On PDG-Based Noninterference and its Modular Proof

Daniel Wasserrab Denis Lohner Gregor Snelting *

Universität Karlsruhe (TH)

{wasserra,lohner,snelting}@ipd.info.uni-karlsruhe.de

Abstract

We present the first machine-checked correctness proof for information flow control (IFC) based on program dependence graphs (PDGs). IFC based on slicing and PDGs is flow-sensitive, context-sensitive, and object-sensitive; thus offering more precision than traditional approaches. While the method has been implemented and successfully applied to realistic Java programs, only a manual proof of a fundamental correctness property was available so far.

The new proof is based on a new correctness proof for intraprocedural PDGs and program slices. Both proofs are formalized in Isabelle/HOL. They rely on abstract structures and properties instead of concrete syntax and definitions. Carrying the correctness proof over to any given language or dependence definition reduces to just showing that it fulfills the necessary preconditions, thus eliminating the need to develop another full proof.

We instantiate the framework with both a simple while language and Java bytecode, as well as with three different control dependence definitions. Thus we obtain 6 IFC correctness proofs for the price of $1\frac{1}{2}$.

Categories and Subject Descriptors 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, verification

General Terms Languages, Security, Verification

Keywords Program Slicing, Program Dependence Graph, Noninterference, Correctness Proof, Modularity

1. Introduction

"Quis custodiet ipsos custodes? Who will guard the guards?" When Juvenal posed this question about 100 A.D., he did not yet know about language-based security and information flow control (IFC). Today, correctness of IFC algorithms is a topic of burning pressure. It must be demonstrated that fulfillment of an algorithm's specific criterion implies security, where the latter is usually expressed as some form of noninterference. But most IFC algorithms come with a manual proof only, and some are published without any proof. As manual proofs for such complex algorithms are notoriously error-prone, machine-checked correctness proofs have gained popularity. The seminal Volpano/Smith security type system [31], e.g., originally had a manual correctness proof; recently, two different machine checked proofs for (variations of) the Volpano/Smith system were published, demonstrating in detail that the original proof was correct [34, 6].

Volpano/Smith and its successors opened the door for language-based IFC and security type systems. But most type systems are not flow-sensitive, context-sensitive, let alone object sensitive. This can lead to a loss of precision and false alarms. Another problem of some IFC proposals is scalability with respect to realistic languages and programs: not all methods can handle e.g. full Java and 10kLoC, and some require excessive program annotations.

We thus argued that IFC must better exploit modern program analysis, and proposed to build language-based IFC on top of program dependence graphs (PDGs) and slicing. While not on everybody's radar, PDGs and slicers have, after 25 years of research, become a sophisticated instrument exploiting all the achievements of modern program analysis, and can be applied to realistic languages and programs; commercial implementations of slicers for C are available.

Our PDG-based approach to IFC has been described in [32, 33, 14, 12, 15]. In particular, we developed a precise PDG for full Java bytecode [16, 13]. The forthcoming journal article [15] explains in detail how PDG-based IFC works, why it scales, why it is precise, why it is correct, and how it handles declassification; in addition [13] describes implementation, GUI, and case studies.

This work is however based on a fundamental assumption: it relies on correctness of PDGs and slicing, and builds on a theorem connecting PDGs and noninterference which was proved only manually in [33], and only for Goguen/Meseguer style noninterference. In fact, no machine-checked proof of slicing correctness was available so far, let alone proofs for PDG-based noninterference.

* This work was supported by DFG grant Sn11/10-1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLAS'09 June 15, 2009, Dublin, Ireland.

Copyright © 2009 ACM. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proc. PLAS'09.

In the current paper, we provide the first machine-checked correctness proof for PDG-based static intraprocedural program slicing of single-threaded programs. We then provide a machine-checked proof that our slicing-based IFC criterion – first introduced in [33] and elaborated in [15] – implies low-deterministic security. Both proofs are formalized in Isabelle/HOL [25]. A remarkable feature of the proofs is the separation of language-dependent parts from language-independent parts, where the latter only deals with fundamental properties of CFGs, PDGs, and slices. This separation allows to instantiate the basic proof with different languages and their semantics, considerably simplifying the correctness proof for yet another language. We first instantiated the framework with a simple while language, and then instantiated it with the Jinja [19] definition of Java bytecode. Next, we parameterized the framework with respect to the exact definition of control dependences, since various definitions have appeared in the literature.

