Static Path Conditions for Java

Christian Hammer *
Universität Karlsruhe (TH)
Germany
hammer@ipd.info.uni-karlsruhe.de

Rüdiger Schaade
Universität Passau
Germany

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

Abstract

A static path condition is a precise necessary condition for information flow between two program points. Previous work defined path conditions for procedural languages. Object oriented languages offer additional constructs such as dynamic dispatch, instanceof and exceptions. In this paper, we present an analysis of these constructs, which leads to precise path conditions operating only on the program's variables. This yields a gain in precision, allowing leverage of automatic constraint solving. We present details of path condition generation for Java constructs, and discuss preliminary insight from our prototype implementation.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods; D.1.5 [*Programming Techniques*]: Object-oriented Programming; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

General Terms Algorithms, Languages, Security, Verification, Theory

Keywords Information Flow Control, Path Condition, Dynamic Dispatch, Program Slicing, Java

1. Introduction

Information Flow Control (IFC) is a technique for discovering security leaks which may damage confidentiality or integrity of a software system. Language-based IFC analyses the source code of a program in order to check for confidentiality or integrity, and has become a rapidly developing field (see [28] for an overview). Language-based IFC is related to program semantics as well as program analysis. Many authors proposed to implement IFC using non-standard type systems [29, 25]. For example, the Mobius project is developing sophisticated theoretical foundations as well as practical implementations and machine-checked correctness proofs for type-based IFC and proof-carrying Java code on mobile devices [18, 3].

But type-based approaches, while effective, do not fully exploit the program analysis technology of today. Type-based analysis is usually not flow-sensitive, context-sensitive, or object-sensitive, leading to imprecision and false alarms. A more precise approach to IFC is based on program slicing and was first proposed by the current authors [30, 14, 15]. IFC based on slicing and dependence graphs is flow-sensitive, context-sensitive and object-sensitive [16], and has been proven to be consistent with the traditional notion of noninterference [31]. Slicing-based IFC utilizes the sophisticated program analysis technology available today, and demonstrated to be applicable to several 10000 lines of code. In particular, our Java slicer [16] has been augmented with security levels and declassification, as well as dataflow equations for precise interprocedural computation of security levels; this slicing-based IFC for Java has been used on several realistic case studies [14, 15].

But both type systems and program slicing are "dumb" in the following sense: they can only provide a binary answer to a security question; that is, they can state whether illegal flow between two program points is possible, or whether this is definitely not the case. They may be able to locate the source point making trouble, but they are unable to provide insight into the specific conditions of a security violation. In particular, they can not generate input values which make the security violation visible.

Path conditions, as first proposed in [30], are much more precise because a path condition says *why* (i.e. under which conditions) information flow takes place. Path conditions and its underlying program slicing machinery are designed for high precision as well as scalability, and aim to support realistic languages like C or Java bytecode.

Details of path conditions have been presented in [31]. This paper also described applications of path conditions for IFC in realistic C programs, and established the connection between path conditions and noninterference. For Java, however, how to generate precise path conditions has not been defined yet.

In this paper, we present the first path condition generator for Java which relies solely on static program analysis. Of course, the main problem in generalizing path conditions from C to Java was the treatment of object-oriented features such as dynamic dispatch and dynamic type checks.

1.1 Overview

Path conditions are generated according to all possible dependence paths from the source to the target point. A constraint solver reduces these conditions to input values that trigger information flow between these two points. Such input values have been named "witnesses" in [31], and may be quite helpful e.g. in law suits concerning security violations. However, [31] only defines path conditions for procedural languages. Our preliminary idea for Java presented in [15] suffers from path conditions containing dynamic type checks, which constraint solvers cannot simplify statically. Thus witnesses cannot be generated, and the path conditions from [15] are only a first step towards realistic applications.

* This research was supported by Deutsche Forschungsgemeinschaft (DFG grant Sn11/9-2).

PLAS'08, June 8, 2008, Tucson, Arizona, USA.

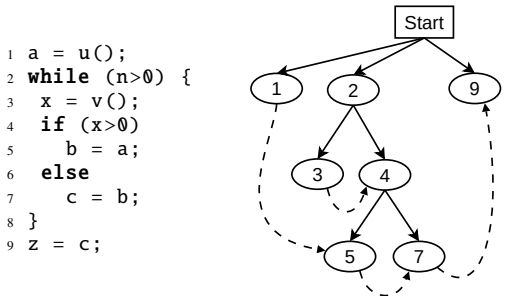

Figure 1: A small program and its dependence graph

This paper presents a detailed study of object-oriented language constructs based on Java's language specification, as well as solutions for their integration into a path condition that contains only program variables and values. We start discussing dynamic type checks à la `instanceof` and extend the solutions found there to dynamic dispatch and exception handling. All these constructs are based on dynamic type checks. These checks can either be approximated conservatively, or — using program slicing — be transformed into a subcondition that no longer involves the program's types but ranges over program variables. Thus we present the first approach to generate realistic path conditions for Java.

We have implemented a prototype path condition generator for Java based on the existing implementation for C and our dependence graph generator for Java [16], and demonstrate the feasibility of our extensions with preliminary case studies.

2. Foundations of Path Conditions

Program dependence graphs (PDG) are a standard tool to model information flow through a program. Program statements or conditions are represented by the nodes, the edges represent data and control dependences between statements or conditions. 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 (possibly) influencing y (the so-called *backward slice*) are easily computed as $BS(y) = \{x \mid x \rightarrow^* y\}$

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

A chop for a chopping criterion (x, y) is the set of nodes that are part of an influence of the (source) node x on the (target) node y. This is the set of nodes that lie on a path from x to y in the PDG: $CH(x, y) = \{z \mid x \rightarrow^* z \rightarrow^* y\}$. For convenience, we will also use $CH(x, y)$ for the set of paths between x and y.

Note that PDGs, slicing and chopping are much more complex for realistic languages with procedures, pointers, complex control flow, and data structures. An overview of fundamental slicing techniques can be found in [20]; technical details of our dependence graph generator for full Java bytecode can be found in our previous work [16].

