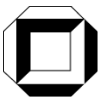
Universität Karlsruhe - Fakultät für Informatik - Bibliothek - Postfach 6980 - 76128 Karlsruhe

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

Christian Hammer, Gregor Snelting

Interner Bericht 2008-16



Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

ISSN 1432-7864

Fakultät für **Informatik**

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 not flow-sensitive, context-sensitive, or object-sensitive; 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 PDG generator for full Java 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 [25] and [24]

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. Hence 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 method; 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 [54], 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.

```
1 if (confidential==1)
2    public = 42
3 else
4    public = 17;
5 ... // no output of public
6 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 [58]. 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 [42], exist. Thus 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 Figure 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 following assignment, and thus declares the fragment to be untypeable.

Classical noninterference [20,21] however only demands that two streams of public output of the same program must be indistinguishable even if they differentiate on secret variables, which is true for this program. Thus secret data in a public variable is perfectly eligible as long as its content does not flow to output. Note that the killing statement may be far away from the supposed illegal flow.

Type-based IFC performs even worse in the presence of unstructured control flow or exceptions. Therefore, type systems overapproximate information flow control, 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 [30], or languages with no support for unstructured control flow [3].

[1] an established term in dataflow analysis

1.3 PDGs and Overview

Fortunately, program analysis has much more to offer than just sophisticated type systems. In particular, the *program dependence graph* (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 [65]. 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 [31,27]. 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 [59]. But more elaborate algorithms were needed to make the approach work and scale for full C and realistic programs [52,60]. The latter article contains a theorem connecting PDGs to the classical noninterference criterion (see also section 2). Later, Hammer et al. developed a precise PDG for full Java [27], which is much more difficult than C due to the effects of inheritance and dynamic dispatch, and due to the concurrency caused by thread programming. Krinke's slicing algorithm [33] 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 which propagate security levels through the program in details. We focus on precise interprocedural analysis with declassification in called methods, a feature that our previous work [25] could only handle in a conservative fashion. 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 on the value of the expression x (which is typically a condition in an if- or while-statement).

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 [28,51,48,11,49,64]. Thus PDGs are *correct*. Exploiting this fundamental property, all statements possibly influencing y (the so-called *backward slice*) are easily computed as

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

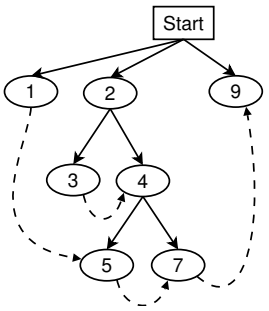


Fig. 2 A small program and its dependence graph

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

where y is called the *slicing criterion* of the backward slice. In particular, the Slicing Theorem [51] 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 Figure 2. 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 figure 1, the PDG contains edges $1 \to 2$ and $1 \to 4$, but *not* $(1) \to^* (6)$ 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 section 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 [62,34]; some techniques are explained in chapter 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 or termination 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 in [20, 21]. Every statement a has a security level $dom(a)$. Noninterference between two security levels, written as $d \not\rightsquigarrow e$ means that no statement with security level d may influence a statement of security level e. A system is thus considered secure according to the original Goguen/Meseguer noninterference criterion, if for all possible statement sequences x and all final statements a

$$output(run(z_0,x),a) = output(run(z_0,purge(x,dom(a))),a) \tag{2}$$

Without discussing details of this formula, we observe: noninterference requires that the final program output must be unchanged if every statement is deleted which – according to its security level – must not influence the final program state. We see that the notion of security is based on observational behavior and not on the source code.

Today, the notion of noninterference is usually defined in a more compact form. In its simplest variant – which assumes only security levels *Low* and *High* – it reads

$$s \cong_{Low} s' \implies [\![c]\!]s \cong_{Low} [\![c]\!]s' \tag{3}$$

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 based on PER relations [55].

The following theorem connects PDGs to the original Goguen/Meseguer definition and demonstrates how PDGs can be used to check for noninterference. Note that the same theorem applies to the *Low*-equivalence based noninterference definition, and we are preparing a machine-checked proof for the latter version [64].

Theorem 1 *If*

$$s \in BS(a) \implies dom(s) \rightsquigarrow dom(a) \tag{4}$$

then the noninterference criterion is satisfied for a.
Proof. *See [60].* □

Thus if $dom(s) \not\rightsquigarrow dom(a)$ (s and a have noninterfering security levels), there must be *no PDG path* $s \rightarrow^* a$, otherwise a security leak has been discovered.

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.

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 [56] is violated. This is not in discrepancy with Theorem 1, 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.

In addition, we proposed an even stronger mechanism on top of PDGs, called *path conditions* [59,52,60,26]. 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)
3     y = a[2*j-42];
```

Here, a necessary condition for a flow $x \rightarrow^* y$ is $\exists i, j.(i > 10) \wedge (i+3 = 2j-42) \equiv false$, 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 [60].

```
1  secret = 1;
2  public = 2;
3  s = f(secret);
4  x = f(public);
5  p = x;
```

Fig. 3 Example for context-sensitivity

3 PDGs for Java

In the following, we assume some familiarity with slicing technology, as presented for example in [62, 34]. 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 [29].[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 Figure 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 make SDG construction more difficult. In particular, in case a Java program creates objects, any precise program analysis such as PDG 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. [53, 35]. Without points-to analysis, all Java program analysis is useless, as objects and pointer assignments are so abundant.

