase, thus avoiding adding $3i$ or 1 to $BS(5)$. This is context-sensitivity. The corresponding constraints are $S(5) \leq R(5), S(4o) \leq S(5), S(4i) \leq S(4o), S(4) \leq S(4i), S(4) \leq S(4o), S(b) \leq S(4o), S(x) \leq S(b), s(a) \leq S(x)$. Constraints for $BS(3)$ are computed similarly. Figure 11 presents the complete list of constraints. It also presents additional constraints which would be added by naive interprocedural slicing (printed in gray). The fixpoint for S (without P constraints) is presented in Fig. 11 (right column; again results based on naive slicing are shown in gray). The precise

[17] In fact we need a version of Theorem 3 working on SDGs and HRB backward slices; which is left as an exercise to the reader.

Constraints	**Minimal Fixpoint**
$S(1) \leq S(3i)$	$S(1) = \mathit{Low}/\mathit{High}$
$S(2) \leq S(4i)$	$S(2) = \mathit{Low}$
$S(3) \leq S(3i) \wedge S(3) \leq S(3o) \wedge S(3) \leq S(f)$	$S(3) = \mathit{Low}/\mathit{High}$
$S(3i) \leq S(3o) \wedge S(3i) \leq S(a)$	$S(3i) = \mathit{Low}/\mathit{High}$
$S(3o) \leq \top$	$S(3o) = \mathit{High}$
$S(4) \leq S(4i) \wedge S(4) \leq S(4o) \wedge S(4) \leq S(f)$	$S(4) = \mathit{Low}$
$S(4i) \leq S(4o) \wedge S(4i) \leq S(a)$	$S(4i) = \mathit{Low}$
$S(4o) \leq S(5)$	$S(4o) = \mathit{Low}$
$S(f) \leq S(a) \wedge S(f) \leq S(b)$	$s(f) = \mathit{Low}$
$S(a) \leq S(x)$	$S(a) = \mathit{Low}$
$S(x) \leq S(b)$	$S(x) = \mathit{Low}$
$S(b) \leq S(3o) \wedge S(b) \leq S(4o)$	$S(b) = \mathit{Low}$
$S(5) \leq R(5) \wedge R(5) = \mathit{Low}$	$S(5) = \mathit{Low}$
$P(1) \leq S(1) \wedge P(1) = \mathit{High}$	
$P(2) \leq S(2) \wedge P(2) = \mathit{Low}$	

Fig. 11 Constraint system for Fig. 3 generated by Algorithm 1. Parts in gray are only generated for context-insensitive analysis.

solution correctly computes $S(1) = \mathit{High}$, and indeed $P(1) = \mathit{High} \leq S(1)$. The naive solution would compute $S(1) = \mathit{Low}$ and generates a false alarm due to $P(1) \not\leq S(1)$.

5.2 Backward Flow Equations

Note that Eq. (13) in fact employs a forward propagation approach: it shows how to compute $S(x)$ if the $S(y)$ for the predecessors y of x are known. The HRB algorithm essentially works just the other way, namely backwards. For reasons of implementation efficiency, previous work has presented flow equations that follow this backward propagation approach.

In this section, we will show how to transform Eqs. (13) and (8) into an equivalent form which mirrors this backward propagation, while Theorem 4 still holds. This will allow a more efficient implementation in connection with the HRB algorithm. The equivalent backward form is based on the following observation: Eq. (5) demands that for every $x \in N$ and $y \in \mathit{pred}(x)$, $S(x) \geq S(y)$ and thus $S(x) \geq \bigsqcup_{y \in \mathit{pred}(x)} S(y)$. The same set of constraints can be expressed as follows: for every $x \in N$ and $y \in \mathit{succ}(x)$, $S(x) \leq S(y)$ (Eq. (11)), and as a consequence,

$$S(x) \leq R'(x) \sqcap \bigsqcap_{y \in \mathit{succ}(x)} S(y) \tag{17}$$

In analogy, for Eq. (8) ($a \in \mathrm{dom}(R)$, $S(a) \leq R(a)$), one gets:

$$\forall a \in \mathrm{dom}(P) : P(a) \leq S(a) \tag{18}$$

Theorem 6 *For any PDG and (intraprocedural) slice in the PDG, the collected instances of Eqs. (11) and (8) generate the same set of constraints as the collected instances of Eqs. (17) and (18).*

Proof. (for full details see [19]). The individual constraints in Algorithm 1 are equivalent due to the duality $a \leq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a$, which has been exploited in the construction of Eqs. (17) and (18). In forward propagation, we are using a two-phase algorithm that initially ignores constraints involving R in the fixpoint iteration and subsequently checks the omitted constraints with the computed fixpoint of S. In backward propagation, the fixpoint for S is determined without constraints involving P constraints, which again are checked in a second phase. Therefore it is obvious that the fixpoint for S in forward propagation differs from the fixpoint in backward propagation, however, both methods check that the whole set of constraints generated on all paths between all nodes in $\mathrm{dom}(P)$ and all nodes in $\mathrm{dom}(R)$ is satisfied and are thus equivalent. □

Computing a minimal fixpoint for $S(x)$ from constraints involving S and R and subsequent checking constraints involving P (backward propagation) is therefore equivalent to computing S's fixpoint from S and P with subsequent checking of R-constraints (forward propagation).

6 Declassification

IFC as described so far is too simplistic because in some situations one might accept that information with a higher security level flows to a "lower" channel. For instance, information may be published after statistical anonymization, secret data may be transferred over the Internet using encryption, and in electronic commerce one needs to release secret data after its purchase. *Declassification* allows the security level of incoming information to be lowered as a means to relax the security policy. The password checking method presented earlier is another example: as password tables are encrypted, it does not matter that information from the password table flows to the visible method result, and hence a declassification to *public* at node 14 (where the illegal flow was discovered, see Sect. 4) is appropriate – a password-based authentication mechanism necessarily reveals some information about the secret password.[18]

When allowing such exceptions to the basic security policy, one major concern is that exceptions might introduce unforeseen information release. Several approaches for a semantics of declassification were proposed, each focusing on certain aspects of "secure" declassification. The current state of the art describes four dimensions to classify declassification approaches according to *where*, *who*, *when* and *what*

[18] We all know that password crackers can exploit this approach in case weak passwords are used, hence uncontrolled introduction of declassification may cause security problems. Thus in the example, at least mechanisms to prevent weak passwords are necessary. In general, a better semantic foundation for declassification will be necessary, which is a topic of ongoing research.

can be declassified [51]. Apart from that, some basic principles are presented that can serve as "sanity checks" for semantic security policies allowing declassifications. These principles are (1) semantic consistency, which is basically invariance under semantics-preserving transformations; (2) conservativity, i.e. without declassification, security reduces to noninterference; (3) monotonicity of release, which states that adding declassification should not render a secure program insecure; (4) non-occlusion which requires that declassification operations cannot mask other covert information release.