2.1 Intraprocedural Path Conditions

In order to make program slicing more precise, Snelting introduced *path conditions* [30], which are necessary conditions for information flow between two nodes. The formulae for the generation of path conditions are quite complex (for details, see [31]), and only the most fundamental formula will be given here:

$$PC(x, y) = \bigvee_{P \in CH(x,y)} \bigwedge_{z \in P} E(z) \quad \text{where}$$

$$E(z) = \bigvee_{P \text{ Control Path } Start \rightarrow^* z} \bigwedge_{\nu \rightarrow \mu \in P} c(\nu \rightarrow \mu) \quad (1)$$

$PC(x, y)$ is a necessary condition for flow from x to y, and $E(z)$ is a necessary condition for the execution of z. A control path is a path that consists of control dependence edges only. Thus, $E(x)$ is computed along all control paths from the *Start* node of the function to x based on the conditions $c(\nu \rightarrow \mu)$ associated with dependence edge $\nu \rightarrow \mu$. For control dependences, $c(\nu \rightarrow \mu)$ is typically a condition from a `while`- or `if`-statement. Program variables in a path condition are (implicitly) existentially quantified, as they are necessary conditions for potential information flow.

In [31] Snelting et al. argue why this formula is correct and precise, and why it improves slicing considerably. Note that cycles in $CH(x, y)$ can safely be eliminated for $PC(x, y)$, such that the formula is always finite. For the example in Figure 1, the following execution and path conditions are computed:

$$c(2 \rightarrow 3) \equiv c(2 \rightarrow 4) \equiv (n > 0), \quad c(4 \rightarrow 5) \equiv (x > 0),$$
$$c(4 \rightarrow 7) \equiv (x \leq 0),$$
$$E(1) \equiv \mathit{true}, \quad E(3) \equiv (n > 0), \quad E(5) \equiv (n > 0) \wedge (x > 0),$$
$$PC(1, 5) \equiv E(1) \wedge E(5) \equiv (n > 0) \wedge (x > 0)$$

In the presence of data structures like arrays or pointers, additional constraints will be generated. For data dependences, $\Phi(\nu \rightarrow \mu)$ is a condition constraining information flow through data types. As an example we consider arrays (a full presentation can be found in [31]): A data dependence $\nu \rightarrow \mu$ between an array element definition $a[E_1] = \ldots$ and a usage $\ldots = a[E_2]$ generates $\Phi(\nu \rightarrow \mu) \equiv E_1 = E_2$; all other data dependences will generate $\Phi(\nu \rightarrow \mu) \equiv \mathit{true}$. The equation to compute a path condition now becomes:

$$PC(x, y) = \bigvee_{P \in CH(x,y)} \left(\bigwedge_{z \in P} E(z) \wedge \bigwedge_{u \rightarrow v \in P} \Phi(u \rightarrow v) \right) \quad (2)$$

For clarification consider the following program fragments and their path conditions:

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

and

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

with their path conditions:

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

and

$$PC(1, 3) \equiv (i > 10) \wedge (j < 5) \wedge (i + 3 = 2j - 42) \equiv \mathit{false},$$

as this condition is not satisfiable.

These examples indicate that path conditions give precise conditions for information flow, and can sometimes determine that such flow is impossible even though there is a path in the graph. Note that in practice path conditions tend to be large and a constraint solver is used to simplify them.

2.2 Interprocedural Path Conditions

In analogy to interprocedural slicing and chopping, interprocedural path conditions need to be restricted to *realizable* [20] paths. Intuitively, this means that a path in the system dependence graph (SDG) [17] — i.e. several PDGs connected by interprocedural edges according to the call graph — that enters a procedure through a certain invocation site must not leave it at a different invocation

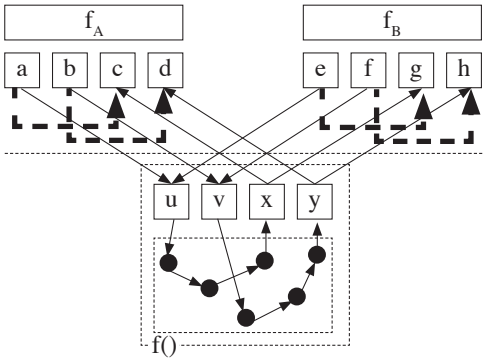

Figure 2: Abstract structure of multiple invocations of method `f`

site. As an illustration, consider Figure 2. Here, there are only two realizable paths at call site `A` (left), namely from parameter `a` to `c` and from `b` to `d`. These transitive dependences, stemming from dependences in method `f`, are represented in the SDG as summary edges (dashed in Figure 2). The path from `a` to `g` is invalid, as parameters from one invocation cannot influence another invocation.

In the following, we will focus on path conditions where start and end node are in the same method, and will concentrate on object-oriented constructs. The general case, where start and end node of the path condition lie in different methods, is not specific to Java and has been addressed in previous work [31].

Path conditions factor out common subpaths with virtual decomposition [31]; common subpaths between formal parameters connected by a summary edge have been found good candidates for such decomposition. Virtual decomposition ignores call and parameter-passing edges but includes summary edges instead. A specific Φ condition (which is conjunctively added to the execution condition) represents the condition for information flow along a summary edge [31]. This condition is induced by the path condition between the corresponding formal parameters combined with conditions that represent parameter binding. In Figure 2 the condition between `a` and `c` would be:

$$\Phi(a \to c) \equiv u = a \wedge PC(u, x) \wedge c = x$$

Thus $PC(u, x)$ can be reused at other call sites with a different parameter binding.

Details of path condition generation are not presented here, but the reader should be aware that making path conditions work for full C and realistic programs required years of theoretical and practical work [30, 27, 26, 31]. Today, the 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 implemented as an extension of ValSoft, thus utilizing the sophisticated ValSoft implementation for procedural language constructs.

3. Dynamic Type Tests

In this section, we explore the precise semantics of the `instanceof` operator in order to utilize it in path conditions. Informally, the result of the expression `e instanceof T` is true iff the value of the expression e is not `null` and e has a runtime type that is below the type constant T in the type hierarchy [12]. A compile time error occurs, when no path exists in the type hierarchy from T to the static type of e. The language thus allows e to have a static type which is equal to or below T in the hierarchy, even though in that case, unless e is `null`, the expression will always evaluate to *true*. All this is well known, but path conditions require a precise formalization of the `instanceof` semantics.