Thus we had to provide one complex correctness proof for the framework, and then provide several proofs that the specific languages and definitions stick to the required properties and thus can be plugged in. For the while language and the dependence definitions, this was straightforward, but for Jinja bytecode, the required "auxiliary proofs" are about half as long as the main proof. Hence we eventually obtained 6 correctness proofs for, roughly, the effort for $1\frac{1}{2}$ traditional machine-checked proofs.

The main technical contributions of our work are:

- an augmentation of our existing framework for dynamic slicing (see [40]) with the first formalization of static intraprocedural slicing in a proof assistant;
- a correctness proof for static intraprocedural slicing (where correctness is much stronger than for dynamic slicing), which is independent of a specific language, and independent of a concrete control dependence definition;
- instantiations of the framework with a simple While as well as with a quite sophisticated object-oriented byte code language;
- instantiations of the framework with three different definitions of control dependence;
- the first proof that slicing-based IFC guarantees low-deterministic security.

Thus we provide a reliable basis for the verification of slicing and its applications, e.g. for applications in software security.

Overview of the paper. We will first recapitulate noninterference, slicing and control dependence in PDGs. We describe in detail the slicing framework formalized in Isabelle (available online, see [39]), and explain the machine-checked correctness proof for static intraprocedural slicing. Based on this result, we formalize low-deterministic security in Isabelle and prove that it is guaranteed by our PDG-based IFC criterion. We proceed to describe the instantiation of the framework with 2 language definitions and, independently, 3 control dependence definitions, obtaining 6 proofs altogether. Discussion of related and future work shows that the extension of our result to interprocedural slicing and the precise interprocedural algorithm from [15] (which includes declassification) will not only require major additional proof efforts, but perhaps also an extension of the traditional notion of noninterference.

2. Slicing Correctness and Noninterference

2.1 Slicing

Given a certain point in a program, a slice collects all statements that can influence this point. Slicing proved to be useful for many applications, e.g. debugging [35], testing [7], reducing communicating automata specifications [22], and software security algorithms [15, 33]. Today, commercial slicing tools such as Codesurfer [2] are routinely used for some of these tasks.

Most slicing tools are based on the PDG or its interprocedural extension, the system dependence graph [17]. PDG nodes correspond to the program statements, and connect them with *data dependences* and *control dependences*. The *static intraprocedural backward slice* of a given program point (i.e. a specific PDG node, called the slicing criterion) is defined as the set of all nodes on which the point is transitively data or control dependent.[1] This set is a conservative approximation of all statements that can influence the slicing criterion. Note that for realistic languages, PDG generation and precise slicing is absolutely nontrivial. Hundreds of papers on slicing have been published in the last two decades, for an overview, see Krinke [21].

2.2 Noninterference

Language-based IFC analyses the program source to discover security leaks, and usually aims to establish noninterference. Informally, noninterference demands that variation in secret variables will not result in variations of public output, thus guaranteeing confidentiality. The most fundamental noninterference definition is *low-deterministic security*:

Let H, L be the secret (high) and public (low) confidentiality levels. For a program statement c, $[\![c]\!]$ denotes the state transformation induced by executing c. Two states s, s' are low-equivalent (written $s \approx_L s'$) if they coincide on variables with confidentiality level L. Low-deterministic security then demands that $\forall s, s'.\ \ s \approx_L s' \implies [\![c]\!]s \cong_L [\![c]\!]s'$. Note that this definition only talks about the initial and final program states; low-deterministic security cannot express security-relevant properties of intermediate states. More elaborate definitions of noninterference exist, such as possibilistic noninterference or probabilistic noninterference.

Our correctness theorem can informally be stated as follow: If the backward slice of all low variables does not contain high variables, then the low-deterministic security property is satisfied and hence the program is secure. In fact, it

[1] Also for dependence relations that do not constitute a PDG, e.g. because they are not binary, the backward slice is computed this way.

is enough to require that backward slices of *final* low variables do not contain *initial* high values, ignoring intermediate states and variables.