6.1 Declassification in SDGs

We model declassification by specifying certain SDG nodes to be declassification nodes. Let $D \subseteq N$ be the set of declassification nodes. A declassification node $x \in D$ must have a required and a provided security level:

$$x \in D \implies \big(x \in dom(P) \cap dom(R)\big) \tag{19}$$

Information reaching x with a maximal security level $R(x)$ is lowered (declassified) down to $P(x)$. Note that usually $R(x) \geq P(x)$, as declassification should lower a level, not heighten it.[19] Now a path from node y to a with $P(y) \not\leq R(a)$ is *not* a violation, if there is a declassification node $x \in D$ on the path with $P(y) \leq R(x)$ and $P(x) \leq R(a)$ (assuming that there is no other declassification node on that path). The actual security level $S(x)$ will be between $P(x)$ and $R(x)$. In the password example, $D = \{14\}, R(14) = secret, P(14) = public$; and the illegal flow described earlier disappears.

According to Sabelfeld and Sands [51], this policy for expressing intentional information release describes *where* in the system information is released: The set D of declassification nodes correspond to code locations—moreover, in the implemented system the user has to specify the code locations, which are mapped to declassification nodes by the system.

In terms of the propagation equations, a declassification simply changes the computation of S. Equation (11) must be extended as follows:

$$S(x) \geq \begin{cases} P(x) & \text{if } x \in D \\ P'(x) \sqcup \bigsqcup_{y \in pred(x)} S(y) & \text{otherwise} \end{cases} \tag{20}$$

Thus the incoming security levels are ignored and replaced by the declassification security level.

Of course, Eq. (8) is still valid for non-declassification nodes, but for $x \in D$ it must be modified as $S(x)$ is the declassified value:

$$\forall x \in dom(R) \setminus D : R(x) \geq S(x)$$
$$\wedge\ \forall x \in D : R(x) \geq \bigsqcup_{y \in pred(x)} S(y) \tag{21}$$

[19] There are examples where the declassification goes sideways in the lattice, hence $R(x)$ and $P(x)$ are incomparable.

which expresses that normal flow of S is interrupted at $x \in D$.

The following definition resembles Definition 2, but incorporates the modified flow equations:

Definition 3 Let a program's SDG be given, together with provided security labels P, required security labels R, and declassification nodes D. The program *maintains confidentiality under declassification*, if for all $a \in dom(R)$ Eqs. (20) and (21) are satisfied.

In case $D = \varnothing$ Definition 3 is identical to Definition 2, thus our approach to declassification is conservative. But let us point out clearly that Definition 3 is not based on a program's semantics alone, and hence a theorem analogous to Theorem 2 or Theorem 4 cannot be proved. The reason is that the engineer is basically free in the placement of declassifications, and hence a semantics of confidentiality under declassification is difficult to formalize. Ongoing research has identified rules for "reasonable" use of declassification, such as [51], but a concise formalization of "sound" declassification has not been achieved yet. Two of the principles identified so far are called monotonicity of release and conservativity, and we will show later that our definition respects both. This demonstrates that Definition 3 is a natural extension of Definitions 1 and 2.

Theorem 7 *For every $a \in dom(R)$, Algorithm 1 (where $D \neq \varnothing$) generates a set of constraints which are correct and complete, and thus enforce confidentiality according to Definition 3.*

Proof. We have already argued (proof for Theorem 5) that for non-declassification nodes the generated constraints correspond exactly to Eqs. (8) and (7), and thus to the non-declassification cases in Eqs. (20) and (21). For declassification nodes $d \in D$, Algorithm 1 does no longer generate constraints $S(w) \leq S(d)$, which is indeed required by (20), case $x \in D$. Instead it generates $R(d) \geq S(w)$ for $w \in pred(d)$, which is equivalent to the constraints required in (21), case $x \in D$. Furthermore, it generates $S(d) \geq P(d)$ which is exactly required by (20), case $x \in D$.

Thus Algorithm 1 generates exactly the constraints required by (20) and (21). Hence they have the same fixpoint, and fulfill the requirements of Definition 3. □

In case $D = \varnothing$, Algorithm 1 by Theorem 5 checks noninterference without declassification. Thus we obtain for free the

Corollary 1 (Conservativity of Declassification) *Algorithm 1 is conservative, that is, without declassification it checks standard noninterference.*

Let us finally point out a few special situations. It is explicitly allowed to have two or more declassification on one specific path, e.g. $x \to^* d_1 \to^* d_2 \to^* y$. But this only makes sense if $P(d_1) \leq R(d_2)$, as otherwise no legal flow is possible on the path. If there is no other path $x \to^* d_2$, it also makes sense to demand $P(d_2) \leq P(d_1)$, as otherwise the second declassification is redundant.

In case there are several declassifications on disjoint paths from x to y, for example $x \to^* d_1 \to^* y$, $x \to^* d_2 \to^* y$, $x \to^* d_3 \to^* y$, ..., it is possible to approximate all these declassifications conservatively by introducing a new declassification d where $R(d) = \bigsqcap_i R(d_i)$ and $P(d) = \bigsqcup_i P(d_i)$. Any flow which is legal through d is also legal through (one of) the d_i, hence the approximation will not introduce new (illegal) flows. This observation seems unmotivated, but indeed was the motivation for a more precise interprocedural IFC, as described in Sect. 7.

6.2 Monotonicity of Release

Another useful property is *monotonicity of release*, which states that introduction of an additional declassification should not make previously secure programs insecure (i.e. generate additional illegal flow). Formally, this can be defined as follows:

Definition 4 Let a program with SDG $G = (N, \to)$, declassification nodes D, required security labels R, and provided security labels P, satisfy confidentiality under declassification according to Definition 3. Let $d \in N$ where $d \notin D \cup dom(R) \cup dom(P)$. Let l, l' be two security levels.

Declassification d *respects monotonicity of release*, if the program with SDG G, declassification nodes $D_d = D \cup \{d\}$, provided security labels $P_d = P[d \mapsto l']$ and required security labels $R_d = R[d \mapsto l]$ again satisfies confidentiality under declassification according to Definition 3.

The following theorem states that if the annotations comply with some basic sanity checks, then monotonicity of release can be guaranteed:

Theorem 8 *Let G, P, R, D, d, P_d, R_d as in Definition 4. Let S_d be computed for G using D_d, P_d, R_d according to Eq. (20).*

If $R_d(d) \geq \bigsqcup_{y \in pred(d)} S(y)$, and $P_d(d) \leq \bigsqcup_{y \in pred(d)} S(y)$, then for all $x \in N$, $S_d(x) \leq S(x)$.