[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 the sense of procedure dependence graphs: they represent that part of a SDG that corresponds to a single procedure or method.

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

Fig. 4 A Java password checker

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 per in- resp. 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 [27].

3.2 Exceptions

Figure 4 shows a fragment of a Java class for checking a password (taken from [42]) which uses fields and exceptions. The *P* and *R* annotations will be explained in section 4. The initial PDG for the `check` method is shown in Figure 5. Solid lines represent control dependence and dashed lines represent data dependence. 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

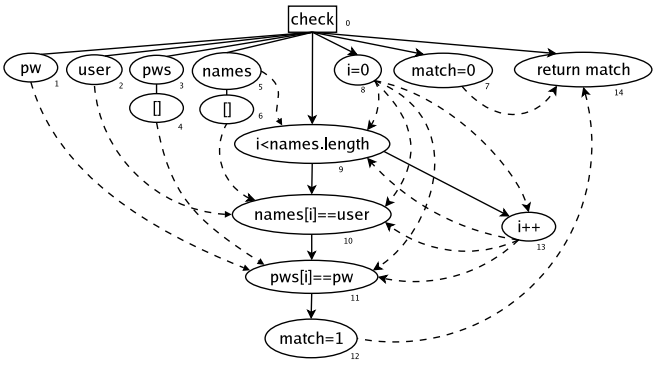


Fig. 5 PDG for `check` in Figure 4

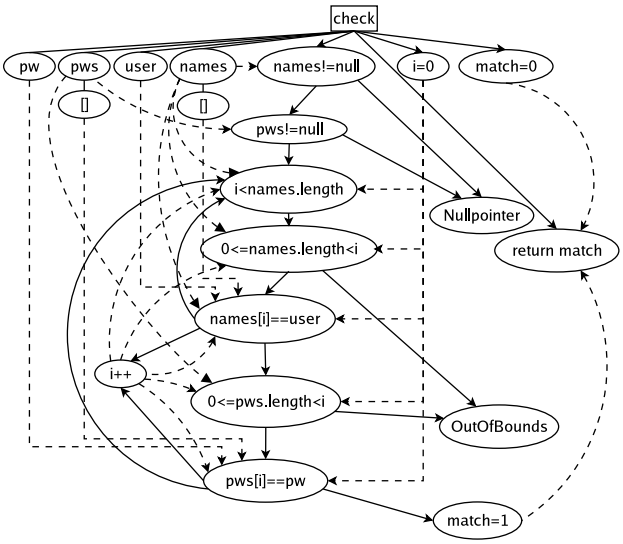


Fig. 6 PDG with exceptions for Figure 4

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 (9) and (11). All these nodes are immediate 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 [12], 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. The resulting final PDG is shown in Figure 6. (For better readability, the following examples will not show the effects of exceptions.)

