Intransitive Noninterference in Dependence Graphs

Christian Hammer
University of Passau, Germany
hammer@fmi.uni-passau.de

Jens Krinke
FernUniversität in Hagen, Germany
krinke@acm.org

Frank Nodes
University of Passau, Germany

***Abstract*— In classic information flow control (IFC), *noninterference* guarantees that no information flows from secret input channels to public output channels. However, this notion turned out to be overly restrictive as many intuitively secure programs do allow some release. In this paper we define a static analysis that allows intransitive noninterference in combination with context-sensitive analysis for Java bytecode programs. In contrast to type systems that annotate variables, our approach annotates information sources and sinks. To the best of our knowledge this is the first IFC technique which is flow-, context-, and object-sensitive. It allows IFC for realistic languages like Java or C and offers a mechanism for *declassification* to accommodate some information leakage for cases where traditional noninterference is too restrictive.**

I. 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*).

State-of-the-art IFC exploits *program analysis* to assign and propagate security levels to variables and expressions, guaranteeing that any potential security leak is found. *Language-Based* IFC [1] utilizes the program source code alone to discover security leaks. This has the huge advantage that it can exploit a long history of research on program analysis, and will discover any security leaks caused by software itself, though this approach may miss information flow through e.g. physical side channels, which are usually handled by separate approaches.

In their recent overview article, Sabelfeld and Myers [1] survey contemporary IFC approaches based on program analysis. Most contemporary analysis methods are based on non-standard type systems. Security levels are coded as types for variables (and fields in object-oriented software) and the typing rules catch illegal flow of information [2], [3].

Most of the type-based approaches partition the set of variables (and fields in object-oriented software) into disjoint sets (such as *secure* and *public* data). If information illicitly flows e.g. from the *secure* to the *public* partition, then the program is already considered insecure, even if the public variable is no longer live after that flow. The original intention was, however, different: Classic noninterference states that the two streams of (public) output of the same program should be indistinguishable even if they differentiate on (secret) input, or in other words *secure* input is not allowed to flow to *public* output channels. So we really should not worry about secret data *in* a variable as long as its content is not fed to output.

```
1 if (confidential==1)
2    public = 42
3 else
4    public = 17;
5 public = 0;
```

Fig. 1. A secure program fragment

Besides, type-based analysis is usually not flow-sensitive, context-sensitive, nor object-sensitive. This leads to imprecision and thus to a high number of 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 information flow from `confidential` to `public` in the if-statement is guaranteed to be killed by the following assignment. Type-based IFC performs even worse in the presence of unstructured control flow or exceptions. Therefore, type systems greatly 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 [4], or languages with no support for unstructured control flow [5].

Fortunately, program analysis has much more to offer than just sophisticated type systems. In particular, the *program dependence graph* (PDG) [6] has become a standard data structure allowing various kinds of powerful program analyses and in particular, efficient program slicing [7]. The first information flow control algorithm based on PDGs for full C was presented by Snelting et al. [8]. Recently, a theorem connecting PDGs to the classical noninterference criterion was proven [9]. Later, Hammer et al. developed a precise PDG for full Java [10], which is much more difficult than C due to the effects of inheritance and dynamic dispatch. Today, we can handle realistic C and Java programs and thus have a powerful tool for IFC available that is much more precise than conventional approaches.

In this paper, we augment PDGs with Denning-Style security level lattices. We focus mainly on the precise interprocedural analysis with declassification in called methods, a feature that our previous work [11] could only handle in a conservative

fashion. Finally, we present several examples and performance data, and compare the approach to type-based IFC systems such as Jif [3].

II. Dependence Graphs and Noninterference

Program dependence graphs are a standard tool to model information flow through a program. Program statements or expressions are the graph nodes. A data dependence edge $x \to y$ means that statement x assigns a variable which is used in statement y (without being reassigned 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).

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. In particular, all statements influencing y (the so-called *backward slice*) are easily computed as

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

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

Noninterference was introduced in [12]. 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. The following theorem demonstrates how PDGs can be used to check for noninterference.

Theorem. If

$$s \in BS(a) \implies dom(s) \rightsquigarrow dom(a) \tag{1}$$

then the noninterference criterion is satisfied for a.

Proof. See [9]

Thus if $dom(s) \not\rightsquigarrow dom(a)$ (s and a have noninterfering security levels), there must be *no PDG path* $s \to^* a$, 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. Applying the theorem results in a linear-time noninterference test for a, as all $s \in BS(a)$ must be traversed once. However, as we will see later, it is not possible to use declassification in a purely slicing based approach.

But note that even the PDG is a conservative approximation; due to imprecision of the underlying program analysis algorithms it may contain too many edges (but never too few). In any case, PDGs are much more precise than type-based systems. The example from the introduction in Figure 1 does *not* have a PDG path (1) $\to^*$ (5) and thus is considered safe; no false alarm is generated.