The first premise avoids that previously legal flow (where $R'(d) = \top$ as $d \notin dom(R)$) is now blocked by a too low or arbitrary $R_d(d)$. The second premise analogously avoids that a declassification generates new illegal flows as the outgoing declassification level is too high. Both premises together imply $P_d(d) \leq R_d(d)$, that is, "sideways" declassification is not covered by our theorem. In practice, both requirements are easy to check and do not restrict sensible declassification.

Proof. In the original SDG, $\bigsqcup_{y \in pred(d)} S(y) \leq S(d) \leq \bigsqcap_{y \in succ(d)} S(y)$, hence in the new SDG $S_d(d) = P_d(d) \leq S(d) = \bigsqcup_{y \in pred(d)} S(y)$ by assumption and Eq. 20. Besides, $S_d(d) \leq S(d) \leq \bigsqcap_{y \in succ(d)} S(y) \leq S(y)$ for all $y \in succ(d)$. Hence $S(y) \geq \bigsqcap_{z \in pred(y)} S(y) \geq S_d(y) = S_d(d) \sqcup \bigsqcup_{z \neq d \in pred(y)} S(z)$. The same argument works for the successors of y. By induction[20] $S_d(x) \leq S(x)$ follows for all x. Thus Eq. (21) still holds in the modified SDG, hence confidentiality under declassification holds. □

[20] technically, a well-known fixpoint induction

```
1  int foo(int x) {
2    y = ... x ... // compute y from x
3    return y; /*D:confidential -> public*/
4  }
5
6  int check() {
7    int secret = ...  /*P:secret*/
8    int high = ...    /*P:confidential*/
9    int x1, x2;
10   x1 = foo(secret);
11   x2 = foo(high);
12   return x2; /*R:public*/
13 }
```

Fig. 12 Example for declassification

Corollary 2 *Under the assumptions of Theorem 8, declassification d respects monotonicity of release.*

Proof. For the original SDG, (8) and (20) are valid for S. In the new PDG, (20) is by construction valid for S_d, and (8) is valid for S_d since by the theorem $S_d(x) \leq S(x)$. □

6.3 Confidentiality check with declassification

The original SDG-based confidentiality criterion no longer works with declassification, as information flow with declassification is no longer transitive and slicing is based on transitive information flow. Thus a $P(x)$ in $BS(a)$ where $P(x) \not\leq R(a)$ is not necessarily an illegal flow, as $P(x)$ can be declassified under way. Instead, the criterion must be modified as follows:

Confidentiality Check With Declassification. For every $a \in dom(R) \setminus D$, compute $S(x)$ for all $x \in BS(a)$ by Algorithm 1, and check the following property:

$$\forall x \in dom(P) \cap BS(a) : P(x) \leq S(x) \tag{22}$$

Theorem 7 guarantees that the constraints generated by Algorithm 1 and thus the S values (being their minimal fixpoint) are correct. Hence the criterion is satisfied iff Definition 3 is satisfied. If the criterion is not satisfied, Eq. (20) is violated and an illegal flow has been detected. As described in Sect. 5.1, the S values are computed first, and the criterion (22) is checked in a second phase; this allows to generate diagnostics by computing chops.

Let us return to the example in Figs. 4 and 9 and assume $R(14) = public$. As described in Sect. 4.3, the analysis reveals an illegal flow $3 \to^* 14$. We thus introduce a declassification: $14 \in D$, $R(14) = secret$, $P(14) = public$ (represented as two colors). Now $S(14) = secret \leq R(14)$, so the confidentiality check will no longer reveal an illegal flow. This may be desirable depending on the security policy, as only a small amount of information leaks from password checking.

As another example consider Fig. 12. In line 3 a declassification $D : confidential \to public$ is present. Hence $3 \in D$, $R(3) = confidential$ and $P(3) = public$. It seems that a *secret* value can flow from line 10 to line 3, hence in line 3 an illegal flow seems possible ($R(3) \not\geq S(3)$) because in line 3 we

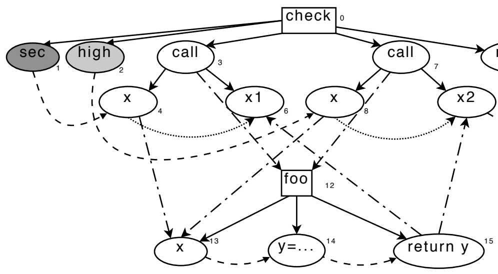

Fig. 13 System Dependence Graph for Fig. 12

can declassify from *confidential* to *public* but not from *secret* to *public*. But in fact the return value in line 3 is only copied to `x1` at line 10, and `x1` is dead (never used afterwards and never output). Thus intuitively, the program seems secure.

The SDG for this program is shown in Fig. 13. By Algorithm 1 `x1` is not in the context-sensitive backward slice for line 12, and thus the SDG-based confidentiality criterion will *not* generate a false alarm, but determine that confidentiality is guaranteed. This example demonstrates once more how context-sensitive backward slices improve precision.

7 Improving Interprocedural Declassification

Algorithm 1 is correct, but in the presence of declassifications, its precision still needs to be improved. The reason is that Algorithm 1 essentially ignores the effect of declassifications in called procedures: summary edges represent a transitive information flow between pairs of parameters, whereas declassification is intransitive. Using them for computation of the actual security level $S(x)$ implies that every piece of information flowing into a procedure with a given provided security level l will be treated as if it flowed back out with the same level. If there is declassification on the path between the corresponding formal parameters, this approach is overly conservative and leads to many false alarms.

As an example, consider Fig. 13 again, which is the SDG for the interprocedural declassification in Fig. 12. The required security level for node 11 is *Low* as specified. Algorithm 1 computes $S(2) = S(8) = S(10) = S(11) = Low$ due to the summary edge. This will result in a false alarm because the declassification at node 15 is ignored.

7.1 Summary Declassification Nodes

In order to respect declassifications in called procedures, and to achieve maximal precision, an extension of the notion of a summary edge is needed. The fundamental idea is to insert a new "summary" declassification node into the summary edge, which represents the effect of declassifications on paths in the procedure body.

