Information Flow Control for Java Based on Path Conditions in Dependence Graphs

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

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

Gregor Snelting
University of Passau,
Passau, Germany.
snelting@fmi.uni-passau.de

Abstract

Language-based information flow control (IFC) is a powerful tool to discover security leaks in software. Most current IFC approaches are however based on non-standard type systems. Type-based IFC is elegant, but not precise and can lead to false alarms.

We present a more precise approach to IFC which exploits active research in static program analysis. Our IFC approach is based on path conditions in program dependence graphs (PDGs). PDGs are a sophisticated and powerful analysis device, and today can handle realistic programs in full C or Java. We first recapitulate a theorem connecting the classical notion of noninterference to PDGs.

We then introduce path conditions in Java PDGs. Path conditions are necessary conditions for information flow; today path conditions can be generated and solved for realistic programs. We show how path conditions can produce witnesses *for security leaks.*

The approach has been implemented for full Java and augmented with classical security level lattices. Examples and case studies demonstrate the feasibility and power of the method.

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*).

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 [23] 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 [23] 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, and the typing rules catch illegal flow of information [18, 24]. Type systems can handle sequential as well as concurrent programs, and can even be used to discover timing leaks [2].

However, 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

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

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.

Fortunately, program analysis has much more to offer than just sophisticated type systems. In particular, the *program dependence graph* (PDG) has become, after lively research in adaptations for real world languages, a standard data structure allowing various kinds of powerful program

*This research was supported by Deutsche Forschungsgemeinschaft (DFG grant Sn11/9-1).

analyses. Recently, we proved a theorem connecting PDGs to the classical noninterference criterion [26].

Still, the PDG can only decide whether there is a potential information flow from a to b, or whether this is definitely not the case. The PDG does not provide detailed insight into the circumstances of a flow. And even PDGs sometimes generate false alarms.

We therefore proposed to base IFC on a combination of *dependence graphs* and *constraint solving*. While Snelting published the original idea already in 1996 [25], more elaborate algorithms were needed to make the approach work and scale for full C and realistic programs [21, 26].

Later, Hammer et al. developed a precise PDG for full Java [11], 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 PDG algorithm [14] for multi-threaded programs has recently been integrated. Today, we can handle realistic Java programs and thus have a powerful tool for IFC available that is much more precise than conventional approaches.

In this paper, we augment Java PDGs with Denning-Style security level lattices and introduce path conditions for Java PDGs. We present a new technique to generate path conditions for dynamic dispatch. Path conditions can (sometimes) be solved for input variables, such solved conditions act as a witness for illegal flow: it will become visible when the program is fed with the witness. We present several examples and performance data, and compare the approach to type-based IFC systems such as Jif [18].

2 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 $x \rightarrow y$ means that statement x assigns a variable which is used in statement y (without being reassigned underway). A control dependence edge $x \rightarrow y$ 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 $x \rightarrow^* y$ 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

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

For the small C program and its dependence graph in figure 1, there is a path from statement 1 to statement 9, indicating that input variable `a` will eventually influence output variable `z`. Since there is no path (1) $\rightarrow^*$ (4), there is definitely no influence from `a` to `x`.

The power of PDGs (compared to e.g. type systems) stems from the fact that they are *flow sensitive*: the order

```
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;
```

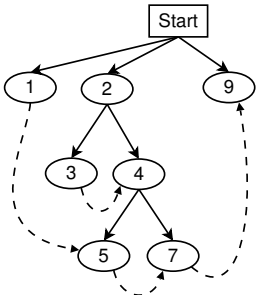

Figure 1. A small program and its dependence graph

of statements does matter and is taken into account; as a result the PDG never indicates influences which are in fact impossible due to the given statement order of the program.

Note that PDGs and slicing are much more complex for realistic languages with procedures, complex control flow, and data structures. An overview of fundamental slicing techniques can be found in [15, 28]; we will not discuss technical details here. For the full C or Java language, the computation of precise dependence graphs and slices is absolutely nontrivial; there is ongoing research worldwide since 15 years. The state of the art in PDGs and slicing is summarized in the recent work by Krinke [13].