The converse of the theorem does not hold, because all implementations of noninterference based on program analysis, including our own, are conservative approximations; that is, they can generate false alarms. The more precise an analysis is, the less false alarms it will generate. To illustrate precision, consider the program

```
procedure swap (var x, y: integer) {
   integer temp = x; x = y; y = temp; }
swap (L1, L2);
swap (H1, H2);
```

It swaps two L values as well as two H values using the same auxiliary method. The program is perfectly secure in the sense of noninterference, but a context-insensitive analysis will generate a false alarm, as it does not distinguish the two calling contexts and thus believes that a H value is assigned (i.e. leaked) to an L variable. Furthermore, some systems require an annotation for `temp` with either H or L, which generates another false alarm. Exploiting modern program analysis will avoid such problems, as modern analysis and slicing is flow-sensitive, context-sensitive, and object sensitive. Still, 100% precision is impossible due to decidability problems.

2.3 Outline of correctness proof

Despite its popularity, few correctness results for slicing exist, e.g. Reps and Yang [28] or Amtoft [1]. Both proofs are based on specific programming languages; hence, algorithms using slicing can only be verified for those languages – a needless restriction as slicing itself is independent of the underlying language. Furthermore, existing proofs only consider one specific definition of control dependence. As more recent, different control dependence definitions as in [27] infer different slices, replacing the standard definition demands a new proof, which is nontrivial.

In order to abstract away from specific languages and control dependences, we present a formalization of static intraprocedural slicing in a highly modularized framework. The full proof is available online [39]. Following Amtoft [1], correctness is stated as a weak simulation property between nodes and states in the original and sliced control flow graph. The proof is based on an abstract control flow graph representation with certain structural and well-formedness conditions, but not restricted to a specific language. The proof is also independent of a specific control dependence definition, but requires it to have one particular property.

Adapting the proof to another language just requires to show that the control flow graph of this language can be embedded into the abstract one by fulfilling all necessary conditions. Likewise, changing the control dependence definition reduces to showing that the one property required holds. Hence future verifications of algorithms basing on this correctness proof immediately gain a high level of robustness.

3. Dependences in PDGs

3.1 Data Dependence

The data dependence definition is based on *Def* and *Use* sets for every node. All variables defined (e.g. assigned) in a statement are in the Def set of the respective node, those which are used (e.g. in a calculation) in its Use set. A node n' is data dependent on a node n, if there is a variable V in the Def set of n which is also in the Use set of n' and there is a CFG *path* (sequence of edges) from n to n' such that the variable is not defined in any other node on this path.

3.2 Control Dependence

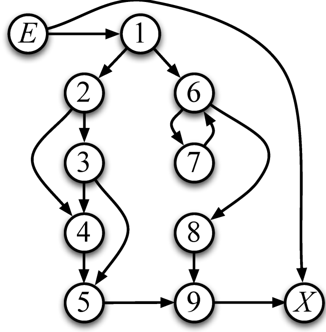
Control dependence captures the effect that nodes can influence whether the control flow reaches other nodes. The definition of Ferrante et al. [9] – one of the first formal definitions – is most widespread and is viewed as a standard.

Nevertheless, this is an area of active research, see Ranganath et al. [27] fo a recent overview. Each control dependence definition serves a different purpose, so the choice of the control dependence affects the semantics of the slice. In this paper we focus on three different kinds of control dependences: (i) *standard control dependence* as it has been in use for years, (ii) *weak control dependence* as defined by Podgurski and Clarke [26], and (iii) *weak order dependence* as defined by Amtoft [1]. The first two are binary relations, so we get program dependence graphs with them, for the third one, a ternary relation, this does not hold. We chose these three definitions to illustrate the flexibility of our modularized proof as they differ in many details. In §5.2 we show in more detail how these as well as other possible control dependence definitions can be "plugged" into our framework.

We now look more closely at each of these three dependence relations, state their definitions informally and illustrate them using the CFG in Fig. 1 where E denotes the entry and X the exit node. Note that the subgraph built of nodes 2 to 5 does not describe the control flow of a structured program; however, it still is a valid CFG in our framework.