3.1 Precise semantics for `instanceof`

Java's reference types are identical to the defined classes (and interfaces) in a program, so the terms type and class (interface) are used interchangeably. We write $A <: B$ if A is a subclass of B. A type hierarchy is the transitive closure of the subclass ($<:$) relation. Let $\mathcal{C}$ be the set of class types, $\mathcal{I}$ the set of interface types, $\mathcal{R} = \mathcal{C} \cup \mathcal{I}$ the set of reference types, $\mathcal{P}$ the set of primitive types and $\mathcal{A}$ the set of array types. Further let $\mathcal{S} = \{\texttt{java.io.Serializable}, \texttt{Cloneable}, \texttt{Object}\}$. The notation $e : \tau$ denotes that e has dynamic/runtime type τ, and $e :: \tau$ denotes static typing, respectively.

The type hierarchy induces concrete sets of types that satisfy the *instanceof* operator: Let Γ_τ denote the set of types in the hierarchy that evaluate to *true* in the `instanceof` τ expression, i.e.

$$e \texttt{ instanceof } \tau \equiv e : \rho \wedge \rho \in \Gamma_\tau \tag{3}$$

For brevity, we assume a special `null`-type with $\Gamma_{\texttt{null}} = \emptyset$ Then the definition of `instanceof` in the JLS [12] requires:
If $\tau \in \mathcal{R}$ then

$$\rho \in \Gamma_\tau \quad \text{iff} \quad (\rho \in \mathcal{C} \wedge \rho <: \tau) \ \vee \ (\rho \in \mathcal{A} \ \wedge \tau \in \mathcal{S}) \tag{4}$$

else if $\tau \in \mathcal{A}$ then

$$\rho \in \Gamma_\tau \quad \text{iff} \quad \rho = \rho'[] \wedge \tau = \tau'[] \wedge \rho' \in \Gamma_{\tau'} \tag{5}$$

else if $\tau \in \mathcal{P}$ then

$$\rho \in \Gamma_\tau \quad \text{iff} \quad \rho = \tau \tag{6}$$

The last term of equation (5) corresponds to Java's covariant array anomaly, that may result in type safety problems when storing into arrays. When the complete type hierarchy is given at analysis time, Γ_τ can easily be computed from the type hierarchy with e.g. class hierarchy analysis (CHA) [7] or more refined analyses like rapid rype analysis (RTA) [2] and the XTA algorithm [33]. A few special cases of Γ_τ are of high importance:

$$\Gamma_{Object} = \mathcal{C} \cup \mathcal{A}, \text{ which is infinite in principle[1] due to } \mathcal{A} \tag{7}$$

$$\Gamma_{Cloneable} = \{X \mid X <: Cloneable\} \cup \bigcup_{c \in \mathcal{R} \cup \mathcal{A} \cup \mathcal{P}} c[] \tag{8}$$

3.2 Path Conditions for `instanceof`

The Γ_τ constructed in the previous section are necessary to describe the generation of precise path conditions for `instanceof`. Note that these path conditions still contain dynamic type tests of the form $expr : \rho$; these will be removed by transformation in the next section.

Algorithm 1 Path condition for `instanceof`

Input: An expression $expr$ `instanceof` τ
Output: Corresponding path condition with dynamic type tests.

1: **if** $\tau = Object \vee (\tau \in \mathcal{S} \wedge expr :: \tau'[]) \vee (\tau = \tau'[] \wedge \tau' \in \mathcal{P})$ **then**
2: **return** $expr \neq null$
3: **else**
4: **return** $expr \neq null \wedge (\bigvee_{\gamma_i \in \Gamma_\tau} expr : \gamma_i)$
5: **end if**

Algorithm 1 presents how path conditions for instanceof expressions are computed; the algorithm is based on the precise `instanceof` semantics above. The differences between equations (4)–(6) and the algorithm stem from optimizations which are done at compile time, see the JLS [12]. For example, $\Gamma_{\texttt{int[]}} =$

[1] In Java, the number of array dimensions is bounded by 255 [21].

```
1  // pre: B extends A
2  public class InstanceOfExample {
3    static boolean pred = true;
4    public static void main(String[] args) {
5      A a = pred ? new A() : new B();
6      System.out.println(instanceOf(a));
7    }
8    public static int instanceOf(A sel) {
9      int result = 0;
10     if (sel instanceof B)
11       result = 42;
12     return result;
13   }
14 }
```

Figure 3: An example for the `instanceof` operator

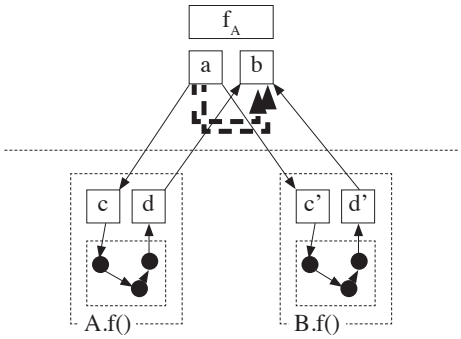

Figure 4: Virtual method call with two possible targets and summary edges

{`int[]`}, therefore the type test is done at compile time and no further runtime constraint is required but the test for `null`.

Obviously, the more refined Γ_τ is determined, the more precise the condition for the instanceof expression becomes. As points-to analysis is usually a prerequisite for precise slicing, points-to results can be leveraged to increase precision of Γ_τ. If the number of possible runtime types thereby reduces to less than or equal to 1, the path condition can immediately be reduced to *true* (if the remaining type is an instance of τ, provided that the expression can never be `null`) or $false$. As usual, determining Γ_τ requires whole-program analysis either without reflection, or using conservative approximations (e.g. [23]).