3.3 Context-Sensitivity and Object-Sensitivity in Action

Figure 7 shows another small example program, and Figure 8 shows its SDG. For brevity we omitted the PDGs of the `set` and `get` methods. The effects of method calls are reflected by *summary edges* (shown as dashed edges in Figure 7) between actual-in and actual-out parameter nodes. Summary edges as introduced by Horwitz et al. [29] represent a transitive dependence between the corresponding formal-in and formal-out node pair. 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 in `set`. Summary edges enable context-sensitive slicing in SDGs in time linear to the number of nodes [29].

```
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 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`. Thus there is no SDG path (13) $\rightarrow^*$ (19) from the initialization of `sec` to the first print statement (i.e.

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

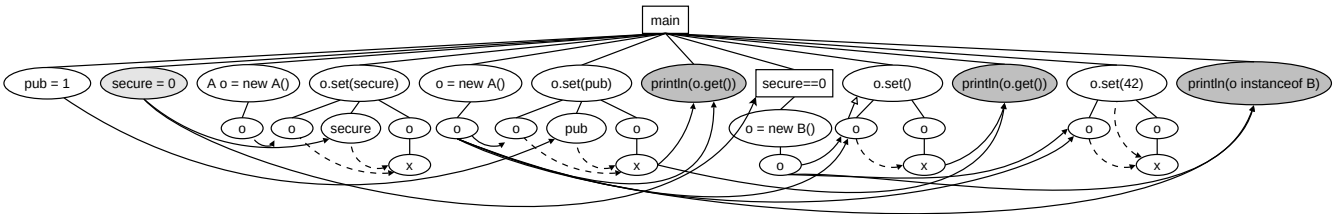


Fig. 8 SDG for the program in Figure 7

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), 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. (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 [33] was the first author to present a precise algorithm for slicing multi-threaded programs. Based on additional dependences between variables in different threads, Krinke's algorithm ensures that the sequence of statements in a PDG path does correspond 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 [43]. 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].

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 [21] utilize a *lattice* $\mathscr{L} = (L; \leq, \sqcup, \sqcap, \bot, \top)$ of security levels, the simplest consisting just of two security levels *High* and *Low*. 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 variables need such annotations.

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. via 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 and output of the system with a certain security level (e.g. assigned by the OS) can be controlled (resp. 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 resp. its PDG node x is written $S(x)$, where $S : N \rightarrow L$.[4] 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*

[4] 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.

are the predecessor and successor functions induced by $\rightarrow$, resp., this fundamental property is easily stated as

$$y \rightarrow x \in G \implies S(x) \geq S(y) \tag{5}$$

and thus by the definition of a supremum

$$S(x) \geq \bigsqcup_{y \in pred(x)} S(y) \tag{6}$$

This fundamental constraint ensures $S(y) \rightsquigarrow S(x)$.[5] Let us point out once more that it is sufficient to consider flow along PDG edges, as flow between unconnected PDG nodes is impossible.

Remember that confidentiality and integrity are dual to each other [8], hence the dual condition for integrity is

$$S(x) \leq \bigsqcap_{y \in pred(x)} S(y) \tag{7}$$

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

Equation (6) assumes that every statement resp. 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.[6] The *provided* security level specifies that a statement sends information with the provided security level, i.e. represents an input channel. The *required* security level requires that only information with a *smaller* or equal security level may reach that statement,[7] i.e. it represents an output channel of the specified security level. 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.

The actual security level $S(x)$ for a statement x must thus not only be greater than the levels of its predecessors, but also greater than its own provided security level. Thus equation (6) refines to

$$S(x) \geq \begin{cases} P(x) \sqcup \bigsqcup_{y \in pred(x)} S(y), & \text{if } x \in dom(P) \\ \bigsqcup_{y \in pred(x)} S(y), & \text{otherwise} \end{cases} \tag{8}$$

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:

$$\forall x \in \mathrm{dom}(R) : R(x) \geq S(x) \tag{9}$$

We can now formally define confidentiality:

Definition 1 Let a program's PDG be given. The program maintains confidentiality, if for all PDG nodes equations (8) and (9) are satisfied.

As mentioned earlier, we are preparing a machine-checked proof [64] that Definition 1 implies noninterference as defined in equation (3). 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 section 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 (8) and (9) 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 statements have a provided and required security level:

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

$$R'(x) = \begin{cases} R(x), & \text{if } x \in dom(R) \\ \top, & \text{otherwise} \end{cases} \tag{11}$$

Note that $\bot$ is the neutral element for $\sqcup$, and $\top$ is the neutral element for $\sqcap$. Now equation (8) simplifies to

$$S(x) \geq P'(x) \sqcup \bigsqcup_{y \in pred(x)} S(y) \tag{12}$$

and equation (9) simplifies to

$$R'(x) \geq S(x) \tag{13}$$

4.2 Solving Flow equations

Equation (12) is satisfied in the most precise way, and hence the risk that equation (9) is violated minimized, if the inequality for S turns into equality:

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

Of course (14) also satisfies (12), and can be read as an algorithm which computes $S(x)$ from $P(x)$ and x's predecessors S values. Thus equation (14) 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 equation (14) corresponds to a well-known concept in program analysis, namely a *monotone dataflow analysis framework* [32], which allows efficient fixpoint computation. Such frameworks start with a lattice of abstract values, which in our case is $\mathscr{L}$. For every

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

[6] For practicability of an analysis, it is important that the number of such annotations is as small as possible.

[7] The term "required" may be misleading here—it is actually more like a limit or maximum

$x \in N$, a so-called *transfer function* $f_x : L \to L$ must be defined, which typically has the form $f_x(l) = g_x \sqcup (l \sqcap \overline{k_x})$.[8] 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.[9] 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 equations (14) and thus (6) for all $x \in N$.

Thus the computed fixpoint for S, together with equation (9), 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 $l \not\geq S(x)$ we have a violation at x because $S(x) \not\rightsquigarrow l$ (the security level of $S(x)$ is not allowed to influence level l). Note that it is $\not\geq$ 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 explicit solution for the fixpoint can be given which will be exploited later:

Theorem 2 *For all $x \in N$, let $S(x)$ be the least fixpoint of equation (14). Then*

$$S(x) = \bigsqcup_{y \in BS(x)} P'(y) \tag{15}$$

Proof. Let $x \in N$. (14) 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 $y \to^* x \in BS(x)$: $P'(y) \leq S(x)$. By definition of a supremum, $S(x) \geq \bigsqcup_{y \in BS(x)} P'(y)$.

On the other hand, (15) is a solution of (14):

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

$$= P'(x) \sqcup \bigsqcup_{y \in pred(x)} \bigsqcup_{z \in BS(y)} P'(z)$$

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. □

[8] 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 $\overline{\bot} = \top$.

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

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 3 *If*

$$\forall a \in \mathrm{dom}(R) : \forall x \in BS(a) \cap \mathrm{dom}(P) : P(x) \leq R(a) \tag{16}$$

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 security level $P(x)$.

Proof. Let $x \in N$. We need to show that (8) and (9) 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 2. Hence (9) is satisfied. Furthermore, by definition of the fixpoint for S, S satisfies (14) and thus (12) and (8). □

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

PDG-Based Confidentiality Check. *For every node in the dependence graph that has a required security level specified, compute the backward slice, and check that all nodes in the slice have lower or equal provided security levels specified.*

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.

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* $CH(x,a) = FS(x) \cap BS(a)$.[10] 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 (Figure 5). We choose a three-level security lattice: *public*, *confidential*, and *secret* where

$$public \leq confidential \leq secret \tag{17}$$

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

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

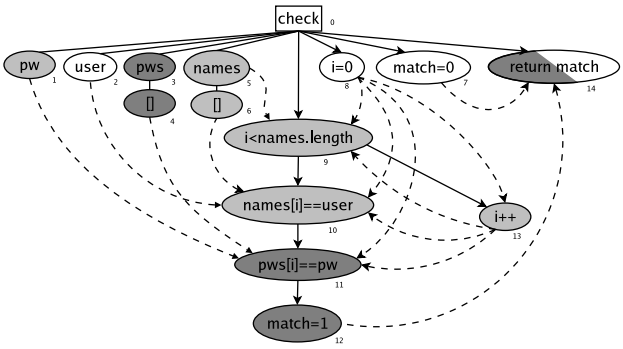


Fig. 9 PDG for Figure 4 with computed security levels

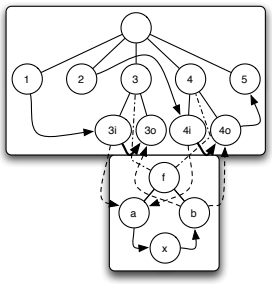


Fig. 10 SDG for Figure 3. The statement `x` computes the return value `b` from the formal input parameter `a`.

PDG has additional dependences for exceptions (see Figure 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 equation (14), are depicted in Figure 9 (white for *public*, light gray for *confidential*, and gray for *secret*[11]). 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 equation (9) 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.

5 Inter-procedural propagation of security levels

Let us now discuss *interprocedural IFC*. To understand the problem of context-sensitivity, consider again Figure 3 and its SDG in Figure 10. In this program fragment, $P(\texttt{secret}) = P(1) = High$, $P(\texttt{public}) = P(2) = Low$, and $R(p) = R(5) = Low$. 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 equation (14), $S(4o) = S(5) = S(p) = High$, and $R(p) \not\geq S(p)$.

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

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.[12] 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 [29, 50] 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 (14) and Theorem 3 still hold.[13]

Equations (14) and (9) 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.[14] 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.

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

[13] in fact they hold for the naive backward slice as well, because even the naive interprocedural backward slice is correct; it is just so imprecise.

[14] 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 := ∅
9    /* Phase 1: slice without descending into called procedures */
10   BS' ←MarkReachingVertices(G,{x},{parameter-out})
11   /* Phase 2: slice called procedures
12        without ascending into call sites */
13   BS ←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 ≠ ∅ do
27     select and remove node n from WorkList
28     M ∪= n
29     foreach w ∈ G such that w ∉ M and G contains an edge w → v
30              whose kind is not in Kinds do
31       WorkList ∪= w
32       if v ∉ D then
33         C ∪= {"S(w) ≤ S(v)"} // cf. eq. (14) or (18)
34         if v ∈ dom(R) then
35           C ∪= {"S(v) ≤ R(v)"} // cf. eq. (9) or (18)
36         if w ∈ dom(P) then
37           C ∪= {"P(w) ≤ S(w)"} // cf. eq. (14) or (19)
38       else
39         C ∪= {"P(v) ≤ S(v)"} // cf. eq. (21)
40         C ∪= {"S(w) ≤ R(v)"} // cf. eq. (22)
41       fi
42     od
43   od
44   return M
45 end
```