Thus the summary edge $x \rightarrow y$, representing all paths from the corresponding formal-in node x' to the formal-out node y', is split in two edges, with a declassification node

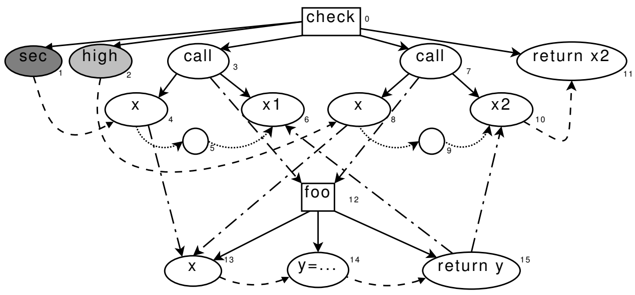

Fig. 14 SDG for Fig. 12 with summary declassification nodes

$d \in D$ in between. This new declassification node d represents the declassification effects on all paths from x' to y'.

The constraints on $R(d)$ and $P(d)$ are chosen such that any legal flow through the procedure body is also a legal flow through $x \rightarrow d \rightarrow y$. In particular, if there is a declassification free path from x' to y', there must not be a summary declassification node, as information flow might be transitive in that case. It is not trivial to determine $R(d)$ and $P(d)$ such that precision is maximized and correctness is maintained, as we will see later. However, once these values have been fixed, Algorithm 1 proceeds as usual.

Figure 14 shows the SDG with summary declassification nodes for the example in Fig. 13. The actual-in nodes 4 and 8 are connected to their corresponding formal-in node 13 with parameter-in edges. The formal-out node 15 is connected to corresponding actual-out nodes 6 and 10 with parameter-out edges. The call nodes 3 and 7 are connected to the called procedure at its entry node 12 with a call edge. The actual-in nodes 4 and 8 are connected via summary edges and summary declassification nodes 5 and 9 to the actual-out nodes 6 and 10. This figure contains only one declassification at node 15 ($R(15) = confidential$, $P(15) = public$), so for the path between node 13 and 15 the summary declassification nodes 5 and 9 will be set to $R(5) = R(9) = confidential$ and $P(5) = P(9) = \bot$. (The algorithm for this will be presented in the next section.)

Exploiting the summary declassification nodes, Algorithm 1 will (a) determine that node 1 is not in the backward slice of node 11 and thus cannot influence node 11, and (b) $confidential = P(8) \leq S(8) \leq S(8) \leq R(9) = confidential$ and $\bot = P(9) \leq S(9) \leq S(10) \leq R(11) = public$, thus no security violation is found in `check`. In the second slicing phase, there is no violation either: $S(13) \leq S(14) \leq R(15) = confidential$ and $public = P(15) \leq S(10) \leq R(11) = public$. Note that the constraint $P(15) \leq S(10)$ is checked in the second phase, such that the trivial constraint $P(9) = \bot$ is sufficient for asserting security, and $R(5) = R(9) = confidential$ is exactly the maximal possible value of $S(13)$. This observation leads to the following algorithm for computing P and R for summary declassification nodes.

Algorithm 2 Computation of $R(d)$ for Summary Declassification (backward propagation)

```
procedure SummaryDeclassification($(G, L, R)$)
input $G$: a system dependence graph
  $L$: a security lattice
  $R$: the required annotations
output: the set of summary declassification nodes (included in $G$)
begin
  $pathEdges = \varnothing$ // set of transitive dependences already seen
  foreach formal−out node $o$ in $G$ do
    $pathEdges \cup= (o, o, \neg(o \in D), \top)$
  od
  $workList := pathEdges$ // deep copy required here
  while $workList$ not empty do
    remove $(x, y, f, l)$ from workList
    if $x$ is an formal−in node then
      addSummaries$(x, y, f, l)$
    else
      foreach edge $w \longrightarrow x \in G$ do
        addToPathEdges(extendPathEdge$((x, y, f, l), w)$)
      od
    fi
  od
end

procedure addToPathEdges$(x, y, f, l)$
input $(x, y, f, l)$: a path edge tuple
begin
  if $(x, y, f', l') \in pathEdges$ where $f' \neq f$ or $l' \neq l$ then
    remove $(x, y, f', l')$ from $pathEdges$
  fi
  if $(x, y, f, l) \notin pathEdges$ then
    $pathEdges \cup= (x, y, f, l)$
    $workList \cup= (x, y, f, l)$
  fi
end
```

Algorithm 3 Auxiliary procedures for Summary Declassification Nodes

```
procedure extendPathEdge$((x, y, f, l), w)$
input: $(x, y, f, l)$: a path edge tuple
  $w$ the extension node
output: a path edge extended by $w$
begin
  if $pathEdges$ contains a tuple $(w, y, f', l')$ then
    retrieve $(w, y, f', l')$ from $pathEdges$
  else
    $l' = \top, f' = false$
  fi
  if $x \in D$ then
    $l' = l' \sqcap R(x)$ // equiv. $S(w) \leq R(x)$
  else
    $l' = l' \sqcap l$ // equiv. $S(w) \leq S(x)$
  fi
  $f' = f' \vee (w \notin D \wedge f)$
  return $(w, y, f', l')$
end

procedure addSummaries$(x, y, f, l)$
input: $(x, y, f, l)$: a path edge tuple
begin
  foreach actual parameter pair $(v, w)$ corresponding to $(x, y)$
    if $f$ then
      add summary edge $v \longrightarrow_{sum} w$ to $G$
      $n := v$
    else
      add summary declassification node $d$ and edges $v \longrightarrow_{sum} d$
        and $d \longrightarrow_{sum} w$ where $R(d) = l$ and $P(d) = \bot$
    fi
    foreach $(w, z, f', l') \in pathEdges$ do
      addToPathEdges(extendPathEdge$((w, z, f', l'), d)$)
    od
  od
end
```

7.2 Computation of $R(d)$ for Summary Declassification Nodes

Let a summary edge $x \longrightarrow y$ for method m be given. To determine an appropriate $R(d)$ value for a summary declassification node d (where $x \longrightarrow d \longrightarrow y$), we exploit three facts:

- If there is an interprocedurally realizable path $x \longrightarrow x' \longrightarrow^+ y' \longrightarrow y$ through the corresponding formal parameters x' and y' of m's body without declassification nodes, the summary edge $x \longrightarrow y$ must not contain a summary declassification node either, as in this case information at y depends on x without being declassified under way.
- Otherwise, phase 2 of the HRB algorithm will follow $x' \longrightarrow^+ y' \longrightarrow y$ through m's body as usual, but it will ignore $x \longrightarrow x'$ (see Algorithm 1): Therefore, the effect of declassification nodes along this path is collected in form of constraints for $S(x')$, and $R(d) = S(x')$.
- In case there are several paths $x \longrightarrow x' \longrightarrow^+ y' \longrightarrow y$ where $x' \longrightarrow a_i$ $(i = 1..k)$ are the corresponding initial edges on each path, the infimum of the $S(a_i)$ as computed by the HRB algorithm for the individual paths must be used, as the "weakest" declassification defines the requirements for the summary edge: $R(d) = \bigsqcap_{i=1..k} S(a_i)$.