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 [9]. 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 safe according to the Goguen/Meseguer noninterference criterion, if for all possible statement sequences x and all final statements a

$$output(run(z_0, x), a) = output(run(z_0, purge(x, dom(a))), a)$$

where *purge* removes all statements from statement sequence x which must not influence a's security level $dom(a)$. That is, the final program output must be unchanged if any statement which must not influence the last statement according to its security level is deleted.

Our following theorem demonstrates how PDGs can be used to check for noninterference.

Theorem. If

$$s \in BS(a) \implies dom(s) \rightsquigarrow dom(a)$$

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

Figure 2. A Java password checker

then the noninterference criterion is satisfied for a.

Proof. See [26]

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

The power and generality of the theorem should not be underestimated. Its generality 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.

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 does *not* have a PDG path (1) $\rightarrow^*$ (5) and thus is considered safe; no false alarm is generated.

3 PDGs for Java

In the following, we assume some familiarity with slicing technology, as presented for example in [28].

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. SDGs are *context-sensitive*, that is, different calls to the same procedure or method are indeed distinguished; avoiding spurious dependences.

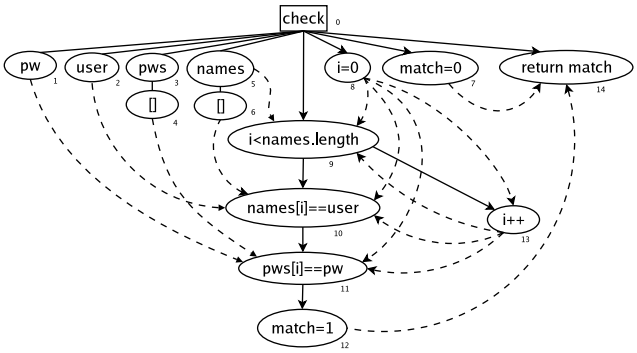

Figure 3. PDG for figure 2

While SDGs in general are well understood, dynamic dispatch and objects as method parameters make SDG construction more difficult. Treatment of dynamic dispatch is well known: possible targets of method calls are approximated statically (in our case using points-to [16, 22] information), and for all possible target methods the standard interprocedural SDG construction is done.

Figure 2 shows a small Java class for checking a password (taken from [18]) which utilizes methods and exceptions. The initial PDG for the `check` method can be seen in figure 3. Solid lines represent control dependence and dashed lines represent data dependence. Node 0 is the method entry with its parameters in node 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).

This PDG is still incomplete, as it does not include exceptions. 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 [6], 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 4.

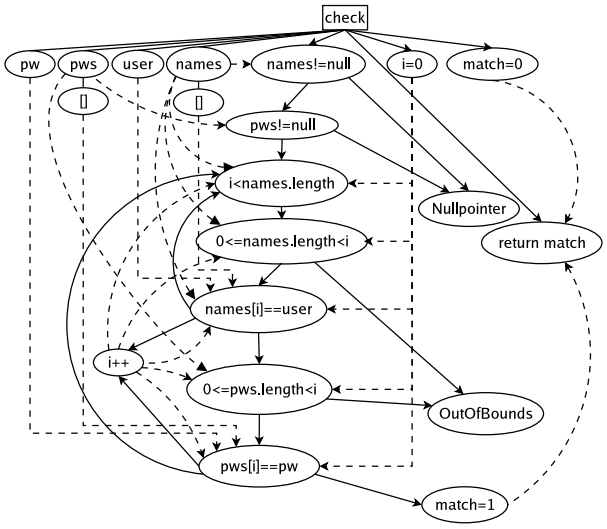

Figure 4. PDG with exceptions for figure 2

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. Thus we obtain a safe dependency criterion which leads to more precise Java slices than previous approaches [11].

Figure 5 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 holds two variables where `secure` is annotated with *High* and the other with *Low*. Thus both 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 provides a level not exceeding a given level, in our case *Low*. Statements that require a given security level are overlined.