Standard Control Dependence. Usually, a node n' is regarded as control dependent on node n if selecting an outgoing edge of n in the CFG affects whether n' is reached; e.g. all nodes in the branches of an if-statement are dependent on the node representing the if-predicate. To define standard control dependence (SCD) we first need to state *postdomination* using a unique reachable exit node. A node n' postdominates node n if every path from n to the exit node contains n'. In the table in Fig. 1 we list the postdominators for every node n, i.e. the set of all nodes n' which postdominate n.

Ferrante et al. [9] stated that a node n' is control dependent on a node n (written $n \longrightarrow_{scd} n'$), if n' postdominates all nodes on a path in the CFG between n and n' but not n. However, we regard an equivalent definition (see the lemma in §5.2) from Wolfe [41] as more suitable, where a node n' is control dependent on a node n, if n has at least two suc-



	postdominators	strong postdominators
E	$\{E\}$	$\{E\}$
1	$\{1,9\}$	$\{1\}$
2	$\{2,5,9\}$	$\{2,5,9\}$
3	$\{3,5,9\}$	$\{3,5,9\}$
4	$\{4,5,9\}$	$\{4,5,9\}$
5	$\{5,9\}$	$\{5,9\}$
6	$\{6,8,9\}$	$\{6\}$
7	$\{6,7,8,9\}$	$\{6,7\}$
8	$\{8,9\}$	$\{8,9\}$
9	$\{9\}$	$\{9\}$

Figure 1. Example CFG and postdominators

cessors, one postdominated by n', while the other one is not. Thus, the example in Fig. 1 implies the following SCDs:

$$E \longrightarrow_{scd} 1, E \longrightarrow_{scd} 9, 1 \longrightarrow_{scd} 2, 1 \longrightarrow_{scd} 5, 1 \longrightarrow_{scd} 6,$$
$$1 \longrightarrow_{scd} 8,\ 2 \longrightarrow_{scd} 3,\ 2 \longrightarrow_{scd} 4, 3 \longrightarrow_{scd} 4,\ 6 \longrightarrow_{scd} 7$$

Weak Control Dependence. Nonterminating loops prevent nodes after the loop from being executed, a fact that cannot be covered with SCD. To capture this effect, control dependence edges between those nodes and the loop predicate are often desired; this is the concept of weak control dependence (WCD). It uses the notion of *strong postdomination*, where no loops[2] on any path between a node and its postdominator are allowed. Otherwise, there would be an infinite path always running through the loop but never reaching the postdominator. Therefore, we define that n' strongly postdominates n if n' postdominates n and there is no loop on any path between n and n'.

WCD itself is then defined analogously to standard control dependence, just replacing postdomination with strong postdomination. We write $n \longrightarrow_{wcd} n'$ if n' is weak control dependent on n. In Fig. 1 the following holds: $E \longrightarrow_{wcd} 1$,

$$1 \longrightarrow_{wcd} 2, 1 \longrightarrow_{wcd} 5, 1 \longrightarrow_{wcd} 6, 1 \longrightarrow_{wcd} 9, 2 \longrightarrow_{wcd} 3,$$
$$2 \longrightarrow_{wcd} 4, 3 \longrightarrow_{wcd} 4, 6 \longrightarrow_{wcd} 7, 6 \longrightarrow_{wcd} 8,\ 6 \longrightarrow_{wcd} 9$$

Note that 8 and 9 now depend on other nodes than before and that due to the loop at node 6, both nodes immediately after this loop (i.e. 8 and 9) are now dependent on 6. As an example where this and the previous definition of control dependence behave differently, regard the set of all nodes from which 8 is transitively control dependent, i.e. the set closed under the respective control dependence. For SCD, this set is $\{E, 1, 8\}$, for WCD however we get $\{E, 1, 6, 8\}$.

[2] Statically, we must assume that any loop may be nonterminating.

Weak Order Dependence. The advantage of weak order dependence (WOD) is that there is no need for a unique reachable end node, as is the case in e.g. reactive systems. Unlike the former two, WOD is not a binary relation, but a set of triples of nodes. Intuitively, two nodes are weak order dependent on another node, if the latter node controls the order in which the other two nodes are executed (which includes that one of these nodes may never be executed). However, the definition is a bit more complicated: we say that two nodes n_1 and n_2 are weak order dependent on node n (all three nodes distinct), written $n \longrightarrow_{wod} n_1,n_2$, if (i) n can reach n_1 in the CFG without visiting n_2, (ii) n can reach n_2 without visiting n_1 and (iii) there exists an immediate successor m of n, such that either (a) m can reach n_1 and all paths from m to n_2 contain n_1 or (b) m can reach n_2 and all paths from m to n_1 contain n_2.