2. In the second phase, starting from the 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 equations (8) and (9). We will show later that a solution to these constraints enforces confidentiality.

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 procedure. In the first phase, no security level is propagated into called procedures and 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.

Definition 2 Let a program's SDG be given. The program maintains confidentiality, if for every $a \in dom(R)$ and its HRB backward slice $BS(a)$, equations (8) and (9) are satisfied.

Again we postpone the proof that this definition implies noninterference (equation (3)), but point out that the definition – as Definition 1 –is solidly based on SDG correctness properties and fundamental definitions for confidentiality.

The latter are expressed in equations (8) and (9). Restricting these equations to the (context-sensitive) backward slices of all points $\in dom(R)$ avoids spurious flow, and is sufficient as these slices contain all nodes affecting equation (9). Thus Theorem 3 is still valid in the interprocedural case, and the proof remains the same.[15] 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 alarm it generates.

Let us now argue that Algorithm 1 is correct.

Theorem 4 *For every $a \in dom(R)$, Algorithm 1 (where $D = \emptyset$) 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 $w \rightarrow v \in BS(a)$, the algorithm generates a constraint $S(w) \leq S(v)$, which is necessary according to equation (14) 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 equation (14) 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 equation (9) 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.

[15] In fact we need a version of theorem 2 working on SDGs and backward slices; which is left as an exercise to the reader.

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

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

Thus Algorithm 1 generates exactly the constraints required by (9), and constraints exactly equivalent to (8). Hence 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 section 4.3.

For the example above (Figure 3/Figure 10), Algorithm 1 computes $BS(5) = \{5, 4o, 4i, 4\}$ in the first phase and adds $\{b, x, a\}$ in the second phase, thus avoiding to add $3i$ 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 Figure 11 (right column; again results based on naive slicing are shown in gray). The precise solution correctly computes $S(1) = High$, and indeed $P(1) = High \leq S(1)$. The naive solution would compute $S(1) = Low$ and generates a false alarm due to $P(1) \not\leq S(1)$.