On the other hand, it must be noted that any slicing based security analysis is sensitive to semantic preserving transformations: Usually, any slicing approach does not eliminate statements like "`if (h > h) then l = 0`" and will assume a transitive dependence from `h` to `l`. Thus, semantic consistency as postulated by Sabelfeld and Sands [13] is hard to achieve.

In the following, we assume some familiarity with slicing technology, as presented for example in [14], [15]; we will not discuss technical details here. Note that the computation of precise dependence graphs and slices is still a complex problem.

Intraprocedural PDGs can easily be constructed for method bodies, using the well-known algorithms from literature. Interprocedural slicing, however, is more tricky. The standard analysis relies on *system dependence graphs* (SDGs), which include dependences for calls as well as transitive dependences between parameters [16]. SDGs are *context-sensitive*, that is, different calls to the same procedure or method are indeed distinguished; avoiding spurious dependences. 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.

Figure 2 shows a small Java class for checking a password (taken from [3]). The PDG for the `check` method can be seen in Figure 3. Solid lines represent control dependence and dashed lines represent data dependence (ignore the shade of the nodes for now). Node 0 is the method entry with its parameters in nodes 1 and 2 (we use pw and pws as a shorthand for password and passwords). Nodes 3 – 6 represent the fields of the class, note that because the fields are arrays, the reference and the elements are distinguished. Nodes 7 and 8 represent the initializations of the local variables `match` and `i` in lines 6 and 8. 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).

While SDGs in general are well understood dynamic dispatch and objects as method parameters make SDG construction more difficult, and the use of points-to analysis alone is not sufficient. Treatment of dynamic dispatch is well known: possible targets of method calls are approximated statically (in our case using points-to information [17], [18]), and for all possible target methods the standard interprocedural SDG construction is done.

Method parameters are another 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 object reference is passed to the method. However, field values stored in actual parameter objects may be changed 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. To improve precision, we made the analysis object-sensitive by representing nested parameter objects as trees. Unfolding object trees stops once a fixed point with respect to the aliasing situation of the containing object is reached.

```
1  class PasswordFile {
2     private String[] names;     /*P:confidential*/
3     private String[] passwords; /*P:secret*/
4     public boolean check(String user,
5                          String password /*P:confidential*/) {
6       boolean match = false;
7       for (int i=0; i<names.length; i++) {
8         if (names[i]==user
9             && passwords[i]==password) {
10          match = true;
11          break;
12        }
13      }
14      return match;  /*R:public*/
15    }
16 }
```

Fig. 2. A Java password checker

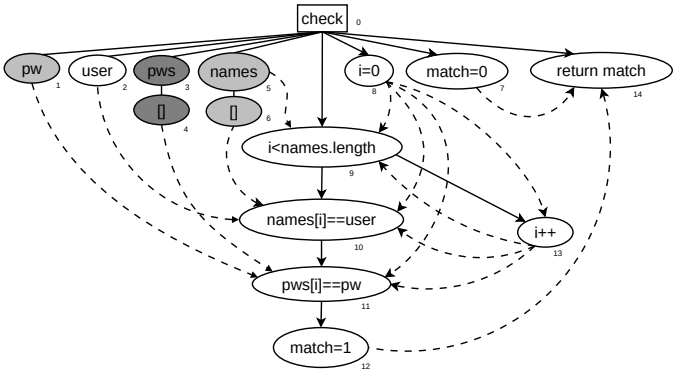

Fig. 3. PDG for check in Figure 2

Dynamic runtime exceptions can alter the control flow of a program an 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 [19], 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.