Figure 6 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 6) between actual-in and actual-out parameter nodes. Summary edges have been introduced by Reps et al. They represent a transitive dependence between

```
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
8  class B extends A {
9    void set() { x = 1; }
10 }
11
12 class InfFlow {
13   static void main(String[] a) {
14     // part 1: no information flow
15     int secure = 0, pub = 1;
16     A o = new A();
17     o.set(secure);
18     o = new A();
19     o.set(pub);
20     System.out.println(o.get());
21     // part 2: dynamic dispatch
22     if (secure==0 && a[0].equals("007"))
23       o = new B();
24     o.set();
25     System.out.println(o.get());
26     //part 3: instanceof
27     o.set(42);
28     System.out.println(o instanceof B);
29   }
30 }
```

Figure 5. Another Java program

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 afterwards 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 (20) is a perfectly legal statement. Information flow control based on slicing can detect this fact: In figure 6 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 PDG based on points-to data, flow-sensitivity of PDGs, and from context-sensitivity of backward slicing with summary edges.

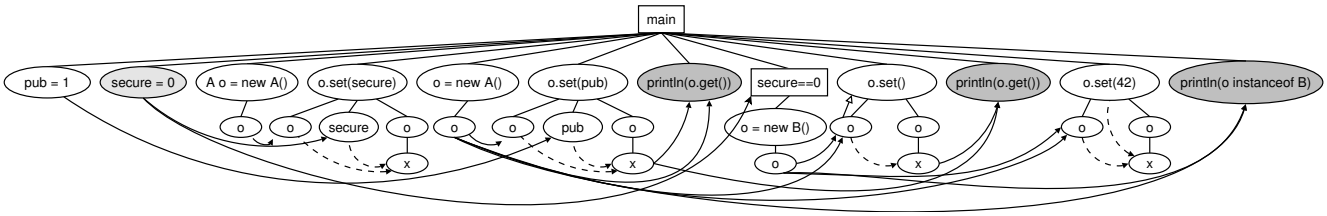

Figure 6. PDG for `main` in figure 5

The next statements show an illegal flow of information: Line (22) checks whether `secure` is zero and creates an object of class `B` in this case. The invocation of `set` on `o` 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. (22) - (24) are analogous to the following 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 (25).

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

Let us finally say a few words about PDGs for concurrent or multi-threaded programs which are common in Java. Krinke [12, 14] was the first author to present a precise algorithm for slicing concurrent programs. The method needs may-happen-in-parallel information to start with; several algorithms are known for this problem. Based on this so-called MHP analysis, Krinke's algorithm does not only add dependences between variables in different threads; it does in particular ensure that the sequence of statements in a PDG path does correspond to a possible execution order. Paths which contain impossible execution orders (and thus would introduce "time-traveling") are filtered out. This algorithm is precise, but expensive and limited to a fixed number of threads. We are currently implementing it and explore variations which trade precision for performance and flexibility.

4 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 resp. its 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

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

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

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

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 statements 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]. From those we compute the *actual* security levels and compare them with the required ones. The provided security levels are defined by a partial function $P : N \rightarrow L$, 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 $R : N \rightarrow L$. The actual security level $S(x)$ for a statement x is computed from the security levels of its predecessor and its own provided security level (if present):

$$S(x) = \begin{cases} P(x) \sqcup \bigsqcup_{y \in pred(x)} S(y) & \text{if } P(x) \text{ defined} \\ \bigsqcup_{y \in pred(x)} S(y) & \text{otherwise} \end{cases}$$

For simplicity in presentation, we extend P to a total function P' such that all statements have a provided security level:

$$P'(x) = \begin{cases} P(x) & \text{if } P(x) \text{ defined} \\ \bot & \text{otherwise} \end{cases}$$

Note that $\bot$ is the neutral element (for $\sqcup$). Now $l = P'(x)$ gives the provided security level of any statement x.

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

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

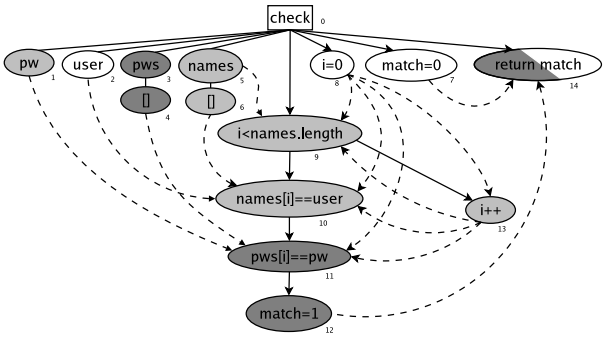

Figure 7. PDG for figure 2 with computed security levels

Due to the monotonicity of the computation and the limited height of the security level lattice, a minimal fixed point for S is guaranteed to exist and can be computed using a standard iteration.