5.2 Backward Flow Equations

Note that equation (14) 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 equations (14) and (9) into an equivalent form which mirrors this backward propagation, while Theorem 3 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: equation (6) demands that for every $x \in N$ and $y \in pred(x)$, $S(x) \geq S(y)$ and thus $S(x) \geq \bigsqcup_{y \in pred(x)} S(y)$. The same set of constraints can be expressed as follows: for every $x \in N$ and $y \in succ(x)$, $S(x) \leq S(y)$ (equation (12)), and as a consequence,

$$S(x) \leq R'(x) \sqcap \prod_{y \in succ(x)} S(y) \tag{18}$$

In analogy, for equation (9) ($a \in \mathrm{dom}(R)$, $S(a) \leq R(a)$), one gets:

$$\forall a \in \mathrm{dom}(P) : P(a) \leq S(a) \tag{19}$$

Theorem 5 *For the same PDG resp. (intraprocedural) slice, the collected instances of equations* (12) *and* (9) *generate the same set of constraints as the collected instances of equations* (18) *and* (19).

Proof (for full details see [22]). 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 equations (18) and (19). 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 to lower

the security level of incoming information 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 section 4) is appropriate – a password-based authentication mechanism necessarily reveals some information about the secret password.[16]

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* can be declassified [56]. 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:

$$x \in D \implies \big(x \in dom(P) \cap dom(R)\big) \wedge \big(R(x) \geq P(x)\big) \quad (20)$$

Information reaching x with a maximal security level $R(x)$ is lowered (declassified) down to $P(x)$ (note that $R(x) \not\geq P(x)$ does not make any sense, as declassification should lower a level, not heighten it). Now a path from node y to a with $P(y) > R(a)$ 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 [56], this policy for expressing intentional information release is describing *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.

[16] We all know that password crackers can exploit this approach in case weak passwords are used, hence just adding a declassification seems too naive. Additional techniques to protect the table are needed.

In terms of the propagation equations, a declassification simply changes the computation of S. Equation (12) must be extended as follows:

$$S(x) \geq \begin{cases} P(x) & \text{if } x \in D \\ P'(x) \sqcup \bigsqcup\limits_{y \in pred(x)} S(y) & \text{otherwise} \end{cases} \quad (21)$$

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

Of course, equation (9) is still valid for non-declassification nodes, but for $x \in D$ it must be modified as $S(x)$ is the declassified value:

$$\forall x \in dom(R) \setminus D : R(x) \geq S(x) \;\wedge\; \forall x \in D : R(x) \geq \bigsqcup_{y \in pred(x)} S(y) \quad (22)$$

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. The program maintains confidentiality, if for all $a \in dom(R)$ equations (21) and (22) are satisfied.

Theorem 6 *For every $a \in dom(R)$, Algorithm 1 (where $D \neq \emptyset$) 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 4) that for non-declassification nodes the generated constraints correspond exactly to equations (9) and (8), and thus to the non-declassification cases in equations (21) and (22). For declassification nodes $d \in D$, Algorithm 1 does no longer generate constraints $S(w) \leq S(d)$, which is indeed required by (21), 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 (22), case $x \in D$. Furthermore, it generates $S(d) \geq P(d)$ which is exactly required by (21), case $x \in D$.

Thus Algorithm 1 generates exactly the constraints required by (21) and (22). Hence they have the same fixpoint, and fulfill the requirements of Definition 3. □

In case $D = \emptyset$, Algorithm 1 by theorem 4 checks noninterference without declassification. Thus we obtain for free the

Corollary 1 (Conservativity of Declassification) *Algorithm 1 is conservative, that is, without declassification reduces to 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, and $P(d_2) \leq P(d_1)$, as otherwise the second declassification is redundant.

In case there are several declassifications on disjoint paths from x to y, for example $x \rightarrow^* d_1 \rightarrow^* y$, $x \rightarrow^* d_2 \rightarrow^* y$, $x \rightarrow^* d_3 \rightarrow^* y$, ..., 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 will be the source for an more precise interprocedural IFC, as described in section 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 satisfy confidentiality according to Definition 3 and let $d \in N$ where $d \notin D \cup dom(R) \cup dom(P)$. Replace d by $d' \in D$ where d' has the same connecting edges as d, but d' is annotated with R and P. Recompute the actual security levels according to (21), yielding $S'(x)$ for $x \in N$. Declassification d' respects monotonicity of release, if equation (22) still holds for all $S'(x)$.

Theorem 7 *If* $R(d') \geq \bigsqcup_{y \in pred(d')} S(y)$, *and* $P(d') \leq \bigsqcup_{y \in pred(d')} S(y)$, *then for* $x \in N$, $S'(x) \leq S(x)$.

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 $R(d')$. Note that $P(d') \leq R(d')$ is required anyway in equation (20). The above premise is more precise and avoids that a declassification generates new illegal flows as the outgoing declassification level is too high, or the incoming limit too low. In practice, both requirements are easy to check and do not restrict sensible declassification.