Figure 4 shows another small example program, which serves to illustrate the effects of dynamic dispatch and object-sensitivity. We will explain the details of security levels in the next section, right here we use two security levels *Low* < *High*. The `main` method contains two variables where `secure` is annotated with *High* and the other with *Low*. Thus both

```
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. 4. Another Java program

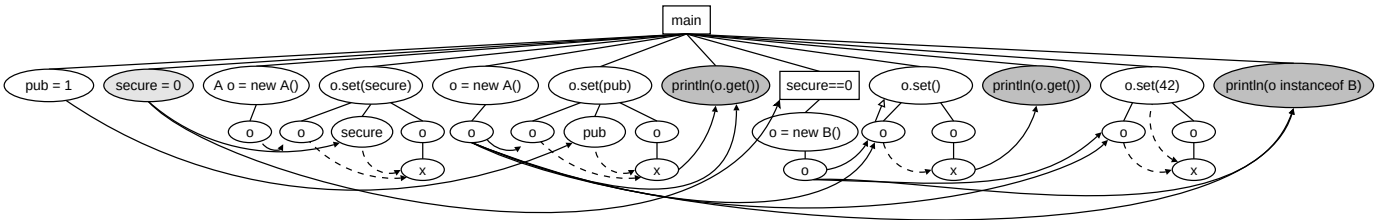

Fig. 5. SDG for the program in Figure 4

statements provide a security level and are underlined in the source code. Secure data must not affect visible output of a program. Hence the arguments to `System.out.println` require that any node on any path to the output node has a security level not exceeding a given level, in our case *Low*. Statements that require a given security level are underwaved.

Figure 5 shows the SDG for this example. 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 4) between actual-in and actual-out parameter nodes. Summary edges as introduced by Horwitz et al. [16] represent a transitive dependence between the corresponding formal-in and formal-out node pair. For example, the call to `o.set(secure)` contains two summary edges, one from the target object `o` and one from `secure` to the field `x` of `o`; representing the side-effect that the value of `secure` 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.

First, a new `A` object is created where field `x` is initialized to `secure`. However, this object is no longer used afterward as the variable is overwritten with a new object whose `x` field is set to `pub`, the variable annotated with *Low*. Thus the value of `x` in `o` does no longer contain *High* information, so (19) is a perfectly legal statement. Information flow control based on slicing can detect this fact: In Figure 4 there exists no path (15) $\rightarrow^*$ (20) from the initialization of `secure` to the first print statement (i.e. the leftmost `println` node). Instead, we have a path from the initialization of `pub` to this output node. The fact that we did not generate a false alarm here stems from the object-sensitivity of our SDG based on points-to data, flow-sensitivity of SDGs, and from context-sensitivity of backward slicing with summary edges.

The next statements show an illegal flow of information: Line (21) checks whether `secure` 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 implicit flow: `if (secure==0 && ...) o.x = 0 else o.x = 1;` In the PDG we have a path from `secure` to the predicate testing `secure` to `o.set()` and its target object `o`. Following the summary edge one reaches the `x` field and finally the second output node. This path is a witness for the illegal flow. Our analysis thus rejects this program because it prints out the *High* value of `o.x` in (24).

But even if the value of `x` was not dependent on `secure` (after statement 26) an attacker could exploit the runtime type of `o` to gain information about the value of `secure` (27). This implicit information flow is detected by our analysis as well, since there is a PDG path (15) $\rightarrow^*$ (28).

In order to compare our approach with type-based IFC, we analyzed the program from Figure 4 using Jif [3]. Jif uses a generalization of Denning's lattices, the so-called decentralized label model. It allows to specify sets of security levels (called "labels" or "principals") for every statement, and to attach a set of operations to any label. This is written e.g. $\{o_1 : r_1, r_2; o2 : r_2; r_3\}$ and thus is slightly more general. Our approach could easily be generalized to use the decentralized label model as well.

We adapted the first part of Figure 4 to Jif syntax and annotated the declaration of `o` and both instantiations of `A` with the principal {`pp:`}. The output statement was replaced by an equivalent code that allowed public output. Jif reports that secure data could flow to that public channel and thus raised a false alarm. Thus even decentralized labels can not overcome the impreciseness of type-based analysis.

III. Security levels and declassification

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 utilize a *lattice* $\mathcal{L} = (L, \sqcup, \sqcap)$ of security levels, the simplest consisting just of two security levels *High* and *Low*. We provide a specification option for the lattice, and an option to mark some (or all) statements with their security level. The security level of statement with PDG node x is written $S(x)$, and confidentiality requires that an information receiver must have at least the security level of any sender. In PDGs, this implies

$$\forall y \in pred(x) : S(x) \geq S(y)$$

which ensures $S(y) \rightsquigarrow S(x)$. The dual condition for integrity is

$$\forall y \in pred(x) : S(x) \leq S(y)$$

However, this assumes that every statement resp. node has a security level specified, which is not realistic. We want to specify *provided* as well as *required* security levels not for all statements, but for certain selected statements only. The provided security level specifies that a statement sends information with the provided security level and the required security level specifies that only information with a *smaller*