The computed S ensures confidentiality; integrity can be computed similarly. However, for real confidentiality, the required security levels have to be checked against the computed ones:

$$\forall l = R(x) : l \geq S(x)$$

Thus, for any $l = R(x)$ such that $l \ngeq S(x)$ we have a confidentiality 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 $\ngeq$ and not $<$ because l and $S(x)$ might not be comparable.

Declassification is introduced into this model by another partial function $D : N \to L$ similar to P and R. This function specifies a declassification to security level d at statement s ($d = D(s)$). A declassification simply changes the computation of S:

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

The incoming security levels are ignored and replaced by the declassification security level. Note that the declassification may raise the security level if the incoming security level is lower than the specified declassification level.

The computation of $S(s)$ can be expressed through *transfer functions* $F : L \to L$ in the form $F(x) = p_s \sqcup x$ or $F(x) = d_s$ where p_s and d_s are constants (predefined by P and D). The transfer functions build a monotone function space and with the lattice of security levels L we have a monotone data flow analysis framework. Thus, all properties of monotone data flow analysis frameworks apply, particularly the existence of a minimal fixed point for S.

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 $P(3) = secret \wedge P(4) = secret$. The list of names and the parameter `password` is *confidential*, because they should never be visible to a user. Thus, $P(1) = confidential \wedge P(5) = confidential \wedge P(6) = confidential$. Starting with these provided security levels, we can compute the actual security levels which are depicted in Figure 7 through white for *public*, light gray for *confidential*, and gray for *secret*. In a first attempt, 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* ($R(14) = public$). However, this shows an interference, because $S(14) = secret$ and $secret \not\rightsquigarrow public$. It is clear that `match` has to be computed from secret information but it reveals no secure data. Therefore, the return statements gets a declassification to *public*, $D(14) = public$.

As usual, procedures or methods require context-sensitive analysis. Consider the following fragment, in which the function `f` has no side-effects and thus is safe:

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

Here, `secret` has a (provided) security level of *High*, `public` has *Low* and s has a (required) security level of *High* and p has *Low*. If we ignore calling context and handle the SDG edges like data or control dependence edges, we will have a violation for `p`: The security level *High* from `secret` will propagate into `f` due to the call `f(secret)`. Because computed security level of `f` is then *High*, the call `f(public)` will propagate *High* to p. To solve this issue, an approach similar to context sensitive slicing is used.

Some IFC checkers such as Jif [18] use a generalization of Denning's lattices, the so-called decentralized label model. This model 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.

But note that even decentralized labels can not overcome the impreciseness of type-based analysis. As an example, we adapted the first part of figure 5 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.

5 Path Conditions

In order to make the analysis more precise, we introduced *path conditions*, which are necessary conditions for information flow between two nodes.

The formulae for the generation of path conditions are quite complex (for details, see [26]), and only the most fundamental formula will be given here:

$$PC(x,y) = \bigvee_{P \text{ Path } x \to^* y} \bigwedge_{u \text{ node in } P} E(u)$$

where $E(u)$ is a necessary condition for the execution of u:

$$E(x) = \bigvee_{P \text{ Control Path } Start \to^* x} \bigwedge_{\nu \to \mu \in P} c(\nu \to \mu)$$

$c(\nu \to \mu)$ is a condition associated with dependence edge $\nu \to \mu$. For control dependences, $c(\nu \to \mu)$ is typically a condition from a while- or if-statement; for data dependences, more conditions constraining information flow through data types are added (see examples below). Program variables in a path condition are (implicitly) existentially quantified, as they are necessary conditions for potential information flow.