Since there are many WOD triples for the example CFG in Fig. 1, we present them as a set of node pairs which are weak order dependent on the node preceding this set, leaving out all tuples where the two components are swapped or where one component is the exit node:

1: $\{(2, 6), (2, 7), (2, 8), (2, 9), (3, 6), (3, 7), (3, 8), (3, 9),$
$(4, 6), (4, 7), (4, 8), (4, 9), (5, 6), (5, 7), (5, 8), (5, 9),$
$(6, 9), (7, 9), (8, 9)\}$, 3: $\{(4, 5), (4, 9)\}$,
2: $\{(3, 4), (3, 5), (3, 9), (4, 5), (4, 9)\}$, 6: $\{(7, 8), (7, 9)\}$

While the number of node triples that are weak order dependent in this example is huge compared to the cardinality of the former two dependence relations, sets closed under WOD[3] may be smaller. E.g. the smallest set which contains 8 and is closed under WOD consists only of the element 8, because its WOD predecessor (1 or 6) may only be in this set if the other matching WOD successor of 1 (2, 3, 4, 5, or 9) or 6 (7) was in it.

4. The Framework

To provide a framework which allows us to formalize and prove properties of slicing regardless of the actual programming language, we need a structure which includes an effectual representation of the program and comprises all information essential for slicing: the control flow graph. Thus, our starting point for the formalization is the definition of an abstract CFG (i.e. without considering features specific for certain languages). By doing so we ensure that our framework is as generic as possible since all proofs in it hold for every language whose CFG conforms to this abstract CFG. The framework is entirely formalized in the proof assistant Isabelle/HOL [25], including all lemmas and theorems, i.e. every proof is machine-checked. Definitions and lemmas taken from Isabelle are written *small and slanted.*

[3] A set is closed under WOD if for any two nodes in this set that are weak order dependent on a node n, n is also in the set.

4.1 Notation

Types include the basic types of truth values, natural numbers and integers, which are called *bool*, *nat*, and *int* respectively. The space of total functions is denoted by $\Rightarrow$. Type variables are written *'a*, *'b*, etc. $t{::}\tau$ means that the HOL (*Higher Order Logic*) term t has HOL type τ.

Sets (type *'a set*) follow the usual mathematical convention. Function *card* returns the cardinality of a finite set. *Lists* (type *'a list*) come with the empty list [], the infix constructor $\cdot$, the infix @ that concatenates two lists, and the conversion function *set* from lists to sets. Variable names ending in "s" usually stand for lists. If $i < |xs|$ then $xs_{[i]}$ denotes the i-th element of *xs*. The function *map*, which applies a function to every element in a list, is also available.

$[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$ abbreviates $P_1 \Longrightarrow (\ldots \Longrightarrow (P_n \Longrightarrow Q)\ldots)$ and is often displayed as inference rule.

4.2 Locales in Isabelle

Locales in Isabelle [3] provide the means to modularize proofs. Within a locale, one can introduce (*fix*) definitions and functions by stating their signature which may also contain type variables. To impose certain constraints on these definitions one has to *assume* that the respective statement holds. When defining new functions or proving lemmas within the locale one can then use these fixed definitions and the assumed constraints. One or multiple locales can also be extended by a new locale with additional definitions and constraints. All the definitions and lemmas proved in the base locales are available in the extended locale.

As a short example, consider this definition of semigroups where we define an operator $\odot$, whose signature depends on the type variable *'a*, and state that this operator is associative by the fact named *assoc*. Defining a new locale *semi-comm* which extends *semi* and requires a commutative operator is also straightforward:

locale *semi* = **fixes** *prod* :: $'a \Rightarrow 'a \Rightarrow 'a$ — written $\odot$
  assumes *assoc*: $(x \odot y) \odot z = x \odot (y \odot z)$