For the example program in Figure 3, the initial path condition between the parameter in line 8 and the return value in line 12 is

$$PC(\texttt{sel}_8, \texttt{result}_{12}) \equiv \mathit{sel}\ \texttt{instanceof}\ B$$

as no other path between these program points exists in the PDG. Since `B` is no special type, Algorithm 1 replaces this condition by

$$PC(\texttt{sel}_8, \texttt{result}_{12}) \equiv \mathit{sel} \neq \mathit{null} \wedge \mathit{sel} : B$$

3.3 Exploiting backward slices in type tests

Path conditions for `instanceof`, as described so far, are of limited practical value, as they may contain type tests with no link to variable values (e.g. $sel : B$), and thus cannot serve as a witness. This section presents a novel technique to transform such conditions into a form containing only program variables and values, which thus can be used to generate witnesses. The fundamental idea is to *replace variables in runtime type checks by their backward slice*.

The conditions in the last section contained terms of the form $\bigvee_{i\in\Gamma_\tau}(e : \gamma_i)$ for some type τ, that are essentially runtime checks. Program slicing offers a means to replace these conditions with terms that only reference program variables. The term $e : \gamma_i$ depends on the last definition of e and will evaluate to *true* only if e had been assigned an instance of γ_i. The program dependence graph allows to resolve places from which γ_i-allocations reach a given statement. This value flow is contained in the so-called backward data slice [4], or more precisely, the statements in a thin slice [32].

The following steps are to be taken to generate this refined path condition:

1. Determine the basic path condition p according to Algorithm 1 in section 3
2. Compute the backward data slice or thin slice for the parameter e of the `instanceof` τ operator
3. For each type in p extract the allocation sites of that type from the slice
4. Concatenate the path conditions from the program's start node to each of these allocation sites and from there to the `instanceof` expression with *logical or*
5. Replace the dynamic type checks in the basic path condition with the term generated in the last step.

Formally, with $J_i := \{a_{i,j} \in BS_{thin}(e) \mid a_{i,j}$ is an allocation site of type $\gamma_i\}$ we obtain the fundamental equation

$$e : \gamma_i \Rightarrow \bigvee_{j=1}^{|J_i|} \Big(PC(\texttt{Start}, a_{i,j}) \wedge PC(a_{i,j}, e) \Big) \tag{9}$$

Informally, when e has runtime type γ_i, the program must have passed an allocation site of that type, and this is only possible at one of the allocation sites that reach expression e. Interprocedural reaching definitions are modeled in the thin slice. $PC(\texttt{Start}, a_{i,j})$ in equation (9) is necessary to reach allocation site $a_{i,j}$ (included in the thin slice) from the program's beginning, and $PC(a_{i,j}, e)$ is required to get from there to expression e. Note that taking one of these paths (i.e. $PC(\texttt{Start}, a_{i,j}) \wedge PC(a_{i,j}, e)$ holds) does not guarantee that e has dynamic type $a_{i,j}$ as slicing may be conservative, so equation (9) is only an implication.

This equation has the additional advantage that the number of terms becomes finite due to the finite number of allocation sites, as opposed to the theoretically infinite sets $\Gamma_{\tau[]}$. If the program representation contains all default initializations of variables explicitly, tests for `null` can be omitted if the backward data slice does not contain such a value.

Considering the example in Figure 3 again, the condition from Algorithm 1 ($sel \neq null \wedge sel : B$) needs to be refined using equation (9). The test for `null` is redundant, as all program paths define `sel` with a non-`null` value. The backward data slice of `sel` yields both allocations in line 5, where only the second has appropriate type. Hence equation (9) collapses to

$$\begin{aligned} PC(\texttt{sel}_8, \texttt{result}_{12}) &\equiv \mathit{sel} : B \\ &\equiv ((\mathit{pred} = \mathit{true}) \wedge !\mathit{pred} \wedge \mathit{true}) \equiv \mathit{false} \end{aligned} \tag{10}$$

Thus the parameter `sel` cannot influence the outcome of this method, even though the program slice says so.

4. Dynamic Dispatch

Dynamic dispatch has great influence on path conditions: In contrast to statically bound methods, a virtual method call might have multiple possible target methods, one of which is executed at runtime according to the type of the target object. As an example, Figure 4 shows a method invocation site for a target object of static type `A`, which could dispatch either to `A.f()` or `B.f()`. The invoca-

```
1  class A {
2    int result = 42;
3    int f(int x) {
4      if (x < 1)
5        result = x;
6      return result;
7    }
8    public static void main(String[] args) {
9      A o = new B();
10     int x = 2;
11     int y = o.f(x);
12 }}
13 class B extends A{
14   int f(int x) {
15     if (x < 2)
16       result = x;
17     return result;
18 }}
19 class C extends A {
20   int f(int x) {
21     if (x < 3)
22       result = x;
23     return result;
24 }}
```

Figure 5: Example program for dynamic binding

tion site holds two parameter nodes `a` and `b` where two summary edges (dashed) model the transitive flow that is possible between the formal parameters in the possible target methods.

4.1 A naive path condition

Since it is statically unknown which target method is executed, one might obtain the following naive path condition for dynamic dispatch of method `f`, which simply disjuncts all possible cases:

$$PC_{\mathtt{f}}(x,y) \equiv \bigvee_{i=1}^{n} PC_{\gamma_i.\mathtt{f}}(x,y) \tag{11}$$

where x and y are in general two actual parameters of the call (connected by a summary node), $PC_{\gamma_i.\mathtt{f}}(x,y)$ is the condition of the invocation target for type γ_i between the formal parameter nodes corresponding to x and y. Note that not all subclasses must redefine `f`, so γ_i.`f` might in fact reference a definition of `f` in a superclass of γ_i. For example, if `C <: B <: A` and only `C` and `A` (re)define `f`, then `B.f` is actually `A.f`.

For Figure 4, equation (11) yields

$$\begin{aligned} PC(a,b) \quad \equiv \quad & ((c = a) \wedge PC_A(c,d) \wedge (b = d)) \\ & \vee ((c' = a) \wedge PC_B(c',d') \wedge (b = d')) \end{aligned}$$

For the example program in Figure 5, the path condition between x and y on line 11 would be:

$$\begin{aligned} PC(x,y) \equiv PC_A(x,y) \vee PC_B(x,y) \vee PC_C(x,y) \\ \equiv x < 1 \vee x < 2 \vee x < 3 \equiv x < 3 \quad (*) \end{aligned}$$

where PC_A, PC_B, PC_C are the standard path conditions for the three (re)definitions of f between the formal parameter nodes corresponding to x and y.