Proof. In the original PDG, $\bigsqcup_{y \in pred(d')} S(y) \leq S(d') \leq \bigsqcap_{y \in succ(d')} S(y)$, hence in the new PDG $S'(d') = P(d') \leq S(d') = \bigsqcup_{y \in pred(d')} S(y)$ by assumption and equation 21. Furthermore, $S'(d) \leq S(d') \leq \bigsqcap_{y \in succ(d')} S(y) \leq S(y)$ for all $y \in succ(d')$. Hence $S(y) \geq \bigsqcap_{z \in pred(y)} S(y) \geq S'(y) = S'(d') \sqcup \bigsqcup_{z \neq d' \in pred(y)} S(z)$. The same argument works for the successors of y. By induction[17] $S'(x) \leq S(x)$ follows for all x. □

Corollary 2 *Under the assumptions of theorem 7, declassification* d' *respects monotonicity of release.*

Proof. For the original PDG, (9) and (21) are valid for S. In the new PDG, (21) is by construction valid for S', and (9) is valid for S' since by the theorem $S'(x) \leq S(x)$. □

[17] 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

6.3 Confidentiality check with declassification

The original PDG-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:

$$\forall x \in dom(P) \cap BS(a) : P(x) \leq S(x) \tag{23}$$

Theorem 6 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, equation (21) is violated and an illegal flow has been detected. As described in section 5.1, the S values are computed first, and the criterion (23) is checked in a second phase; this allows to generate diagnostics by computing chops.

Let us return to the example in Figures 4 and 9 and assume $R(14) = public$. As described in section 4.3, the analysis reveals an illegal flow $3 \rightarrow^* 14$. 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, since only a small amount of information leaks from password checking.

As another example consider Figure 12. In line 3 a declassification $D : confidential \rightarrow public$ 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 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.

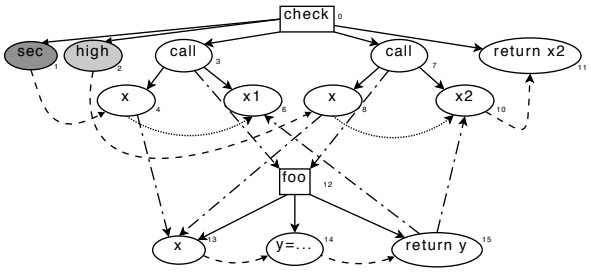


Fig. 13 System Dependence Graph for Figure 12

The SDG for this program is shown in Figure 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.

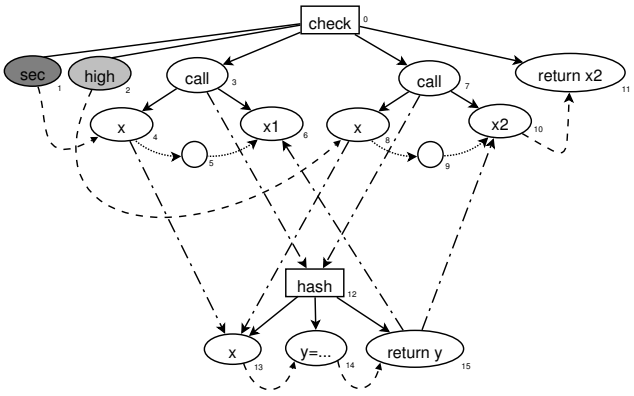


Fig. 14 SDG for Figure 12 with summary declassification nodes

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, but 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 flew 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 Figure 13 again: The required security level for node 11 is *Low* as specified. Algorithm 1 computes $S(2) = S(8) = S(10) = S(11) = \mathit{Low}$ 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 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 $x \to y$, 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 $d \in D$ 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 throw the procedure body is also a legal flow through $x \to d \to y$. 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 Figure 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 ($R(15) = \mathit{confidential}$, $P(15) = \mathit{public}$), so for the path between node 13 and 15 the summary declassification nodes 5 and 9 will be set to $R(5) = R(9) = \mathit{confidential}$ 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) $\mathit{confidential} = P(8) \leq S(8) \leq S(8) \leq R(9) = \mathit{confidential}$ and $\bot = P(9) \leq S(9) \leq S(10) \leq R(11) = \mathit{public}$, thus no security violation is found in `check`. In the second slicing phase, there is no violation either: $S(13) \leq S(14) \leq R(15) = \mathit{confidential}$ and $\mathit{public} = P(15) \leq S(10) \leq R(11) = \mathit{public}$. 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 $R(5) = R(9) = \mathit{confidential}$ 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)

```
1  procedure SummaryDeclassification(G,L,R)
2  input G: a system dependence graph
3    L: a security lattice
4    R: the required annotations
5  output: the set of summary declassification nodes (included in G)
6  begin
7    pathEdges = ∅ // set of transitive dependences already seen
8    foreach formal−out node o in G do
9      pathEdges∪= (o,o,¬(o ∈ D),⊤)
10   od
11   workList := pathEdges
12   while workList not empty do
13     remove (x,y,f,l) from workList
14     if x is an formal−in node then
15       addSummaries(x,y,f,l)
16     else
17       foreach edge w → x ∈ G do
18         addToPathEdges(extendPathEdge((x,y,f,l),w))
19       od
20     fi
21   od
22 end
23
24 procedure addToPathEdges(x,y,f,l)
25 input (x,y,f,l): a path edge tuple
26 begin
27   if (x,y,f′,l′) ∈ pathEdges where f′ ≠ f or l′ ≠ l then
28       remove (x,y,f′,l′) from pathEdges
29   fi
30   if (x,y,f,l) ∉ pathEdges then
31     pathEdges∪= (x,y,f,l)
32     workList∪= (x,y,f,l)
33   fi
34 end
```