Algorithms 2 and 3 implement this approach as an extension of the summary edge algorithm described by Reps et al. [45], which is a prerequisite for precise HRB slicing. Algorithm 2 inserts summary declassification nodes representing declassification effects of called methods; Algorithm 3 contains auxiliary procedures.

The set $pathEdges$ contains quadruples (x, y, f, l) representing the start x and end node y of a path, together with a flag f indicating whether a path represented by this quadruple contains no declassifications, and the computed security level l. Like the original summary algorithm, the extended algorithm inserts all formal-out nodes of G into this set, and initializes a $workList$ with the same elements. Just like the original algorithm, summary information is added once a formal-in node is encountered. In all other cases, the pathEdge information is extended according to the annotations of all predecessors w of the start node x. The new quadruple is inserted into both $pathEdges$ and the $workList$; if old information was present in $pathEdges$, then this information is removed first. Note that due to the monotonicity of the operations in extendPathEdge and the limited height of the Boolean and the security lattice, this algorithm is guaranteed to terminate.

Procedure addSummaries takes a quadruple from a formal-in to a formal-out node and inserts summary information between all corresponding actual parameter pairs: if f is true, i.e. there is a path between x and y with no declassification, a traditional summary edge is inserted. Otherwise, the security level will be propagated from the actual-out node into the called method in the second phase of backward slicing. The provided security level for the actual-out node does not need to impose any additional constraints (and is therefore $\bot$). As explained above, the required security level corresponds to the infimum of all required security levels of declassifications on the way which are not killed by later declassifications. This value has been determined in l, so $R(d) = l$. Finally, all quadruples starting at the actual-out node corresponding to y need to be extended by d and inserted into the worklist.

The procedure extendPathEdge extends the path by a predecessor w of start node x. If *pathEdges* already contains a quadruple from w to y, then it is retrieved, otherwise the f' and l' entries are initialized with neutral elements. Now the security level l' of the new quadruple is computed according to the backward flow equations (e.g. 17), and the new flag f' includes whether a path to w contains no declassification. The quadruple (w, y, f', l') thus contains the information concerning all paths from w to y seen so far. If another path from w to y is found later in the analysis, this quadruple will be retrieved in the first step and taken as operand to the infima as required by the flow equations.

As an example, consider Fig. 12 again. Algorithm 2 starts with adding node 15 as $(15, 15, \mathit{false}, \top)$ into *pathEdge*. Note that the third element of this tuple is *false*, because 15 is a declassification node. When this tuple is removed from the *workList*, all predecessors of 15 are processed, in particular node 14. For this node, extendPathEdge will not find a previous tuple in *pathEdges* and thus initializes l' and f' to the neutral elements for $\sqcap$ and $\vee$. As node 15 is in D, $l' = l' \sqcap R(15) = R(15)$ and $f' = f' \vee \mathit{false} = \mathit{false}$, which yields a pathEdge tuple $(14, 14, \mathit{false}, \mathit{confidential})$. For its predecessor 13, we get a pathEdge tuple $(13, 13, \mathit{false}, \mathit{confidential})$ and no other path leads to 13. Since 13 is a formal-in node, addSummaries will add a summary declassification node d where $R(d) = \mathit{confidential}$ and $P(d) = \bot$ between the corresponding actual parameters 4 and 6, and 8 and 10, exactly as we defined these nodes in the previous section.

Theorem 9 *IFC with Algorithm 1 and summary declassification nodes determined according to Algorithms 2 and 3 is sound and precise.*

Proof (for full details see [19]). We want to show that Algorithms 2 and 3 results in a superset of the constraints generated for all interprocedurally realizable paths. This guarantees soundness. To demonstrate precision, we show that the additional constraints are trivially satisfied by choosing $P(d) = \bot$ for all summary declassification nodes d, and thus do not change the computed fixpoint. As these algorithms are straightforward extensions of the algorithm presented in [26, 45], we can assume that these algorithms traverse all interprocedurally realizable paths between formal-in and formal-out edges, including recursive calls. For soundness, we need to show two subgoals:

1. If there is an interprocedurally realizable path between a formal-in and a formal-out parameter of the same call-site that does not contain a declassification node, then the algorithm will only generate a traditional summary edge, but no summary declassification node. Due to transitivity of information flow on that path, the summary information must conservatively obey transitivity as well. Algorithm 3 adheres to this requirement using the flag f in line 58. An induction over the length of the pathEdge will show this property:
   If the length of the pathEdge is 0 ($x = y$), line 9 asserts that if $x \in D$ then the flag f is *false*, else *true*. So let's assume f correctly represents the fact if the pathEdge (x, y, f, l) contains no declassifications. Then line 50 asserts that f' in the extended pathEdge (w, y, f', l') is *true* (i.e. there is no declassification on the path $w \to y$), if $w \notin D \wedge f$ holds. Note that if there have been other paths between w and y previously explored (the condition in line 40 holds), we will only remember if there is *any* path without declassification due to the disjunction in line 50.
2. Otherwise, if all paths between a formal-in and a formal-out parameter of the same call-site contain a declassification, we need to show that for each interprocedurally realizable path, the HRB algorithm with summary declassification nodes computes a superset of the constraints generated for that path. As a consequence of not traversing parameter-in edges, the constraint $S(\text{act-in}) \leq S(\text{form-in})$ is not directly generated by Algorithm 1, and thus must be imposed by the summary declassification node d. As the value of $R(d)$ is determined by computing $S(\text{form-in})$ with the same constraints as in Algorithm 1, we only need to show that using $S(\text{form-out}) = \top$ we get the same result as with the constraint $S(\text{form-out}) \leq S(\text{act-out})$, which is generated by the HRB algorithm. But this follows from the independence of $S(\text{form-in})$ and $S(\text{form-out})$, as each path in-between contains a declassification node which induces no constraint of the form $S(w) \leq S(v)$ for an edge $w \to v$ but only $S(w) \leq R(v)$. □

8 Implementation and preliminary experience

We have implemented SDG-based IFC, including declassification, as described in this paper. The prototype is an Eclipse plugin, which allows interactive definition of security lattices, automatic generation of SDG's, annotation of security levels to SDG nodes via source annotation and automatic security checks [16]. All mechanisms described in previous parts of the paper have been implemented and are working.

In this article, we will not explain details about the implementation and its user interface; nor will we present empirical studies about precision, scalability, and practicability. All these results will be presented in a separate, forthcoming article as well as in [19]. Right here, we present just a few remarks about preliminary case studies.