security level may reach that statement[1]. The provided security levels are defined by a partial function $P : N \to L$, where N is the set of nodes resp. statements of the programs. Thus, $l = P(s)$ specifies the statement's security level. The required security levels are defined similarly as a partial function $R : N \to L$. Thus, $P(s)$ specifies the security level of the information generated at s and $R(s)$ specifies the maximal allowed security level of the information reaching s. In analogy to Theorem 1 from Section II, information with security level l that is generated at some node x in the dependence graph, is propagated along the dependences and should not reach another node a which has a required security level which is smaller than l. Thus a program represented as a dependence graph does not violate confidentiality, iff

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

i.e. 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 $P(x)$.

Usually, the number of nodes that have a specified security level is low, e.g. points of output. Therefore, the above criterion can easily be transformed into an algorithm that checks a program for confidentiality:

SDG-Based Confidentiality Check. *For every node in the dependence graph that has a required security level specified, compute the backward slice, and check that no node in the slice has a higher provided security level specified.*

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 ($P(x) > R(a)$), the responsible nodes can be computed by a chop $CH(x, a)$. The chop computes all nodes that are part of path from node x to node a, thus it contains all nodes that may be involved in the propagation from x's security level to a.

As an example, consider the PDG for the password program (Figure 3) again. We choose a three-level security lattice: *public*, *confidential*, and *secret* where *public* $\rightsquigarrow$ *confidential* $\rightsquigarrow$ *secret*. The list of passwords is *secret*, thus $P(3) = P(4) =$ *secret*. The list of names and the parameter `password` is *confidential*, because they should never be visible to a user. Thus, $P(1) = P(5) = P(6) =$ *confidential*. Figure 3 shows the annotated PDG: The security levels are depicted through white for *public*, light gray for *confidential*, and gray for *secret*.

According to the criterion, we require that no confidential or secret information flows out of the method, thus we require the return statement to have a required security level of *public* ($R(14) =$ *public*). A backward slice for node 14 will reveal that nodes 1 and 3–6 are included in the slice and have a higher security level, thus a security violation is revealed.

A. Declassification

In practice the above approach is too simple because in some situations one might accept that information with a higher security level flows to a "lower" channel. A typical example is the password checking method presented earlier: The result of the method will eventually be used to access the user's private data or to output an error message that the login was not successful. Of course, that areas will not have the same security level (*secret*) as the list of passwords. *Declassification* allows to lower the security level of incoming information at specified points. In the example, declassification would reduce the security level at the return node 14 to security level *public* such that the result of the password check can be used in low security areas.

[1]The term required may be misleading here—it is actually more like a limit

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

Fig. 6. Example

We model declassification by specifying certain PDG nodes to be declassification nodes: Let D be the set of declassification nodes. A declassification node $x \in D$ has to have a required and a provided security level: $r = R(x)$ and $p = P(x)$. Information reaching x with a maximal security level r is lowered (declassified) down to p. Now a path from node y to a with $P(y) > R(a)$ is not a violation, if there is a declassification node x 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).

According to Sabelfeld and Sands [13], our 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.

The above slicing solution no longer works with declassification, as information flow with declassification is no longer transitive and slicing is based on transitive information flow. A simple solution is to represent declassification nodes as barriers where slicing stops [20] and restart the computation from the declassification nodes. However, this would not only loose some context-sensitivity, but is also too restrictive. Consider the example in Figure 6: Assuming a security lattice *low* < *high* < *secret* the above approach will detect the following security violation: Line 3 requires a maximum level of *high* but may be reached with *secret* from line 10. A close examination shows that this is overly restrictive: The

information with level *secret* from line 10 may indeed reach the declassification in line 3, but the information in `x1` is never used afterward and does not reach an output channel (i.e. a node with any required security level).

This can easily be solved by applying the basic Theorem 1. First, we compute the backward slice for every node a with a required security level specified. By definition of a slice, no node outside this slice can have any influence on a. The subsequent analysis (see below) will only consider nodes and edges that are part of this slice, i.e. the dependence graph is reduced to the subgraph represented by the initial slice for a. This avoids spurious dependencies and false alarms caused by potential security violations outside the backward slice of a, as in the last example: `x1` is not part of the backward slice for line 12, thus avoiding the spurious security violation at line 3. Note that we exploit context-sensitive slicing here.