In Figure 1, we have

$$\begin{aligned}
&c(2 \to 3) \equiv c(2 \to 4) \equiv (n > 0),\\
&c(4 \to 5) \equiv (x > 0), \quad c(4 \to 7) \equiv (x \le 0),\\
&E(1) \equiv \mathit{true}, \quad E(3) \equiv (n > 0),\\
&E(5) \equiv (n > 0) \wedge (x > 0),\\
&PC(1,5) \equiv E(1) \wedge E(5) \equiv \exists n, x.(n > 0) \wedge (x > 0)
\end{aligned}$$

Slightly more interesting are the following program fragments and their path conditions:

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

$$PC(1,3) \equiv \exists i, j.(i > 10) \wedge (i + 3 = 2j - 42)$$

and

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

$$\begin{aligned}
PC(1,3) \quad &\equiv \quad \exists i, j.(i > 10) \wedge (j < 5)\\
&\qquad\qquad \wedge (i + 3 = 2j - 42)\\
&\equiv \quad \mathit{false}
\end{aligned}$$

These examples indicate that path conditions give precise conditions for information flow and can even determine that such flow is impossible even though there is a path in the graph.

We will not go into the details of path condition generation, but the reader should be aware that making path conditions work for full C resp. Java and realistic programs is non-trivial. Theoretical and practical work can be found in [10, 13, 20, 21, 26]. In particular, interprocedural path conditions, like PDGs and security levels, must be context-sensitive. Just to mention a few more things: the program must be transformed into single assignment form first; and while PDG cycles can be ignored, due to the high number of cycle-free PDG paths in realistic programs, interval analysis for irreducible graphs must be exploited to obtain a hierarchy of nested sub-PDGs; BDDs must be used to minimize the size of path conditions. Today, our implementation ValSoft can handle C programs up to approx. 10000 LOC and generate path conditions in a few seconds or minutes.

Path conditions for Java are even more complex, as inheritance and dynamic dispatch must be considered. As it is not known statically which method (re)definition will be called at runtime, interprocedural path conditions become *implications*. As an example, consider the call

```
y = o.f(x)
```

where f is redefined several times in classes A, B, and C. The path condition $PC(x,y)$, constraining information flow from the actual method parameter x to the return value y, will basically look as follows:

$$\begin{aligned}
PC(x,y) \quad \equiv \quad &(o \text{ instanceof } A \Rightarrow PC_A(x,y))\\
&\wedge (o \text{ instanceof } B \Rightarrow PC_B(x,y))\\
&\wedge (o \text{ instanceof } C \Rightarrow PC_C(x,y))
\end{aligned}$$

where PC_A, PC_B, PC_C are the standard path conditions for the three (re)definitions of f.

The example also shows that runtime type checks can be part of path conditions. Note that the number of implications can often be drastically reduced by exploiting points-to information. Other Java features such as exceptions, generic classes, static variables etc. do not influence the path condition generation: the PDG handles them correctly, and the standard formulae for path conditions apply.

Thus we omit further details on path conditions for Java, but present the path conditions for figure 5. First, remember that there is no path (17) $\to^*$ (20), thus $PC(17,20) \equiv$ false: no false alarm, thanks to flow and object sensitivity. But as mentioned earlier, there is a path from line 15 (node 2 in figure 6) to 25 via 18, 19, 20, 22 and 24. Line 25 prints the value of o's field x, and this value depends on the `secure` variable. Thus the path indicates a potential security leak, which is not obvious due to the dynamic dispatch in line 24. The path condition for the path is

$$\begin{aligned}
PC(15,25) \quad \equiv \quad &\mathit{true} \wedge \mathit{true} \wedge (\mathit{secure} == 0 \wedge a[0].\mathit{equals}(\text{“007”}))\\
&\wedge \mathit{true} \wedge \mathit{true} \wedge (o \text{ instanceof } A \Rightarrow \mathit{true})\\
&\wedge (o \text{ instanceof } B \Rightarrow \mathit{true})\\
\equiv \quad &\mathit{secure} == 0 \;\&\&\; a[0].\mathit{equals}(\text{“007”})
\end{aligned}$$

This path condition seems trivial enough, but demonstrates the concept of a witness: if the first program parameter is "007", then an illegal flow (15) $\rightarrow^*$ (24) will happen, and the printed value of o.x will depend on the value of the `secure` variable. Such a witness will be quite convincing to a jury in a law suit.