locale *semi-comm* = *semi* + **assumes** *comm*: $x \odot y = y \odot x$

4.3 The Abstract Control Flow Graph

Fig. 2 depicts the definition of a CFG as a locale, on which we will take a closer look in the following. The abstract CFG consists of nodes of type *'node* and edges of type *'edge*, with an edge a being in the set of CFG edges if it fulfills some property *valid-edge a*, a parameter of the instantiating language. A node n is in the node set of a CFG if it fulfills the property *valid-node n*, which is not assumed but defined in the locale, stating that n is the source or target node of a *valid-edge*: *valid-node n* $\equiv \exists a.$ *valid-edge a* $\wedge$ (n = *src a* $\vee$ n = *trg a*). Functions *src*, *trg* and *kind* determine the source node, target node and edge kind of an edge, respectively. Edges carry semantic information, the edge kind states the action taken when traversing this edge. We have two edge kinds of type *'state edge-kind*, both parameterized with a state type variable *'state*: updating the current state with a function $f{::}'state \Rightarrow 'state$, written $\Uparrow f$, or assuring that a predicate $Q{::}'state \Rightarrow bool$ in the current state is fulfilled, written $(Q)_{\surd}$. To traverse edges in a state s, we define function *transfer* to update the state accordingly to the edge kind, and function *pred* to check that the respective edge kind predicate holds:

locale *CFG* =
  fixes *valid-edge* :: *'edge* $\Rightarrow$ *bool*
  fixes *src* :: *'edge* $\Rightarrow$ *'node*
  fixes *trg* :: *'edge* $\Rightarrow$ *'node*
  fixes *kind* :: *'edge* $\Rightarrow$ *'state edge-kind*
  fixes *Entry* :: *'node* — written (*-Entry-*)
  assumes *Entry-target*: *valid-edge a* $\Longrightarrow$ *trg a* $\neq$ (*-Entry-*)
  and *no-multi-edges*: $[\![$*valid-edge a*; *valid-edge a'*;
    src a = *src a'*; *trg a* = *trg a'*$]\!] \Longrightarrow a = a'$

locale *CFGExit* = *CFG* +
  fixes *Exit* :: *'node* — written (*-Exit-*)
  assumes *Exit-source*: *valid-edge a* $\Longrightarrow$ *src a* $\neq$ (*-Exit-*)
  and *Entry-Exit-edge*: $\exists a.$ *valid-edge a* $\wedge$ *src a* = (*-Entry-*) $\wedge$
    trg a = (*-Exit-*) $\wedge$ *kind a* = $(\lambda s.\ False)_{\surd}$

Figure 2. Locale defining the structure of the abstract CFG

$$transfer\ \Uparrow f\ s \;=\; f\ s, \qquad transfer\ (Q)_{\surd}\ s \;=\; s$$
$$pred\ \Uparrow f\ s \;=\; True, \qquad pred\ (Q)_{\surd}\ s \;=\; Q\ s$$

We assume an (*-Entry-*) node, which may not have incoming edges. Also we do not allow multi-edges, i.e. if the source and target nodes of two valid edges coincide, so do the two edges.

Edges can also be combined to paths: $n\ {-as{\rightarrow}*}\ n'$ denotes that node n can reach n' via edges *as*::*'edge list*. These paths are inductively defined using these rules:

$$\frac{valid\text{-}node\ n}{n\ {-[]{\rightarrow}*}\ n}$$

$$\frac{n''\ {-as{\rightarrow}*}\ n' \qquad valid\text{-}edge\ a \qquad src\ a = n \qquad trg\ a = n''}{n\ {-a{\cdot}as{\rightarrow}*}\ n'}$$

We define *srcs*, *trgs* and *kinds* as mappings of the respective functions to edge lists using standard function *map*. We also lift *transfer* and *pred* to lists of edge kinds.

If a unique end node is required, we assume its existence in locale *CFGExit*, call it (*-Exit-*) and allow only incoming edges. We also assume a special edge from (*-Entry-*) to (*-Exit-*) of kind $(\lambda s.\ False)_{\surd}$, a predicate that can never be fulfilled. It is needed for control dependences based on postdomination to behave correctly.