The analysis will then compute the actual required security level for every node in the (sliced) program dependence graph by a backward analysis. The actual required security level of a node is the maximal security level that may reach the node without causing a security violation at the criterion node under observation.

The actual incoming (required) security level $S_{\mathrm{IN}}(x)$ for a statement x is computed from the outgoing security levels of its successors $y \in succ(x)$:

$$S_{\mathrm{IN}}(x) = \begin{cases} \top & \text{if } succ(x) = \emptyset \\ \bigsqcap_{y \in succ(x)} S_{\mathrm{OUT}}(y) & \text{otherwise} \end{cases}$$

At nodes without declassification, the outgoing security level is simply the incoming security level: $S_{\mathrm{OUT}}(x) = S_{\mathrm{IN}}(x)$. At declassification nodes $x \in D$ with a declassification from $R(x)$ down to $P(x)$, S is replaced with the new required level: $S_{\mathrm{OUT}}(x) = R(x)$. Thus

$$S_{\mathrm{OUT}}(x) = \begin{cases} R(x) & \text{if } x \in D \\ S_{\mathrm{IN}}(x) & \text{otherwise} \end{cases}$$

These equations are data flow analysis equations and can be iteratively solved using a standard algorithm, with a proper initialization of S_{OUT}. The initialization is done based on the criterion node a under observation, i.e. it is checked that no security violation occurs due to $R(a)$:

$$S_{\mathrm{OUT}}(x) = \begin{cases} R(a) & \text{if } x = a \\ R(x) & \text{if } x \in D \\ \top & \text{otherwise} \end{cases}$$

Note that $\top$ is the neutral element (for $\sqcap$). Due to the monotonicity of the computation and the limited height of the security level lattice, a minimal fixed point for S_{IN} is guaranteed to exist and can be computed using a standard iteration. The computed S_{IN} have then to be checked for confidentiality:

Confidentiality Check With Declassification. *For every node a in the dependence graph that has a required security level specified which is not a declassification node, compute the incoming security levels $S_{\mathrm{IN}}(x)$ of all statements x in its backward slice and check the following property:*

$$\forall x \in dom(P) \cap BS(a) : P(x) \leq S_{\mathrm{IN}}(x) \qquad (2)$$

Thus, for any $l = P(x)$ such that $l \not\leq S_{\mathrm{IN}}(x)$ we have a confidentiality violation at x because $l \not\rightsquigarrow S_{\mathrm{IN}}(x)$ (the security level l is not allowed to influence the required level of $S_{\mathrm{IN}}(x)$. Note that it is $\not\leq$ and not $>$ because l and $S_{\mathrm{IN}}(x)$ might not be comparable. Because the slice $BS(a)$ has been computed first, the confidentiality check can be done during the dataflow analysis: If the current node has a provided security level $P(x)$, a computed required level $S_{\mathrm{IN}} \not\geq P(x)$ is a violation. Declassification nodes themselves are not considered as information sinks in the above check, even though they have to have a required security level. Otherwise, Sabelfeld's and Sand's *monotonicity of release* principle would be violated.

It is easy to show that the above above check is equivalent to noninterference according to Theorem 1 for programs without declassification and thus, the presented policy is a conservative extension of noninterference [13]. Moreover, as the presence of declassifications cannot be observed by an attacker, the presented policy obeys the *monotonicity of release* principle. It also avoids occlusion and is safe against trailing attacks.

Let us return to the example in Figure 3 and assume $R(14) =$ *public*. The computation to check confidentiality for criterion node 14 will start with a backward slice $BS(14)$. Because node 14 can be reached from every node in the PDG, the slice will contain the complete PDG, thus it is not reduced. The subsequent computation of the actual required security levels will result in $S_{\mathrm{IN}}(x) =$ *public* for all nodes x of the example. The confidentiality check will reveal violations at nodes 1 and 3–6 because for these nodes, the specified provided level is higher than the computed actual required.

Now assume the node 14 is a declassification node *secret* $\rightarrow$ *public*: $14 \in D$, $R(14) =$ *secret*, $P(14) =$ *public*. The computation of the actual required security levels will result in $S_{\mathrm{IN}}(x) =$ *secret* for $x < 14$. The confidentiality check will no longer reveal a security violation, which may be desirable depending on the security policy, since only a negligible amount of information leaks from password checking.

B. Interprocedural Analysis with Declassification

The above criterion can handle declassification even in the interprocedural case, but a plain extension would lose context-sensitivity and thus precision. Context-sensitivity can be achieved by computing the S_{IN} in a two-phase mode similar to context-sensitive slicing. Therefore, let us shortly review standard interprocedural slicing, before explaining the improved criterion.

Context-sensitive slicing relies on an explicit handling of interprocedural edges. Besides summary edges, there are parameter-in and -out edges, which connect the actual-in with the formal-in and the formal-out with the actual-out nodes, call edges connect the call site with the entry node of the method. Summary edges represent transitive information flow between an actual parameter (called actual-in node) and a variable receiving a return value or the effects of a side-effect (called actual-out node). As an example, consider Figure 7, which shows the system dependence graph for Figure 6: Node 2 has a transitive influence on the node 11 along the summary

edge from node 8 to 10. Slicing with summary edges is done in two phases:

- In the first phase all edges (including the summary edges) are traversed except parameter-out edges connecting e.g. the result node with the caller's result variable.
- In the second phase one starts with the omitted edges and traverses all edges except parameter-in and call edges (connecting the passed parameters, and the call node to the callee's entry node); i.e. one never comes back to the caller.

Without this two phase approach, the traversal would no longer be context-sensitive: In that case, a path $1 \to 4 \to 13 \to 14 \to 15 \to 10 \to 11$ would propagate the actual required security from node 11 (*low*) to node 1, where the confidentiality check would generate a false alarm.

Indeed, the successor function in the definition of S_{IN} can be adapted to support a similar two-phased approach. Still, this approach does not compute the most precise solution (although it is correct), because the effects of declassification in called procedures are ignored. The problem with summary edges is that they represent a transitive information flow between pairs of parameters, but declassification is intransitive. Using them for computation of the actual security level $S_{\text{IN}}(x)$ 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 7 again: The required security level for node 11 is *low* as specified. In the first phase $S_{\text{IN}}(2) = S_{\text{IN}}(8) = S_{\text{IN}}(10) = S_{\text{OUT}}(11) = \mathit{low}$ due to the summary edge. This will result in a false alarm because the declassification at node 15 is ignored.

Thus using summary edges for information flow control will not generate unsound results, but ignore possible declassification possibilities and therefore produce too many false alarms, as declassifications along transitive paths between procedure parameters are ignored.

To achieve maximal precision, we additionally assume that every summary edge from x to y, representing that there exists at least one path 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. For example, there is a (first half) summary edge from x to the declassification node d and a (second half) summary edge from d to y. This new declassification node d represents the declassification effects on all paths from x' to y' that are visible at x'. Therefore they are called *summarizing declassification nodes*. The provided security level of such node is $P(d) = \bot$, thus the confidentiality check (Equation 2) will never fail. If there is a declassification free path from x' to y', $R(d)$ will be set to $R(d) = \top$ such that the declassification node d will not be the source of a confidentiality violation.

If there is no declassification free path from x' to y', $R(d)$ will represent the maximal security level l that is allowed to reach x' such that there will be no security violation on a path

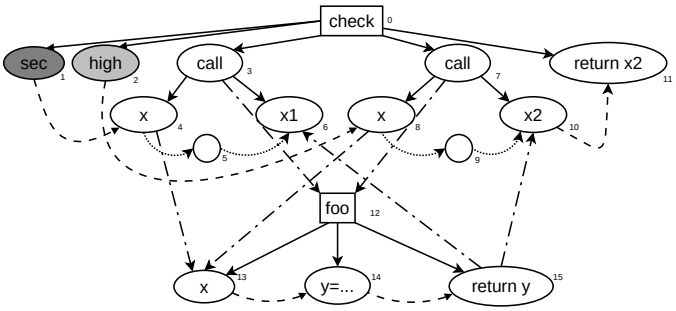

Fig. 7. SDG for Figure 6 with summarizing declassification nodes

from x' to a declassification node z ($l \leq R(z)$, $z \in D$) which is a prefix of a path from x' to y'.

Figure 7 shows a SDG with summarizing declassification nodes for the example in Figure 6. 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 via the declassification nodes 5 and 9 to the actual-out nodes 6 and 10.

1) Computation of $R(d)$ for Summarizing Declassification Nodes: Lets assume that node d is the node between actual-in x and actual-out y. To compute $R(d)$, it is necessary to propagate all $R(z)$ that belong to declassification nodes $z \in D$ on a path from x' to y'. This is done with the data flow equations and fixed-point iteration shown in Section III-A, where $S_{\text{OUT}}(y')$ is set to $\top$ and all nodes and edges not belonging to a path from x' to y' are ignored. The resulting $S_{\text{IN}}(x')$ is the computed value for $R(d)$. However, the computation of $S_{\text{IN}}(x')$ may depend on another summarizing declassification node e, for which $R(e)$ has not yet been computed. Moreover, $R(d)$ and $R(e)$ may depend on each other because of recursion. Again, a proper fixed-point iteration is needed: All summarizing declassification nodes $R(n)$ are initialized to $R(n) = \top$ and the $R(n)$ are computed iteratively until a fixed-point is reached. Because of the sound initialization (without fixed-point iteration the results of the confidentiality checks will be sound but produce many false alarms), the fixed-point iteration will not result in an unsound solution.

In Figure 7 the computation is only needed for the declassification at node 15 ($R(15) = \mathit{high}$, $P(15) = \mathit{low}$) and for the path between node 13 and 15. The computation will result in $S_{\text{IN}}(14) = \mathit{high}$ and $S_{\text{IN}}(13) = \mathit{high}$, thus the summarizing declassification nodes 5 and 9 will be set to $R(5) = \mathit{high}$ and $R(9) = \mathit{high}$ ($P(5) = P(9) = \bot$).

2) The interprocedural IFC check: With summarizing declassification nodes it is possible to compute the actual security levels inside a procedure or method based on data flow equations as presented in Section III-A. The summary edges with the summarizing declassification nodes will include the

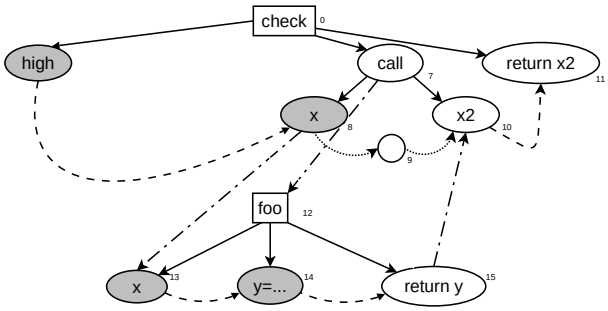

Fig. 8. Reduced dependence graph with computed security levels

declassification effects of called procedures. The last step that is needed is to propagate the actual security levels to called and calling procedures. Again, we have to make sure to stay context sensitive. Therefore, the two phase approach from backward slicing is used, where the computation and propagation of actual security levels is done in two phases:

1) The first phase ignores edges from the called procedure to the call site (parameter-out edges). Thus, it will only propagate the computed actual required security levels from called procedures to the call site, ascending the call chain. Due to the summary edges with the summarizing declassification nodes which are not ignored, the security levels at the call site will be propagated as if they were propagated through the called procedure.
2) In the second phase, the computed actual required security levels are propagated into called procedures. This is achieved by ignoring edges from the the call site to the called procedure (parameter-in and call edges). Again, summary edges are used.

The summary edges with summarizing declassification nodes have an essential effect, because they ensure that security levels are propagated 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.

Of course, the checking for security violations (Equation 2) is done at last after the second phase.

Lets return to the example in Figure 6 with its SDG in Figure 7, where we want to check if the required security level at line 12 cannot be reached with a higher security level. Thus, node 11 is the criterion node. The analysis starts with computing the backward slice $BS(11)$ and removes nodes 1 and 3–6 from the SDG.

The initialization of the subsequent fixed-point iteration sets $S_{\text{OUT}}(11) = R(11) = \mathit{low}$, $S_{\text{OUT}}(15) = R(15) = \mathit{high}$ due to the declassification node 15, and $S_{\text{OUT}}(9) = R(9) = \mathit{high}$ due to the summarizing declassification, and all other S_{OUT} to $\top$. The first phase will compute the fixed-point with the following values: $S_{\text{IN}}(0) = S_{\text{IN}}(7) = S_{\text{IN}}(9) = S_{\text{IN}}(10) = \mathit{low}$, $S_{\text{IN}}(2) = S_{\text{IN}}(8) = S_{\text{IN}}(12) = S_{\text{IN}}(13) = S_{\text{IN}}(14) = \mathit{high}$ and $S_{\text{IN}}(11) = S_{\text{IN}}(15) = \top$. Note that during the first phase the edges $8 \to 13$ and $7 \to 12$ are considered but edge $15 \to 10$ is ignored.

The second phase will propagate the actual security levels further into the called procedures which is in this example just `hash`. Because the parameter-out edge $15 \to 10$ is now allowed to be traversed (but $8 \to 13$ and $7 \to 12$ not), the second phase will compute $S_{\text{IN}}(15) = \mathit{low}$. The other S_{IN} will not change during this phase. Figure 8 shows the dependence graph with the computed S_{IN}, where light gray means *low* and gray *high*.

The last step of checking to confidentiality will not reveal a security violation, because $P(2) \leq S_{\text{IN}}(2)$ and $P(15) \leq S_{\text{IN}}(15)$ ($P(9) \leq S_{\text{IN}}(9)$, even though summarizing declassification nodes need not to be checked).

IV. Related Work

Several papers have been written about PDGs and slicers for Java, but to our knowledge only the Indus slicer [21] is—besides ours—fully implemented and can handle full Java. Indus is customizable, embedded into Eclipse, and has a very nice GUI.

The work described in this paper improves our previous algorithm [11], which was not able to handle declassification in called procedures precisely. However, that work also describes the generation and use of path conditions for Java PDGs (i.e. necessary conditions for an information flow between two nodes), which can uncover the precise circumstances under which a security violation can occur.

We already mentioned the overview article by Sabelfeld and Myers [1], which surveys language-based IFC methods. The focus is on type-based approaches; dependences and slicing are mentioned, but the authors obviously do not consider them a realistic option for IFC. This is amazing, since PDGs have been around for years.

Abadi et al. [22] were the first to connect slicing and noninterference, but only for λ-calculus. It is amazing that our Theorem 1 from Section II (which holds for imperative languages and their PDGs) was not discovered earlier. Only Anderson et al. [23] 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.

Myers [24] defines JFlow, an extension of the Java language with a type system for information flow. The JIF compiler [3] implements this language. We already discussed in Section II that his approach is less precise, but it is more efficient and supports generic classes and the decentralized label model; labels and principals are first class objects.

Barthe and Rezk [25] 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.

Amtoft et al. [5] 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 [26] 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. [1, Sect. II E]).

Genaim [27] 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 [10] 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.

A combined static and dynamic approach for detection of illegal information flow was presented recently [28]. 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.

An area uncovered by our system is security policies, defining under which circumstances declassification is allowed. Li and Zdancewic [29] 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.

Static analysis is often used for source code security analysis [30]. Information flow control is closely related to tainted variable analysis. There are even approaches like the one from Pistoia et al. [31] that use slicing for such kind analysis or the one from Livshits and Lam [32], [33] 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.