Our largest object of study is the `Purse` applet from the "Pacap" case study [9]. This program is written in JavaCard and contains all JavaCard API PDGs and stubs for native API methods. The program is 9,835 lines long. The SDG (including necessary API parts) consists of 135,271 nodes and 1,002,589 edges. The time for SDG construction was 145 seconds, plus 742 seconds for generation of summary edges. The latter is measured separately as it is only necessary for context-sensitive slicing but requires an $O(n^3)$ algorithm.

Next, 30,332 backward slices were selected by choosing a random node as a starting point. The average slice size is 86,023 nodes, which is about 64% of the whole source code. This is more than what is typical for backward slices, due to the higher coupling of JavaCard in contrast to normal Java, and illustrates why precise witnesses can only be achieved via path conditions as described in [21, 22, 47].

As a case study for IFC we chose another JavaCard applet called `Wallet`.[21] It is only 252 lines long but with the necessary API parts and stubs the SDG consists of 18,858 nodes and 68,259 edges. The time for SDG construction was 8 seconds plus 9 for summary edges.

The Wallet stores a balance that is at the user's disposal. Access to this balance is only granted after supplying the correct PIN. We annotated all statements that update the balance with the provided security level *High* and inserted a declassification to *Low* into the `getBalance` method, as the result is meant to be displayed at the JavaCard terminal. The methods `credit` and `debit` may throw an exception if the maximum balance would be exceeded or if there is insufficient credit. In such cases JavaCard applets throw an exception, and the exception is clearly dependent on the result of a condition involving balance. The exception is not meant to be caught but percolates to the JavaCard terminal, so we inserted declassifications for these exceptions, as well. Besides this intended information flow, which is only possible upon user request and after verifying the PIN, our analysis proved that no further information flow is possible from the balance to the output of the JavaCard.

Further evaluations with respect to precision, scalability, and usability, in particular with standard Java programs are described in a forthcoming thesis [19].

9 Related Work

9.1 SDGs and IFC

Several papers have been written about SDGs and slicers for Java, but to our knowledge only the Indus slicer [28] is—besides ours—fully implemented and can handle full Java. Indus is customizable, embedded into Eclipse, and has a very nice GUI, but is less precise than our slicer. In particular, it does not fully support context-sensitivity but only k-limiting of contexts, and it allows time traveling for concurrent programs.

The work described in this paper improves our previous algorithm [21], which was not able to handle declassification in called procedures precisely. However, that work also describes the generation and use of path conditions for Java PDGs (i.e. necessary conditions for an information flow between two nodes), which can uncover the precise circumstances under which a security violation can occur.

While a close connection between IFC and dataflow analysis had been noticed very early [7], Abadi et al. [1] were the first to connect slicing and noninterference, but only for type system based slicing of a variant of λ-calculus. It is amazing that our Theorem 3 from Sect. 2 (which holds for imperative languages and their PDGs) was not discovered earlier. Only Anderson et al. [4] presented an example in which chopping can be used to show illegal information flow between components which were supposedly independent. They do not employ a security lattice, though.

Yokomori et al. [58] also developed an IFC analysis for a procedural language based on program slicing. It checks for traditional noninterference, and supports the minimal lattice $Low < High$ only. Their analysis is flow-sensitive, but not context-sensitive nor object-sensitive.

9.2 Security type systems

Volpano and Smith [56] presented the first security type system for IFC. They extended traditional type systems in order to check for pure noninterference in simple while-languages with procedure calls. The procedures can be polymorphic with respect to security classes allowing context-sensitive analysis. They prove noninterference in case the system reports no typing errors. An extension to multi-threaded languages is given in [53].

Myers [38] defines Jif, an extension of the Java language with a type system for information flow. We already discussed in Sect. 2 that type systems are less precise, but are more efficient. JIF supports generic classes and the decentralized label model [38]; labels and principals are first class objects. Note that our PDG-based approach can be generalized to utilize decentralized labels.

Barthe and Rezk [5] present a security type system for strict noninterference without declassification in a Java-like bytecode language, handling classes and objects. `NullPointerException` is the only exception type allowed. Only values annotated with *Low* may throw exceptions. Constructors are ignored, instead objects are initialized with default values. A proof showing the noninterference property of the type system is given.

Mantel and Reinhard [35] defined the first type system for a multi-threaded while language that controls the *what* and the *where* dimension of declassification simultaneously.

[21] `http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html`

The type system is based on a definition for the *where* dimension that supersedes their previous definition of intransitive noninterference [36], and two variants of a definition for the *what* dimension similar to selective dependency [14]. However, they do not show whether their approach is practically viable.

9.3 Static analysis for security

Static analysis is often used for source code security analysis [12]. For example, IFC is closely related to tainted variable analysis. There are even approaches like the one from Pistoia et al. [43] that use slicing for taint analysis or the one from Livshits and Lam [34] that uses IPSSA, a representation very similar to dependence graphs. However, these analyses only use a trivial security level (untainted/tainted) with a trivial declassification (untaint) and could greatly benefit from our approach. Scholz et al. [52] present a static analysis that tracks user input on a data structure similar to a dependence graph. Like our analysis, it is defined as a dataflow analysis framework and reduce the constraint system using properties of SSA form. Again, this analysis is targeted to bug tracking and taint analysis.

Clark et al. [13] compute dependences using a flow logic instead of using a PDG. Their direct dataflow equations are solved by fixedpoint iteration and yield a flow-sensitive solution. This approach handles while-languages with non-recursive functions and no dynamic memory allocation. But note that this approach is not context-sensitive, as actual parameters of different calling contexts are merged. A manual correctness proof is available but no implementation or evaluation has been reported.

Pistoia et al. [42] survey recent methods for static analysis for software security problems. They focus on stack- and role-based access control, information flow and API conformance. A unified access-control and integrity checker for information-based access control, an extension of Java's stack-based access control mechanism has been presented in [41]. They show that an implicit integrity policy can be extracted from the access control policy, and that the access control enforces that integrity policy.

10 Future Work

The reader will have noticed that this article focused on theoretical and algorithmic foundations of our IFC method. It did not say much about practical aspects of our work. Neither did we describe the implementation and its GUI in detail, nor did we present empirical data on precision and scalability. Both aspects have been left out for lack of space, but will be discussed in the forthcoming PhD thesis [19]. In any case, more case studies are needed, and a more detailed comparison with other IFC algorithms in terms of precision, scalability and practicability is required.