While this path condition is correct (it is a necessary condition for information flow) and easy to build, it is too imprecise since the program semantics disallows more than one target method. We will therefore develop an approach to handle dynamic dispatch similar to the `instanceof` expression in the last section.

4.2 Exploiting slices again

Previous work [15] already presented a first step towards interprocedural path conditions, precisely expressing the semantics of dynamic dispatch. The basic terms essentially become *implications* on the target object's runtime type.[2] For the example program in Figure 5, the path condition between x and y on line 11 would be:

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

In general, for a dynamically dispatched call `y = o.f(x)`, the JLS [12] induces the following fundamental condition:

$$PC_{\mathtt{f}}(x,y) \equiv \bigwedge_{i=1}^{n} (o : \gamma_i \Rightarrow PC_{\gamma_i.\mathtt{f}}(x,y)) \tag{12}$$

An alternative formulation of this condition is:

$$PC_{\mathtt{f}}(x,y) \equiv \bigvee_{i=1}^{n} (o : \gamma_i \wedge PC_{\gamma_i.\mathtt{f}}(x,y)) \tag{13}$$

The equivalence proof for equations (12) and (13) can be found in the appendix. The proof relies on the fact that not more than one of the disjunctions can be satisfied at the same time, and holds provided $o \neq \text{null}$. In case $o = \text{null}$, formula (13) is more precise than (12), as it correctly evaluates to false: a flow of information through the method body is impossible, since an exception is thrown.

However, both path conditions are of limited practical value, as they again contain dynamic type tests and thus cannot serve as a witness. But again equation (9) is applicable to this condition, so the dynamic type tests can be transformed to a complex path condition with no explicit type tests. In general, we obtain the equation

$$\begin{aligned} PC_{\mathtt{f}}(x,y) \equiv \bigvee_{i=1}^{n} \Big(\Big(\bigvee_{j=1}^{|J_i|} PC(\mathtt{Start}, a_{i,j}) \wedge PC(a_{i,j}, o) \Big) \\ \wedge PC_{\gamma_i.\mathtt{f}}(x,y) \Big) \end{aligned} \tag{14}$$

Note that the condition of equation (14) is slightly weaker (but still conservative) than equation (13), as equation (9) is no equivalence but an implication. For Figure 5, we obtain:

$$\begin{aligned} PC(x,y) \equiv (false \wedge PC_{\mathtt{A}}(x,y)) \vee (true \wedge PC_{\mathtt{B}}(x,y)) \vee \\ (false \wedge PC_{\mathtt{C}}(x,y)) \equiv x < 2 \quad (***) \end{aligned}$$

as the backward data slice contains only the allocation in line 9, so this condition is more precise than the basic formula (11). Note that $(**)$ was already more precise than $(*)$, while $(***)$ now collapses to a simple condition without type tests "$x < 2$", which is more precise than the "$x < 3$" in $(*)$.

5. Exceptions

In principle any subtype of `Throwable` can be caught in Java, even subtypes of `Error`. The latter indicate a VM failure from which recovery is typically not possible, so catching `Errors` or `Throwable` is discouraged and not discussed in this paper.

As `finally` blocks are always executed, they can be incorporated into the CFG as usual and do not impose new challenges for path conditions. However, `catch` blocks can alter the control and data flow and must therefore be treated accordingly. It is possible to have multiple `catch` blocks for the same `try` block. In this

[2] [15] used 'instanceof' in the example, which might be misleading due to Java's `instanceof` operator. The formulae in this work present the exact semantics.

```
1  public static void main(String[] args) {
2   System.out.println(exceptionMethod(1));
3  }
4  public static int exceptionMethod(int i) {
5   try {
6     return 5/i;
7   } catch (ArithmeticException e) {
8     return 0;
9   } catch (RuntimeException e) {
10    return Integer.MAX_VALUE;
11 }}
```

Figure 6: Example for exception handling

case, the appropriate handler is determined according to the exception's class. The (textually) first catch block a thrown exception matches handles that exception, matching is done according to the `instanceof` relation [21].

In order to generate appropriate path conditions for this well-known exception behavior pattern, we model multiple `catch` as follows: Multiple catch blocks can be translated to a `typeswitch` construct which is branched to when an exception is raised. This modeling results in control dependences labeled with type boundaries, for which execution conditions based on `instanceof` expressions can be leveraged in a straightforward manner. If multiple blocks would match, it is conservative to have multiple conditions evaluate to *true*. However, to represent Java's semantics precisely and to achieve maximum precision, we need to ensure that types that are caught in previous catch blocks may not evaluate to true.

Formally, let $E =< e_1, \ldots, e_k >$ be the sequence of exception handlers associated with a `try` block with type boundaries e_i. Then the control condition for the `typeswitch` branch involves a dynamic type test of the form $e : \rho \wedge \rho \in \Gamma_{e_i}$. Using the adjusted definition of $\Gamma'_{e_j} = \Gamma_{e_i} \setminus (\bigcup_{j=1}^{i-1} \Gamma_{e_j})$ represents the exact semantics of exception handling and thus will report more precise results when applied to Algorithm 1 and equation (9).

As an example consider Figure 6, where the path condition between parameter `i` and the second `catch` block is to be computed. The original path condition yields:

$$\begin{aligned} & i = 0 \wedge exc_1 \in \Gamma_{\texttt{RuntimeExc}} \\ & \quad \equiv i = 0 \wedge (exc_1 : \texttt{RuntimeExc} \vee exc_1 : \texttt{ArithmethicExc}) \\ & \qquad \equiv i = 0 \wedge (i = 0 \vee false) \equiv i = 0 \end{aligned}$$

as no other exception but `ArithmeticException` can be thrown in `exceptionMethod`, while Algorithm 1 with the refined Γ' will result in the condition

$$\begin{aligned} i = 0 \wedge exc_1 \in \Gamma'_{\texttt{RuntimeExc}} & \equiv i = 0 \wedge exc_1 : \texttt{RuntimeExc} \\ & \equiv i = 0 \wedge false \equiv false \end{aligned}$$

showing that this `catch` block is actually dead code.