V. 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 much more precise than traditional, type-based IFC systems. In particular, unstructured control flow and exceptions cannot lead to false alarms. 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 the full Java bytecode and can analyze medium-sized programs, which are typical in a security setting with restricted environments like JavaCard; still, a better scale-up is an issue. Another well-known problem when this approach is to be extended to full Java is the API, which loads hundreds of library classes even for the smallest programs, with lots of native code, most of which have influence on information flow and must therefore be modeled in a conservative manner. A version for C is planned.

Our preliminary results indicate that the number of false alarms is drastically 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.

Thus PDG-based IFC is much 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 consequences of illegal information flow or myriad false alarms to be resolved manually.

Acknowledgment. Gregor Snelting provided valuable remarks.

References

[1] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, January 2003.

[2] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *Proceedings of the Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, CA, January 1998, pp. 355–364.

[3] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic, "Jif: Java information flow," http://www.cornell.edu/jif, 1999.

[4] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 2006, pp. 79–90.

[5] T. Amtoft, S. Bandhakavi, and A. Banerjee, "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*. New York, NY, USA: ACM Press, 2006, pp. 91–102.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, July 1987.

[7] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, July 1984.

[8] T. Robschink and G. Snelting, "Efficient path conditions in dependence graphs," in *Proceedings International ACM/IEEE Conference on Software Engineering (ICSE'02)*, Orlando, FL, May 2002, pp. 478–488.

[9] G. Snelting, T. Robschink, and J. Krinke, "Efficient path conditions in dependence graphs for software safety analysis," *ACM Transactions on Software Engineering and Methodology*, 2007, to appear.

[10] C. Hammer and G. Snelting, "An improved slicer for java," in *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM Press, 2004, pp. 17–22.

[11] C. Hammer, J. Krinke, and G. Snelting, "Information flow control for java based on path conditions in dependence graphs," in *Proc. IEEE International Symposium on Secure Software Engineering (ISSSE'06)*, Mar. 2006.

[12] J. Goguen and J. Meseguer, "Interference control and unwinding," in *Proc. Symposium on Security and Privacy*. IEEE, 1984, pp. 75–86.

[13] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, 2005, pp. 255–269.

[14] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, Sept. 1995.

[15] J. Krinke, "Program slicing," in *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing, 2005, vol. 3: Recent Advances.

[16] S. B. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[17] A. Rountev, A. Milanova, and B. G. Ryder, "Points-to analysis for Java using annotated constraints," in *Proc. 16th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, 2001, pp. 43–55.

[18] O. Lhotak and L. Hendren, "Scaling Java points-to using Sparc," in *Compiler Construction, 12th International Conference*, ser. LNCS, 2003, pp. 153–169.

[19] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan, "Dependence analysis for java," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 1999, pp. 35–52.

[20] J. Krinke, "Slicing, chopping, and path conditions with barriers," *Software Quality Journal*, vol. 12, no. 4, 2004.

[21] J. H. Ganeshan Jayaraman, Venkatesh Prasad Ranganath, "Kaveri: Delivering the indus java program slicer to eclipse," in *Fundamental Approaches to Software Engineering: 8th International Conference*, ser. LNCS, vol. 3442, 2005, pp. 269–272.

[22] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *Proc. Symposium on Principles of Programming Languages (POPL'99)*. ACM, 1999, pp. 147–160.

[23] P. Anderson, T. Reps, and T. Teitelbaum, "Design and implementation of a fine-grained software inspection tool," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, aug 2003.

[24] A. C. Myers, "Jflow: practical mostly-static information flow control," in *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1999, pp. 228–241.

[25] G. Barthe and T. Rezk, "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*. New York, NY, USA: ACM Press, 2005, pp. 103–112.

[26] P. Bieber, J. Cazin, A. E. Marouani, P. Girard, J.-L. Lanet, V. Wiels, and G.Zanon, "The PACAP prototype: a tool for detecting Java Card illegal flow," in *Java Card Forum*, Cannes, France, Sept. 2000.

[27] S. Genaim and F. Spoto, "Information flow analysis for java bytecode," in *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, ser. LNCS, vol. 3385. Paris, France: Springer, Jan. 2005, pp. 346–362.

[28] C. Hammer, M. Grimme, and J. Krinke, "Dynamic path conditions in dependence graphs," in *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, Jan 2006, pp. 58–67.

[29] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 2005, pp. 158–170.

[30] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, November/December 2004.

[31] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar, "Interprocedural analysis for privileged code placement and tainted variable detection," in *ECOOP 2005 – 19th European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, vol. 3586, 2005.

[32] V. B. Livshits and M. S. Lam, "Tracking pointers with path and context sensitivity for bug detection in C programs," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003, pp. 317–326.

[33] ——, "Finding security vulnerabilities in java applications with static analysis," in *USENIX Security Symposium*, 2005.