Note that right now, we can handle only medium-sized programs up to 100kLOC. Security kernels are usually not really big; still, a better scale-up is an issue for future work. Another well-known problem in Java is the API: even the smallest programs loads hundreds of library classes, which must be analyzed together with the client code. The bottleneck is always the points-to analysis, as precise points-to for Java is notoriously difficult. But there has been tremendous progress in scalability of program analyses such as points-to, and we will be able to exploit this to improve scalability and precision.

Another technical issue is compositionality: it must be possible to analyze isolated methods or classes and combine the results later – but as said, our analysis as well as today's PDG and points-to technology is a whole-program analysis and requires the complete program. Furthermore, analysis of multi-threaded programs, as sketched in Sect. 3.4, has not yet been integrated into our IFC.

From the information flow view, an important issue is modeling of declassification. Even IFC experts feel somewhat uncomfortable with the declassification approach, as it has an ad-hoc flavor. Uncontrolled declassification can introduce several problems, loss of monotonicity of release being one of them. Thus a lot of work on foundations of declassification is done in the IFC community (e.g. [35]). While we can guarantee monotonicity of release (albeit under kind of restricted context constraints), more research in declassification concepts and their relation to PDGs is needed.

Another approach is to use path conditions (as sketched in Sect. 2.3) in order to obtain more semantically convincing characterizations and context constraints for sound declassification. Our approach to declassification does currently not offer per-se checks of semantic properties as stipulated by [51], but will rely on path conditions to provide precise necessary conditions for a declassification to take place. This approach falls into the category "when" declassification may occur.

11 Conclusion

We presented a system for IFC in PDGs, integrating method calls and declassification without losing precision at call sites. Our approach is fully automatic, flow-sensitive, context-sensitive, and object-sensitive. Thus it is more precise than traditional IFC systems. In particular, unstructured control flow and exceptions are handled precisely.

The presented approach has been implemented inside the IDE Eclipse. The plugin allows definition of security lattices, automatic generation of SDG's, annotation of security levels to SDG nodes via source annotation and automatic security checks. We can handle full Java bytecode and can analyze medium-sized programs, which are typical in a security setting with restricted environments like JavaCard.

Our preliminary results indicate that the number of false alarms is reduced compared to type-based IFC systems, while of course all potential security leaks are discovered. Future case studies will apply our technique to a larger benchmark of IFC problems, and provide quantitative comparisons concerning performance and precision between our approach and other IFC systems.

In any case, PDG-based IFC is more expensive than type-based IFC. But a precise security analysis which costs minutes or even hours of CPU time is not too expensive compared to possible costs of illegal information flow or many false alarms.

Acknowledgements Jens Krinke contributed to previous versions of this work; we particularly thank Jens for contributions to the flow equations (as first described in [20, 21]), and for discussions on IFC. Frank Nodes implemented the Eclipse integration. The anonymous reviewers provided valuable comments on a preliminary version of this article.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 147–160. ACM, New York, NY, USA (1999). doi:10.1145/292540.292555
2. Agat, J.: Transforming out timing leaks. In: POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 40–53. ACM, New York, NY, USA (2000). doi:10.1145/325694.325702
3. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 91–102. ACM, New York, NY, USA (2006). doi:10.1145/1111037.1111046
4. Anderson, P., Reps, T., Teitelbaum, T.: Design and implementation of a fine-grained software inspection tool. IEEE Transactions on Software Engineering **29**(8) (2003). doi:10.1109/TSE.2003.1223646
5. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation, pp. 103–112. ACM Press, New York, NY, USA (2005). doi:10.1145/1040294.1040304
6. Bell, D.E., LaPadula, L.J.: Secure computer systems: A mathematical model, volume II. Journal of Computer Security **4**(2/3), 229–263 (1996). Based on MITRE Technical Report 2547, Volume II
7. Bergeretti, J.F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Trans. Program. Lang. Syst. **7**(1), 37–61 (1985). doi:10.1145/2363.2366
8. Biba, K.J.: Integrity considerations for secure computer systems. Tech. Rep. MTR-3153, The Mitre Corporation (1977). doi:100.2/ADA039324
9. Bieber, P., Cazin, J., Marouani, A.E., Girard, P., Lanet, J.L., Wiels, V., Zanon, G.: The PACAP prototype: a tool for detecting Java Card illegal flow. In: Proc. 1st International Workshop, Java Card 2000, *LNCS*, vol. 2041, pp. 25–37. Springer, Cannes, France (2000). doi:10.1007/3-540-45165-X_3
10. Binkley, D., Harman, M., Krinke, J.: Empirical study of optimization techniques for massive slicing. ACM Trans. Program. Lang. Syst. **30**(1), 3 (2007). doi:10.1145/1290520.1290523
11. Chambers, C., Pechtchanski, I., Sarkar, V., Serrano, M.J., Srinivasan, H.: Dependence analysis for Java. In: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, pp. 35–52. Springer-Verlag (1999). doi:10.1007/3-540-44905-1_3
12. Chess, B., McGraw, G.: Static analysis for security. IEEE Security and Privacy **2**(6), 76–79 (2004). doi:10.1109/MSP.2004.111
13. Clark, D., Hankin, C., Hunt, S.: Information flow for Algol-like languages. Computer Languages, Systems & Structures **28**(1), 3–28 (2002). doi:10.1016/S0096-0551(02)00006-1
14. Cohen, E.S.: Foundations of Secure Computation, chap. Information Transmission in Sequential Programs, pp. 297–335. Academic Press, Inc., Orlando, FL, USA (1978). Paper presented at a 3 day workshop held at Georgia Inst. of Technology, Atlanta, Oct. 1977
15. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987). doi:10.1145/24039.24041
16. Giffhorn, D., Hammer, C.: Precise analysis of Java programs using JOANA (tool demonstration). In: Proc. 8th IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 267–268 (2008). doi:10.1109/SCAM.2008.17
17. Giffhorn, D., Hammer, C.: Precise slicing of concurrent programs – an evaluation of precise slicing algorithms for concurrent programs. Journal of Automated Software Engineering **16**(2), 197–234 (2009). doi:10.1007/s10515-009-0048-x
18. Goguen, J.A., Meseguer, J.: Interference control and unwinding. In: Proc. Symposium on Security and Privacy, pp. 75–86. IEEE (1984). doi:10.1109/SP.1984.10019
19. Hammer, C.: Information flow control for Java. Ph.D. thesis, Universität Karlsruhe (TH) (2009). Forthcoming
20. Hammer, C., Krinke, J., Nodes, F.: Intransitive noninterference in dependence graphs. In: Proc. Second International Symposium on Leveraging Application of Formal Methods, Verification and Validation (ISoLA 2006), pp. 119–128. IEEE Computer Society, Washington, DC, USA (2006). doi:10.1109/ISoLA.2006.39
21. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: Proc. IEEE International Symposium on Secure Software Engineering (ISSSE'06), pp. 87–96 (2006)
22. Hammer, C., Schaade, R., Snelting, G.: Static path conditions for Java. In: PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security, pp. 57–66. ACM, New York, NY, USA (2008). doi:10.1145/1375696.1375704
23. Hammer, C., Snelting, G.: An improved slicer for Java. In: PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 17–22. ACM Press, New York, NY, USA (2004). doi:10.1145/996821.996830
24. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 226–238. ACM, New York, NY, USA (2009). doi:10.1145/1480881.1480911
25. Horwitz, S., Prins, J., Reps, T.: On the adequacy of program dependence graphs for representing programs. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 146–157. ACM, New York, NY, USA (1988). doi:10.1145/73560.73573
26. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990). doi:10.1145/77606.77608
27. Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 79–90. ACM Press, New York, NY, USA (2006). doi:10.1145/1111037.1111045
28. Jayaraman, G., Ranganath, V.P., Hatcliff, J.: Kaveri: Delivering the Indus Java program slicer to Eclipse. In: Proc. Fundamental Approaches to Software Engineering (FASE'05), *LNCS*, vol. 3442, pp. 269–272. Springer (2005). doi:10.1007/b107062
29. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica **7**(3), 305–317 (1977). doi:10.1007/BF00290339
30. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. **28**(4), 619–695 (2006). doi:10.1145/1146809.1146811