After having defined the structural properties of the CFG, we furthermore need: (i) some well-formedness properties for its edges, (ii) the *Def* and *Use* sets for the valid nodes, which collect the defined and used variables in this node, respectively, and (iii) a function *state-val s V* returning the value currently stored in variable V in state s. Variables (or more generally said: locations) are of type *'var*, values of type *'val*. The formalization as locales is shown in Fig. 3, in words:

locale *CFG-wf* = *CFG* +
fixes *Def* :: *'node* ⇒ *'var set*
fixes *Use* :: *'node* ⇒ *'var set*
fixes *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
assumes *Entry-empty*: $Def\ (\text{-}Entry\text{-}) = \{\} \wedge Use\ (\text{-}Entry\text{-}) = \{\}$
and *no-Def-equal*: $[\![valid\text{-}edge\ a;\ V \notin Def\ (src\ a)]\!]$
$\Longrightarrow state\text{-}val\ (transfer\ (kind\ a)\ s)\ V = state\text{-}val\ s\ V$
and *transfer-only-Use*: $[\![valid\text{-}edge\ a;$
$\forall V \in Use\ (src\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V]\!]$
$\Longrightarrow \forall V \in Def\ (src\ a).\ state\text{-}val\ (transfer\ (kind\ a)\ s)\ V =$
$state\text{-}val\ (transfer\ (kind\ a)\ s')\ V$
and *Uses-pred-equal*: $[\![valid\text{-}edge\ a;\ pred\ (kind\ a)\ s;$
$\forall V \in Use\ (src\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V]\!]$
$\Longrightarrow pred\ (kind\ a)\ s'$

locale *CFGExit-wf* = *CFGExit* + *CFG-wf* +
assumes *Exit-empty*: $Def\ (\text{-}Exit\text{-}) = \{\} \wedge Use\ (\text{-}Exit\text{-}) = \{\}$

Figure 3. Well-formedness properties of the abstract CFG

- *Def* and *Use* sets of (*-Entry-*) (and (*-Exit-*), if defined) are empty,
- traversing an edge leaves all variables which are not defined in the source node of this edge unchanged,
- if two states agree on all variables in the *Use* set of the source node of an edge, then after traversing this edge the two states agree on all variables in the *Def* set of this node; i.e. different values in the variables not in the *Use* set cannot influence the values of the variables in the *Def* set after the semantic action,
- if two states agree on all variables in the *Use* set of the source node of a predicate edge and this predicate is valid in one state, it is also valid in the other one.

If we also have an operational semantics of the language – where $\langle c,s\rangle \Rightarrow \langle c',s'\rangle$ means that evaluating statement c in state s results in a final statement c' and final state s' – and a mapping from a node n to its corresponding statement c via n *identifies* c, we have another well-formedness property (called *semantically well-formed*):

$$\frac{n\ identifies\ c \qquad \langle c,s\rangle \Rightarrow \langle c',s'\rangle}{\exists\, n'\ as.\ n\ {-as}{\rightarrow}{*}\ n' \wedge transfers\ (kinds\ as)\ s = s' \wedge preds\ (kinds\ as)\ s \wedge n'\ identifies\ c'}$$

This property states that if the complete evaluation of statement c in state s results in a state s' and node n corresponds to statement c, then there is a path in the CFG beginning at n to a node n' that corresponds to the final statement c', on which, taking s as initial state, all predicates in predicate edges hold and the traversal of the path edge kinds also yields state s'.

5. Static Intraprocedural Slicing is Correct

We base our work on the proof by Amtoft [1], who defines the correctness of static intraprocedural slicing as a *weak simulation property* of the observable behavior of the original and the sliced program, regarding the CFG as a labeled transition system (for details see §5.1):

Correctness Property:
If an observable move is possible in the original graph, then an observable move is also possible in the sliced graph, if the respective initial nodes and states and also the resulting nodes and states of both moves are weakly similar.

This correctness property for static slicing is stronger than the one given in [40] for dynamic slicing, as it does not depend on specific input such as program runs or input states. To prove this correctness property, Amtoft defines for every node in a CFG its set of observable nodes in a given backward slice. The actual slicing is done by rewriting a given code map to return no-op statements whenever the respective node is not in the backward slice. So the "sliced graph" is just the original graph (as no nodes or edges are removed), but the effects of traversing nodes not in the backward slice are eliminated.