Algorithm 3 Auxiliary procedures for Summary Declassification Nodes

```
35 procedure extendPathEdge((x,y,f,l),w)
36 input: (x,y,f,l): a path edge tuple
37   w the extension node
38 output: a path edge extended by w
39 begin
40   if pathEdges contains a tuple (w,y,f′,l′) then
41     retrieve (w,y,f′,l′) from pathEdges
42   else
43     l′ = ⊤, f′ = false
44   fi
45   if x ∈ D then
46     l′ = l′ ⊓ R(x)
47   else
48     l′ = l′ ⊓ l
49   fi
50   f′ = f′ ∨ (w ∉ D ∧ f)
51   return (w,y,f′,l′)
52 end
53
54 procedure addSummaries(x,y,pf,l)
55 input: (x,y,f,l): a path edge tuple
56 begin
57   foreach actual parameter pair (v,w) corresponding to (x,y)
58     if f then
59       add summary edge v →sum w to G
60       n := v
61     else
62       add summary declassification node d and edges v →sum d
63         and d →sum w where R(d) = l and P(d) = ⊥
64     fi
65     foreach (w,z,f′,l′) ∈ pathEdges do
66       addToPathEdges(extendPathEdge((w,z,f′,l′),d))
67     od
68   od
69 end
```

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

As summary declassification nodes represent the effect of declassifications on paths in the procedure body, and these can in turn call procedures (even recursively), a simple transitive closure between formal parameters does not yield a correct solution for summary declassification nodes [29,50]. Instead, a specialized algorithm for summary edges must be leveraged [50] where the computation of the security levels for summary declassification nodes can be integrated. The result can be seen in Algorithms 2 and 3, which incorporate a backward IFC propagation into the algorithm described by Reps et al. [50].

As an example, consider Figure 12 again. Here, 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 $\sqcap$, resp. $\vee$. 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 8 *IFC with Algorithm 1 and summary declassification nodes determined according to Algorithms 2 and 3 is sound and precise.*

Proof (for full details see [22]). 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 [29, 50], 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 59. 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 $w \rightarrow y$), 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 $w \rightarrow v$ 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 [18]. At the time of this writing, the Java slicer and security levels are fully operational.

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 [22]. 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 9835 lines long. The PDG (including necessary API parts) consists of 135271 nodes and 1002589 edges. The time for PDG construction was 145 seconds plus 742 seconds for generation of summary edges.

Next, 30332 backward slices were selected by choosing a random node as a starting point. The average slice size is 86023 nodes, which is about 68% 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 [26, 25, 52].

As a case study for IFC we chose another JavaCard applet called `Wallet`[18]. It is only 252 lines long but with the necessary API parts and stubs the PDG consists of 18858 nodes and 68259 edges. The time for PDG 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 statement that update the balance with the provided security level *High* and inserted a declassification to *Low* into the `getBalance` method. The methods `credit` and `debit` may throw an exception if the maximum balance would be exceeded or if there is insufficient credit, resp. 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.

9 Related Work

9.1 PDGs and IFC

Several papers have been written about PDGs and slicers for Java, but to our knowledge only the Indus slicer [31] 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 [25], 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

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

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 4 from Section 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. [66] were probably the first to propose and implement an IFC analysis based on program slicing for a procedural language. 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.

Hammer et al. combined static and dynamic PDG analysis for detection of illegal information flow [23]. It allows the a-posteriori analysis of programs showing unexpected behavior and the computation of an exact witness for reconstruction of the illegal information flow.

9.2 Security type systems

Volpano and Smith [63] 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 proof noninterference in case the system reports no typing errors. An extension to multi-threaded languages is given in [58].

Myers [41] defines JFlow, an extension of the Java language with a type system for information flow. The JIF compiler [42] implements this language. We already discussed in Section 2 that type systems are less precise, but are more efficient. JIF supports generic classes and the decentralized label model [41]; 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, 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.

Strecker [61] formulated a non-deterministic type system including the noninterference proof in Isabelle [44]. It handles major concepts of MicroJava such as classes, fields and method calls, but omits arrays and exceptions.

Mantel and Reinhard [39] defined the first type system for a multi-threaded while language that controls the what and the where dimension simultaneously. The type system is based on a definition for the where dimension that supersedes their previous definition of intransitive noninterference [40], 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 Verification and IFC

Amtoft et al. [3] present an interprocedural flow-sensitive Hoare-like logic for information flow control in a rudimentary object-oriented language. Casts, type tests, visibility modifiers other than public, and exception handling are not yet considered. Only structured control flow is allowed.

The Pacap case study [9] verifies secure interaction of multiple JavaCard applets on one smart card. They employ model checking to ensure a sufficient condition for their security policy, which is based on a lattice similar to noninterference without declassification. Implicit exceptions are modeled, but such unstructured control flow may lead to *label creep* (cf. [54, Sect. II E]).