31. Krinke, J.: Context-sensitive slicing of concurrent programs. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 178–187. ACM, New York, NY, USA (2003). doi:10.1145/940071.940096
32. Krinke, J.: Program slicing. In: Handbook of Software Engineering and Knowledge Engineering, vol. 3: Recent Advances. World Scientific Publishing (2005)
33. Lhoták, O., Hendren, L.: Scaling Java points-to using Spark. In: Proc. 12th International Conference on Compiler Construction,, *LNCS*, vol. 2622, pp. 153–169 (2003). doi:10.1007/3-540-36579-6_12
34. Livshits, B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: Proceedings of the Usenix Security Symposium, pp. 271–286. Baltimore, Maryland (2005). URL http://portal.acm.org/citation.cfm?id=1251416
35. Mantel, H., Reinhard, A.: Controlling the what and where of declassification in language-based security. In: ESOP '07: Proceedings of the European Symposium on Programming, *LNCS*, vol. 4421, pp. 141–156. Springer (2007). doi:10.1007/978-3-540-71316-6
36. Mantel, H., Sands, D.: Controlled declassification based on intransitive noninterference. In: Proceedings of the 2nd Asian Symposium on Programming Languages and Systems, APLAS 2004, *LNCS*, vol. 3302, pp. 129–145. Springer, Taipei, Taiwan (2004). doi:10.1007/b102225
37. Myers, A.C., Chong, S., Nystrom, N., Zheng, L., Zdancewic, S.: Jif: Java information flow. URL http://www.cs.cornell.edu/jif/
38. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (2000). doi:10.1145/363516.363526
39. Nanda, M.G., Ramesh, S.: Interprocedural slicing of multithreaded programs with applications to Java. ACM Trans. Program. Lang. Syst. **28**(6), 1088–1144 (2006). doi:10.1145/1186632.1186636
40. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002). URL http://www4.informatik.tu-muenchen.de/~nipkow/LNCS2283/
41. Pistoia, M., Banerjee, A., Naumann, D.A.: Beyond stack inspection: A unified access-control and information-flow security model. In: SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy, pp. 149–163. IEEE Computer Society, Washington, DC, USA (2007). doi:10.1109/SP.2007.10
42. Pistoia, M., Chandra, S., Fink, S.J., Yahav, E.: A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM Syst. J. **46**(2), 265–288 (2007). doi:10.1147/sj.462.0265
43. Pistoia, M., Flynn, R.J., Koved, L., Sreedhar, V.C.: Interprocedural analysis for privileged code placement and tainted variable detection. In: Proceedings of the 9th European Conference on Object-Oriented Programming, *LNCS*, vol. 3586, pp. 362–386. Springer (2005). doi:10.1007/11531142_16
44. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**(5), 27 (2007). doi:10.1145/1275497.1275502
45. Reps, T., Horwitz, S., Sagiv, M., Rosay, G.: Speeding up slicing. In: SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, pp. 11–20. ACM, New York, NY, USA (1994). doi:10.1145/193173.195287
46. Reps, T., Yang, W.: The semantics of program slicing. Tech. Rep. 777, Computer Sciences Department, University of Wisconsin-Madison (1988). URL http://www.cs.wisc.edu/techreports/viewreport.php?report=777
47. Robschink, T., Snelting, G.: Efficient path conditions in dependence graphs. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, pp. 478–488. ACM Press, New York, NY, USA (2002). doi:10.1145/581339.581398
48. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pp. 43–55. ACM, New York, NY, USA (2001). doi:10.1145/504282.504286
49. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003). doi:10.1109/JSAC.2002.806121
50. Sabelfeld, A., Sands, D.: A PER model of secure information flow in sequential programs. Higher Order Symbol. Comput. **14**(1), 59–91 (2001). doi:10.1023/A:1011553200337
51. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations, pp. 255–269. IEEE Computer Society, Washington, DC, USA (2005). doi:10.1109/CSFW.2005.15
52. Scholz, B., Zhang, C., Cifuentes, C.: User-input dependence analysis via graph reachability. In: Proc. Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 25–34 (2008). doi:10.1109/SCAM.2008.22
53. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 355–364. ACM (1998). doi:10.1145/268946.268975
54. Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: SAS '96: Proceedings of the Third International Symposium on Static Analysis, pp. 332–348. Springer-Verlag, London, UK (1996). doi:10.1007/3-540-61739-6_51
55. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Trans. Softw. Eng. Methodol. **15**(4), 410–457 (2006). doi:10.1145/1178625.1178628
56. Volpano, D.M., Smith, G.: A type-based approach to program security. In: TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, *LNCS*, vol. 1214, pp. 607–621. Springer-Verlag, London, UK (1997). doi:10.1007/BFb0030629
57. Wasserrab, D., Lohner, D., Snelting, G.: On PDG-based noninterference and its modular proof. In: PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, pp. 31–44. ACM, New York, NY, USA (2009). doi:10.1145/1554339.1554345
58. Yokomori, R., Ohata, F., Takata, Y., Seki, H., Inoue, K.: An information-leak analysis system based on program slicing. Information and Software Technology **44**(15), 903–910 (2002). doi:10.1016/S0950-5849(02)00127-1