Generally speaking, path conditions for illegal flow which can be solved for the program's input variables act as a witness for a security leak. Providing input values according to the solved conditions makes any illegal information flow visible immediately. We have already seen a simple example for a witness. The general mechanism to compute witnesses works as follows.

When we discover an interference at a statement s where the required security level $R(s)$ is not larger than the actual (computed) security level $S(s)$, we can investigate the source of this interference. We distinguish between *immediate* and *transitive* interference. The immediate interference exists between s and its predecessors which lead to the computed security level $S(s)$. Usually, only a subset of the predecessors is responsible for the interference—it is the minimal subset $N \subseteq pred(s)$ that lead to $S(s)$: $S(s) = \bigsqcup_{y \in N} S(y)$.

Path conditions give the condition PI of the immediate interference and we can compute it through

$$PI = \bigwedge_{y \in N} PC(y, s).$$

Often, we are more interested in the *transitive* interference, i.e. the interference between a statement s with a required security level of l and a statement x with a provided security level p, where there is a path $x \rightarrow^* s$ which "transmits" p to s. To investigate the transitive interference, we use the correspondence between slicing and noninterference. The first step is to compute the backwards slice $BS(s)$ that gives all statements that may influence s. From $BS(s)$ we extract all statements with a provided security level as the possible set of information sources:

$$T = \{x \in BS(s) \mid x \in \text{dom } P\}$$

The computed security level cannot be smaller as (or not comparable to) any provided security level at its sources: $\forall x \in T : P(x) \leq S(s)$. Again, we need the minimal subset T' of T that computes $S(s)$: $S(s) = \bigsqcup_{y \in T'} S(y)$. Path conditions give the condition PT of the transitive interference and we can compute it through

$$PT = \bigwedge_{y \in T'} PC(y, s)$$

Note that there may exist multiple minimal subsets N and T' and we might want to examine all of them.

Applying these formulae to figure 7, we obtain the following witness:

$$\begin{aligned} PT \;\equiv\; & PC(3, 14) \equiv (i < \texttt{names.length}) \\ & \wedge(\texttt{names}[i] = \texttt{user}) \wedge (\texttt{passwords}[i] = \texttt{password}) \end{aligned}$$

This simple path condition is already in solved form. But in general solving path conditions for input variables, in order to obtain explicit witnesses, is not easy for realistic programs: they consist of huge heaps of conditions extracted from control statements such as if, while, switch; combined into substantial amounts of conjunctions and disjunctions with implicit existential quantifiers upfront. As a first step, a minimal disjunctive normal form is computed before any further constraint solving is attempted.

Since path conditions are existentially quantified, it is natural to apply quantifier elimination [30] and use systems such as Redlog [7]. Quantifier elimination replaces an existentially quantified variable by constraints on other variables, and the theory guarantees that both formulae are equivalent. But note that not all path conditions can be solved due to decidability problems. In practice, path conditions can be solved for medium-sized programs containing mostly arithmetic conditions.

6 Preliminary Experience

At the time of this writing, the Java slicer and security levels are fully operational, while the path condition generator is being adapted from the C version. To give the reader an idea how the system performs on realistic programs, we report some preliminary findings.

Our largest object of study is the `Purse` applet from the "Pacap" case study [5]. 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 184590 nodes and 1484975 edges. The time for PDG construction was 277 seconds plus 2338 seconds for generation of summary edges. The latter is measured separately as it is only necessary for context-sensitive slicing but requires an $O(n^3)$ algorithm.

Next, 10620 backward slices were selected by choosing a random node as a starting point. The average slice size is 109093 nodes, which is about 50% of the whole source code. This is typical for backward slices and illustrates why precise witnesses can only be achieved via path conditions as described in section 5.

As a case study for IFC we chose another JavaCard applet called `Wallet`[2]. It is only 252 lines long but with the necessary API parts and stubs the PDG consists of 21274 nodes and 87726 edges. The time for PDG construction was 16 seconds plus 19 for summary edges.

Access to the wallet is granted only after supplying the correct PIN which we annotated with the provided security level *high*. We annotated the response method to the terminal with required level *low*. A check with our system found—as expected—that the PIN value might leak

[2] `www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html`

out through the method checking the entered PIN. This is a classic application of declassification: after downgrading the boolean result to *low* this method showed no further violations.