5.1 Interprocedural Exceptions

Java supports two types of exceptions: Unchecked and checked exceptions. The former are any subtype of `RuntimeException` with the main purpose of signaling a problem of bytecode interpretation. Most bytecode instructions involved with object references and array access may for example throw `NullpointerException`s or `ArrayIndexOutOfBoundsException`s. Nearly every method in Java might throw an exception, and this needs special attention when modeling interprocedural exception handling. For each invocation site, our SDG contains two return value nodes (one for the usual return value and one for an uncaught exception), and two successors: one if method invocation terminated normally and the other

```
1  public class Weighing {
2    public static final int NORMAL = 0;
3    public static final int PAPER_OUT = 1;
4    public static void main(String[] args) {
5      char input = args[0].charAt(0);
6      weigh(PAPER_OUT, 1.0f, input);
7    }
8    public static void weigh(int status,
9        float kal_kg,  char input) {
10     float u = 1.0f; // calibration factor
11     float u_kg = 0.0f; // initial value
12     while (true) {
13       u_kg = u * kal_kg;
14       if (status == PAPER_OUT) {
15         if (input == '+') {
16           kal_kg = 1.1f;
17         }
18         if (input == '-') {
19           kal_kg = 0.9f;
20         }
21       }
22       print(u_kg);
23     }
24   }
25   public static void print(float u_kg) {...}
26 }
```

Figure 7: Simplified weighing machine controller

```
( NOT Weighing.print((1.0 * kal_kg))
∧ (input = 43)
∧ (1 = 1) )
∨
( NOT Weighing.print((1.0 * kal_kg))
∧ (input ≠ 43)
∧ (input = 45)
∧ (1 = 1) )
```

Figure 8: Path condition of Figure 7 from line 9 (`input`) to 22

for abrupt termination due to an uncaught exception [6]. Those two successors are control dependent on the call site. However, the predicate of the call site is not the result of the call but induced by the semantics of our model. In our SDG it corresponds to the term $exc \neq null$, where exc is the variable that stores the uncaught exception. Therefore, the control dependences can be viewed as summarizing the conditions which lead or do not lead to abrupt termination.[3] In a conservative approximation, these conditions can be set to *true*, assuming that both cases are feasible.

A more precise modeling retraces the conditions for abrupt termination to occur. This corresponds to generating the subconditions between the invocation node and the return value node for normal termination, and the exception node for abrupt termination, respectively. Considering Figure 6 again, the `print` statement is only executed, if `exceptionMethod` terminates normally. As we have already seen, all exceptions in `exceptionMethod` are caught, so the print statement is always executed, the PC for normal termination of `exceptionMethod` reduces to *true*.

6. Implementation and Examples

We have extended the ValSoft infrastructure [31] to include a prototype implementation of Java path conditions. For the prototype, the

[3] This is equivalent to manually checking for error codes in C

```
1  public class PCExc {
2    int secret;
3    public static void main(String[] args) {
4      System.out.println(excMethod(1));
5    }
6    public static int excMethod(int check) {
7      try {
8        PCExc pce = new PCExc();
9        pce.secret = check;
10       throw pce;
11     } catch (PCExc pce) {
12       return pce.secret;
13     }
14 }}
```

```
1   exc_0 instanceof PCExc
2 ∧ (exc_0 ≠ null)
```

Figure 9: Illicit information flow through an exception and corresponding path condition

Java SDG was adapted to interface with the C path condition generator. Thus the procedural Java constructs are tackled by the existing path condition machinery. For the object-oriented constructs, we concentrated on dynamic dispatch — the most characteristic feature of object oriented programming — according to equation (14). At the time of this writing, the precise conditions for `instanceof` and exceptions are not yet integrated and are approximated conservatively; the same is true for some other Java constructs. The precise formulae for these constructs will be integrated in the near future. Still, all conditions presented in this section have been generated by the current prototype.

First, consider the example in Figure 7, which is a simplified version of a program used in previous work [13] and does not use dynamic dispatch. The example shows that procedural Java constructs are handled quite similar to the C case. Thus the path condition in Figure 8 naturally supports standard Java and finds the illicit paths from the keyboard buffer to the printed weight. The first line says that the `print` method may not throw an exception, so the `while` loop may execute another time (The current implementation does not yet replace interprocedural exception handling with the corresponding summarizing path condition, as presented in section 5.1). The second line says that `input` must be the ASCII of '+', and the third that `status` equals `PAPER_OUT`. Note that values are often substituted for variables when they have been found constant by the SSA-form.

As a second example, we examined the program from Figure 3 with our prototype which yields

$$PC(\texttt{sel}_8, \texttt{result}_{12}) \equiv (\texttt{new class A}) \ : \ \texttt{B}$$

Since `A` is not a subtype of `B`, this condition is not satisfiable, and the PDG contains no other dependence between `sel` and `result`. Thus the parameter `sel` cannot influence the return value, even though the program slice says so.

Figure 9 demonstrates precise exception handling. The program indirectly transmits a secret value via a caught exception, which is then made public by printing it to the screen. This illicit flow is detected by the path condition, which checks information flow between line 6 (`check`) and the return value in line 12. The first line represents the catch block that only accepts a `PCExc`, and the second line requires this exception not to be `null`. Both conditions are always true in this catch block, so the path condition reduces to *true*, illustrating that the illicit flow will always take place.

```
1  public class A {
2    public int foo(int x) {
3      if (x < 17) {
4        return x;
5      } else {
6        return 0;
7      }
8    }
9  }
10 public class B extends A {
11   public int foo(int x) {
12     if (x > 42) {
13       return x;
14     } else {
15       return 0;
16     }
17 }}
18 public class SolvableDynDispatch {
19   int main_decd;
20   int main_inp;
21   public static void main(...) {
22     System.out.println(
23       dynDisp(main_decd, main_inp));
24   }
25   static int dynDisp(int dcd,int in){
26     A dynamic;
27     int result;
28     if (dcd == 1) {
29       dynamic = new A();
30     } else
31       dynamic = new B();
32     result = dynamic.foo(in);
33     return result;
34 }}
```

Figure 10: Example for dispatch

The last program in Figure 10 illustrates exploitation of backward slices for dynamic dispatch of the method call in line 32. The condition computed by our tool between `in` and `result` on this line is shown in Figure 11. It is determined based on the detailed path condition presented in section 4.2. Two cases are possible for the virtual bound method: Either the target object has type `A` (upper case), or `B` (lower case). A third case contains conflicting terms and is therefore omitted in Figure 11 as it will be removed by automatic constraint solvers. A closer examination of the conditions reveals: Line 1 stems from the condition in `A.foo`, the next two lines show the subcondition from dynamic binding: line 3 stems from line 28. The term after the disjunction represents the analogue condition for a `B` object.