Genaim [16] defines an abstract interpretation of the CFG looking for information leaks. It can handle all bytecode instructions of single-threaded Java and conservatively handles implicit exceptions of bytecode instructions. The analysis is flow- and context-sensitive but does not differentiate fields of different objects. Instead, they propose an object-insensitive solution folding all fields of a given class. In our experience [27] object-insensitivity yields too many spurious dependences. The same is true for the approximation of the call graph by CHA. In this setting, both will result in many false alarms.

An area uncovered by our system is security policies, defining under which circumstances declassification is allowed. Li and Zdancewic [36] define a framework for downgrading policies for a core language with conditionals and fixed-points, yielding a formalized security guarantee with a program equivalence proof.

9.4 Static analysis for security

Static analysis is often used for source code security analysis [13]. For example, information flow control is closely related to tainted variable analysis. There are even approaches like the one from Pistoia et al. [47] that use slicing for taint analysis or the one from Livshits and Lam [38, 37] that uses IPSSA, a representation very similar to dependence graphs. However, these analyses only use a trivial security level (tainted/untainted) with a trivial declassification (untaint) and could greatly benefit from our approach. Scholz et al. [57] 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.

Pistoia et al. [46] 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 [45]. 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 [22]. 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.

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. [39]). 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. One could, for example, disallow multiple declassifications on one PDG path or chop, and we suspect that this will simplify matters considerably without damaging practicality.

Another approach is to use path conditions (as sketched in section 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 [56], but will rely on path conditions to provide precise necessary conditions for a declassification to take place. This approach falls into the category "how" declassification may occur, which has not yet been extensively researched.

Let us finally mention that we have used PDGs not only to check for classical noninterference, but to implement possibilistic or probabilistic noninterference. For details, see [19].

11 Conclusion

We presented a system for information flow control 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 We thank Jens Krinke, who contributed to previous versions of this work, for ongoing discussions on IFC; and Frank Nodes for implementing the Eclipse integration.

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., G.Zanon: 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. Binkley, D., Horwitz, S., Reps, T.: The multi-procedure equivalence theorem. Tech. Rep. 890, Computer Sciences Department, University of Wisconsin-Madison (1989). URL http://www.cs.wisc.edu/techreports/viewreport.php?report=890
12. 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
13. Chess, B., McGraw, G.: Static analysis for security. IEEE Security and Privacy **2**(6), 76–79 (2004). DOI 10.1109/MSP.2004.111
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. Genaim, S., Spoto, F.: Information flow analysis for java bytecode. In: 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005), *LNCS*, vol. 3385, pp. 346–362. Springer, Paris, France (2005). DOI 10.1007/b105073
17. Giffhorn, D., Hammer, C.: An evaluation of precise slicing algorithms for concurrent programs. In: SCAM'07: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 17–26. Paris, France (2007). DOI 10.1109/SCAM.2007.9
18. 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
19. Giffhorn, D., Lochbihler, A.: Information flow control for concurrent programs via program slicing. Submitted for publication
20. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proc. Symposium on Security and Privacy, pp. 11–20. IEEE (1982). DOI 10.1109/SP.1982.10014
21. 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
22. Hammer, C.: Information flow control for java. Ph.D. thesis, Universität Karlsruhe (TH) (2009). Forthcoming
23. Hammer, C., Grimme, M., Krinke, J.: Dynamic path conditions in dependence graphs. In: PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 58–67. ACM Press, New York, NY, USA (2006). DOI 10.1145/1111542.1111552
24. 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
25. 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)
26. 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
27. 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
28. 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
29. 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
30. 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
31. 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
32. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica **7**(3), 305–317 (1977). DOI 10.1007/BF00290339
33. 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
34. Krinke, J.: Program slicing. In: Handbook of Software Engineering and Knowledge Engineering, vol. 3: Recent Advances. World Scientific Publishing (2005)
35. 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
36. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 158–170. ACM Press, New York, NY, USA (2005). DOI 10.1145/1040305.1040319
37. 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
38. Livshits, V.B., Lam, M.S.: Tracking pointers with path and context sensitivity for bug detection in c 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. 317–326. ACM, New York, NY, USA (2003). DOI 10.1145/940071.940114

39. 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
40. 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
41. Myers, A.C.: JFlow: practical mostly-static information flow control. In: POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 228–241. ACM Press, New York, NY, USA (1999). DOI 10.1145/292540.292561
42. Myers, A.C., Chong, S., Nystrom, N., Zheng, L., Zdancewic, S.: Jif: Java information flow. URL http://www.cs.cornell.edu/jif/
43. 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
44. 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/
45. 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
46. 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
47. 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
48. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Softw. Eng. **16**(9), 965–979 (1990). DOI 10.1109/32.58784
49. 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
50. 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
51. 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
52. 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
53. 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
54. 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
55. 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
56. 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
57. 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
58. 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
59. 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
60. 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
61. Strecker, M.: Formal analysis of an information flow type system for MicroJava (extended version). Tech. rep., Technische Universität München (2003)
62. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3**(3), 121–189 (1995)
63. 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
64. Wasserrab, D.: Towards certified slicing. In: G. Klein, T. Nipkow, L. Paulson (eds.) The Archive of Formal Proofs (2008). URL http://afp.sf.net/entries/Slicing.shtml. Formal proof development
65. Weiser, M.: Program slicing. IEEE Trans. Software Eng. **10**(4), 352–357 (1984)
66. 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