7 Related Work

Several papers have been written about PDGs and slicers for Java, but to our knowledge only the Indus slicer developed by Hatcliff et al. [19] 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 allows time traveling for concurrent programs.

We already mentioned the overview article by Sabelfeld and Myers [23], 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. But perhaps the IFC community ignored PDGs so far, because precise PDGs for full C or Java are so difficult to construct.

Abadi et al. [1] was the first author to connect slicing and noninterference, but only for λ-calculus. It is amazing that our theorem from section 2 (which holds for imperative languages and their PDGs) was not discovered earlier. Only Anderson et al. [3] 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.

Volpano and Smith [29] were the first to present a nonstandard 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 [24].

Myers [17] defines JFlow, an extension of the Java language with a type system for information flow. The JIF compiler [18] implements this language. We already discussed 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 [4] 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 [27] formulates a non-deterministic type system including the noninterference proof. It handles mayor concepts of MicroJava such as classes, fields and method calls, but omits arrays and exceptions.

The Pacap case study [5] verifies secure interaction of multiple JavaCard applets on one smartcard. 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 by bytecode instructions are modeled, but such unstructured control flow may lead to *label creep* (cf. [23, Sect. II E]).

Genaim [8] 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 [11] 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 [10]. 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.

8 Conclusion

We presented a system for information flow control in Java programs, which is based on path conditions in dependence graphs. Such path conditions are very precise necessary conditions for information flow between two program points. 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. We can handle the full Java language (except reflection) and can analyze medium-sized programs. Our approach defines and exploits, for the first time, path conditions in Java PDGs.

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.

Right now, we can handle only medium-sized programs. Security kernels are usually not really big; still, a better scale-up is an issue. 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.

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. Our final outcome, namely witnesses for illegal information flow, will be very valuable in law suits against malicious software vendors.

Acknowledgment. We thank Frank Nodes and Florian Tausch for their implementation work.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. Symposium on Principles of Programming Languages*, pages 147–160. ACM, 1999.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 40–44. ACM Press, 2000.

[3] P. Anderson, T. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Transactions on Software Engineering*, 29(8), aug 2003.

[4] 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*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[5] 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.

[6] 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*, pages 35–52. Springer-Verlag, 1999.

[7] A. Dolzmann and T. Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.

[8] S. Genaim and F. Spoto. Information flow analysis for java bytecode. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *LNCS*, pages 346–362, Paris, France, Jan. 2005. Springer.

[9] J. Goguen and J. Meseguer. Interference control and unwinding. In *Proc. Symposium on Security and Privacy*, pages 75–86. IEEE, 1984.

[10] 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)*, pages 58–67, Jan 2006.

[11] 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*, pages 17–22. ACM Press, 2004.

[12] J. Krinke. Static slicing of threaded programs. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, 1998.

[13] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, July 2003.

[14] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proc. FSE/ESEC*, pages 178–187. ACM Press, 2003.

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

[16] O. Lhotak and L. Hendren. Scaling Java points-to using Sparc. In *Compiler Construction, 12th International Conference*, LNCS, pages 153–169, 2003.

[17] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the ACM Symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[18] A. C. Myers, N. Nystrom, L. Zhebg, and S. Sdanewic. Jif: Java information flow. http://www.cornell.edu/jif.

[19] V. Ranganath, T. Amtoft, A. Banerjee, M. Dwyer, and J. Hatcliff. A new foundation for control dependence and slicing for modern pogram structures. In *Proceedings of the European Symposium on Programming (ESOP'05)*, pages 77–93. LNCS 3444, 2005.

[20] T. Robschink. *Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik*. PhD thesis, Universität Passau, Januar 2005. in German.

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

[22] 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*, pages 43–55, 2001.

[23] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

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

[25] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *Proc. Static Analysis Symposium*, volume 1145 of *LNCS*, pages 332–348, 1996.

[26] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology*, to appear 2005.

[27] M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[28] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[29] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. TAPSOFT'97*, LNCS 1214, pages 607–621, Lille, France, Apr. 1997.

[30] V. Weispfenning. Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation*, 24(2):189–208, 1997.