This condition can easily be transformed to input values that trigger output of the second input value by a constraint solver, e.g 1,16 for an `A` or 0,43 for a `B` object. These inputs can serve as a witness for information flow from the input to the result.

Generating this path condition takes less than a second on a standard PC. Although the implementation of precise conditions for Java features has just begun, we expect that generating Java path conditions takes about the same time as traditional ones. Empirical evaluation in previous work has shown that this is possible within a few minutes even for larger programs [31].

7. Related Work and Future Directions

Information flow control analyses today are mainly based on non-standard type systems. For object-oriented languages, only Jif [25] can handle a Java-like language, but Jif programs and Java programs are not compatible and need manual conversion. The anno-

```
1 ( (17 > x)
2 ∧ dynamic = new class A
3 ∧ (dcd = 1) )
4 ∨
5 ( (x > 42)
6 ∧ dynamic = new class B
7 ∧ (dcd ≠ 1) )
8 ∨ ...
```

Figure 11: Excerpt of path condition for Figure 10

tation effort and the amount of restrictions imposed by Jif make this task expensive. Apart from that, while type checking is efficient, type systems like Jif are less precise than program slicing, amongst others due to missing flow- and context-sensitivity, which may lead to many spuriously reported violations.

Taint analysis is an important branch of integrity checking concerned with tracking user input in programs. Current implementations like [22] find critical bugs related to missing user input validation. However, these approaches focus on direct value flow, ignoring some or all implicit flow through control dependence.

Symbolic execution is a technique that executes a program with symbols instead of concrete values for the parameters. During execution, a predicate Φ (initially *true*) is built that constrains the values. When a branch is taken, the predicate is updated to reflect the condition for that branch. As it represents the conditions for taking the current path through the CFG, it is often called *path condition* as well, however as it only represents the condition for taking a specific path, it is more like an execution condition in our work. Today's systems for symbolic execution are mainly based on theorem provers or model checking. Theorem prover based systems like in ESC/Java [10] require manual annotations to generate verification conditions, while our path conditions analyze a given chop fully automatically. Other systems rely on model checkers e.g. [8] which are semi-automatic but need to cope with state explosion. Most symbolic execution systems perform a per-method analysis only, while our approach automatically generates precise interprocedural path conditions.

Recent work by Jhala has been focusing on path slicing [19]. It takes as input one particular path in the CFG and eliminates all the operations that are irrelevant towards the reachability of the target location. The result is a condition for the reachability of the target location, its infeasibility is sufficient for the infeasibility of the path. The technique does not work on the PDG but on the CFG only. It has shown effective for elimination of counterexamples provided by the model checker Blast. For our application this approach does not seem beneficial as it needs to check every single path on its own, while path conditions produce a necessary condition for all paths between two statements and share common subterms.

Parametric program slicing [9] allows specification of constraints over the program's input. A term rewriting system extracts a program slice satisfying these constraints. Conditioned program slicing [5] is a similar technique that slices based on a first order logic formula on the input variables. The conditioned slice is based on deleting statements while preserving the program's behavior. Both approaches differ from path conditions in that they do not determine input values but take them as input. In contrast, path conditions provide a logic formula that must be satisfied for an information flow to be feasible. Constraint solvers reduce this condition to input values that satisfy the formula.

Dynamic Path Conditions [13] present two possibilities to refine a static path condition: First, dynamic slicing (more precisely chopping) greatly reduces the number of control paths. As these paths form the basis of the path condition computation, the number of subclauses diminishes. Second, the variable trace is transformed to a logical formula, the *restrictive clause*, which is conjunctively linked to the path condition. The restrictive clause can be constructed even in the case that the variable trace is fragmentary, which allows using a flight recorder principle, where only the last N events are stored. Combining these methods yields the greatest effect. Hybrid approaches do not suffer as much from imprecision as purely static analyses but need to cope with the vast amount of trace data that dynamic approaches produce, and yield valid results only for one execution path, so they are more adequate for post-mortem analyses, when the deadly failure needs to be determined. We plan to incorporate dynamic path conditions into this work as this would allow — for example — to resolve dynamic binding into a single target, or `instanceof` expressions into a boolean constant without requiring subconditions.

Boolean path conditions as presented in this work and in [30, 27, 26, 31] cannot express temporal properties. For example, they cannot express that it is necessary for a specific flow that a loop condition holds and *later* it does no longer, such that the loop terminates. Boolean conditions become conservative when analyzing loops and conditions that involve loop variables. A recent approach by Lochbihler and Snelting [24] extends path conditions with temporal logic to circumvent these imprecisions. Witnesses are created by model checking instead of constraint solvers.

Another important language feature of Java has not been addressed in this work, namely concurrency. Two approaches have been proposed in [31] to generate conditions in the presence of concurrency: Interference dependence (inter-thread data dependence) can either be treated like usual data dependence. However, since interference is not transitive, this will result in overly conservative conditions. Alternatively, only possible program executions, so-called *threaded witnesses*, are considered valid paths for path condition computation, which requires more expensive slicing and chopping algorithms. An evaluation of precise concurrent slicing algorithms can be found in Giffhorn and Hammer's work [11].

Path conditions can be applied to several areas of software engineering like program understanding or test case generation. But our main goal was and is to apply them to security-critical software to statically enforce a given IFC security policy. We expect them to be especially valuable for providing precise conditions under which declassification of sensitive data is required.

Regarding the correctness of path conditions, Snelting et al. [31] does not provide a correctness proof for their definition. However, they provide a strong empiric validation that supports trust in its soundness. Nevertheless, a new project — Quis Custodiet [1] — is about to prove the soundness of IFC based on program slicing [34] and path conditions using an interactive theorem prover. Path conditions for Java, as presented in this paper, will be included in this proof.

8. Summary

Path conditions make slicing-based IFC much more precise and in particular give precise conditions for illicit information flow. We presented the first fully static approach to precise path conditions for Java. Our fundamental idea is to eliminate dynamic type checks and dynamic dispatch by the backward slice of the involved variables. Our technique allows automatic constraint solvers to reduce the derived path conditions to input values which make an illicit flow visible.

The feasibility of our method has been shown by applying our prototype implementation to several examples. Our work is not finished at this point: A full implementation must transform dynamic type tests for all Java language features, and test the scalability on bigger case studies.

Acknowledgments

We would like to thank Torsten Robschink and Jens Krinke for help on extending their implementation of Valsoft. Andreas Lochbihler and the anonymous reviewers provided useful comments.

Appendix: Proof of equivalence for equations (12) and (13)

Proposition. Let $o \neq \text{null}$. Then

$$\bigwedge_{i=1}^{n}(o : \gamma_i \Rightarrow PC_{\gamma_i.\mathtt{f}}(x,y)) \equiv \bigvee_{i=1}^{n}(o : \gamma_i \wedge PC_{\gamma_i.\mathtt{f}}(x,y))$$

Proof. 1. "⇒": Let $\bigwedge_{i=1}^{n}(o : \gamma_i \Rightarrow PC_{\gamma_i.\mathtt{f}}(x,y))$. As exactly one of the potential target methods of dynamic dispatch will be executed, we know $\exists!\ k.\ o : \gamma_k$ (where γ_k is the run-time type of o). Thus from the premise we conclude $PC_{\gamma_k.\mathtt{f}}(x,y))$. Hence $o : \gamma_k \wedge PC_{\gamma_k.\mathtt{f}}(x,y))$,[4] therefore $\bigvee_{i=1}^{n}(o : \gamma_i \wedge PC_{\gamma_i.\mathtt{f}}(x,y))$.

2. "⇐": Let $\bigvee_{i=1}^{n}(o : \gamma_i \wedge PC_{\gamma_i.\mathtt{f}}(x,y))$. As above we know that $\exists!\ k.\ o : \gamma_k$. Then also $\exists!\ k.\ o : \gamma_k \wedge PC_{\gamma_k.\mathtt{f}}(x,y)$ holds. Now let $i \in 1..n$. If $i \neq k$, then $\neg(o : \gamma_i)$, thus the implication $o : \gamma_i \Rightarrow PC_{\gamma_i.f}(x,y)$ holds trivially. If $i = k$, by assumption $PC_{\gamma_i.\mathtt{f}}(x,y)$ holds and hence the implication $o : \gamma_i \Rightarrow PC_{\gamma_i.\mathtt{f}}(x,y)$.[5] Thus $\bigwedge_{i=1}^{n}(o : \gamma_i \Rightarrow PC_{\gamma_i.\mathtt{f}}(x,y))$. □

References

[1] Quis custodiet. http://pp.info.uni-karlsruhe.de/project.php/id=31 funded by DFG Sn11/10-1.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM Press.

[3] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *European Symposium On Research In Computer Security (ESORICS '07)*, 2007.

[4] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 361–377, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[5] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11–12):595–607, Dec 1998. Special issue on Program Slicing.

[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] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

[8] X. Deng, Robby, and J. Hatcliff. Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. In *SEFM '07: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, pages 273–282, Washington, DC, USA, 2007. IEEE Computer Society.

[9] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–392, New York, NY, USA, 1995. ACM Press.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[11] D. Giffhorn and C. Hammer. An evaluation of precise slicing algorithms for concurrent programs. In *SCAM'07: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 17–26, Paris, France, Sept. 2007.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley Professional, 3rd edition, 2005. `http://java.sun.com/docs/books/jls/`.

[13] C. Hammer, M. Grimme, and J. Krinke. Dynamic path conditions in dependence graphs. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 58–67, New York, NY, USA, 2006. ACM Press.

[14] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *Proc. Second International Symposium on Leveraging Application of Formal Methods, Verification and Validation (ISoLA 2006)*, Paphos, Cyprus, Nov. 2006.

[15] 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.

[16] C. Hammer and G. Snelting. An improved slicer for java. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22, New York, NY, USA, 2004. ACM Press.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[18] 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*, pages 79–90, New York, NY, USA, 2006. ACM Press.

[19] R. Jhala and R. Majumdar. Path slicing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–47, New York, NY, USA, 2005. ACM Press.

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

[21] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Prentice Hall, 2nd edition, Apr. 1999.

[22] B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Baltimore, Maryland, August 2005.

[23] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for java. In *Proceedings of the 3rd Asian Symposium (APLAS 2005)*, volume 3780 of *LNCS*, pages 139–160, Tsukuba, Japan, November 2005.

[24] A. Lochbihler and G. Snelting. On temporal path conditions in dependence graphs. In *SCAM'07: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 49–58, Paris, France, Sept. 2007.

[25] 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*, pages 228–241, New York, NY, USA, 1999. ACM Press.

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

[4] Note that in general $((A \Rightarrow B) \wedge A) \Rightarrow (A \wedge B)$

[5] Note that in general $(A \wedge B) \Rightarrow (A \Rightarrow B)$.

[27] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 478–488, New York, NY, USA, 2002. ACM Press.

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

[29] 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*, pages 355–364, San Diego, CA, January 1998.

[30] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 332–348, London, UK, 1996. Springer-Verlag.

[31] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[32] M. Sridharan, S. J. Fink, and R. Bodík. Thin slicing. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 112–122, New York, NY, USA, 2007. ACM Press.

[33] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM Press.

[34] D. Wasserrab and A. Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In *Proceedings of the 21st International Conference of Theorem Proving in Higher Order Logics*, Montréal, Québec, Canada, August 2008. Springer.