Even though Amtoft restricts his work to a While language and proves the correctness of slicing just for weak order dependence, this work is ideal to be included in our framework as (i) his code map conforms to applying the functions *kind* and *transfer* to the corresponding CFG edges in our framework and (ii) the characteristics of weak order dependence are just needed in exactly one lemma where Amtoft proves that the observable set for any node is at most a singleton; if one can show this property for another control dependence, the whole proof still holds for this new control dependence. Thus we go beyond Amtoft's work as we are able to eliminate the concrete language as well as the concrete control dependence definition; the next section rephrases the formalization of the correctness proof focusing on the latter abstraction.

5.1 The Correctness Proof

To state correctness for static slicing we need: (i) a formalization of a statically sliced graph, (ii) the notion of observable moves in the original and sliced graph and (iii) a weak simulation between start and end points of these moves.

The statically sliced graph. A static (backward) slice for a node n_c (the slicing node) determines which nodes are in the sliced graph. Basically, every node that potentially influences control or data flow to n_c is in the backward slice of n_c, so the slice is defined in terms of data and control dependence. Data dependence is defined as stated in §3.1:

$$n\ influences\ V\ in\ n' \equiv \exists\, a'\ as'.\ V \in Def\ n \wedge V \in Use\ n' \wedge n\ {-a'{\cdot}as'}{\rightarrow}{*}\ n' \wedge (\forall\, n''{\in}set\ (srcs\ as').\ V \notin Def\ n'')$$

To abstract from a concrete control dependence definition in the slice, we use a locale named *BackwardSlice* (see Fig. 4) to fix a function from a node to a set of nodes called *backward-slice* with properties that guarantee that the resulting node sets are indeed backward slices of the parameter node. Hence we formulate three assumptions: (i) every node is in its own *backward-slice*, (ii) if a node n' is in *backward-slice* n_c and this node is data dependent on node n, then n must

locale *BackwardSlice* = *CFG-wf* +
fixes *backward-slice* :: *'node* ⇒ *'node set*
assumes *refl*: *valid-node* $n_c \Longrightarrow n_c \in$ *backward-slice* n_c
and *dd-closed*: ⟦$n' \in$ *backward-slice* n_c; n *influences* V *in* n'⟧
$\Longrightarrow n \in$ *backward-slice* n_c
and *obs-singleton*: *valid-node* n
$\Longrightarrow$ *card* (*obs* n (*backward-slice* n_c)) ≤ 1

Figure 4. Locale abstracting from a specific backward slice

also be in *backward-slice* n_c (i.e. *backward-slice* n_c is closed under data dependence), and (iii) the set of observable nodes in *backward-slice* n_c is for every valid node at most a singleton. Only this last assumption *obs-singleton* is influenced by the control dependence used in the slice.

The observable set of node n in set S contains all nodes n' in S that can be reached via a CFG path from n such that no other node on this path is in S:

$$\frac{n \; {-as\rightarrow*} \; n' \qquad \forall\, nx{\in}set\ (srcs\ as).\ nx \notin S \qquad n' \in S}{n' \in obs\ n\ S}$$

So every node being itself in set S has the singleton observable set only containing itself. Using this definition, we can say that all nodes having the same set of observable nodes in set *backward-slice* n_c – being at most a singleton by assumption from locale *BackwardSlice* – can be regarded as "equal" from the point of view of the slicing.

Now, we define the sliced graph for n_c using its *backward-slice*. Instead of really eliminating nodes not in *backward-slice* n_c from the original graph, we just eliminate all effects of the edges leaving those nodes. To this end we define a function *slice-kind*, parameterized with the slicing node n_c, which, analogously to *kind*, maps edges to the effect this edge has on the state, i.e. its *edge-kind*. The rules are simple: if the source node of the considered edge is in *backward-slice* n_c, *slice-kind* n_c behaves just like *kind*. Otherwise, *slice-kind* n_c returns the respective no-op for this edge, $\Uparrow id$ for update, $(\lambda s.\ True)_{\surd}$ or $(\lambda s.\ False)_{\surd}$ for predicate edges. The rules defining this operation guarantee that only one predicate edge leaving a node is set to $(\lambda s.\ True)_{\surd}$, all others are set to $(\lambda s.\ False)_{\surd}$; thus we make sure not to introduce nondeterminism.

Moves in the graphs. Moves in the original and sliced graph are relations between (node,state) tuples. A move traverses edge a whose source node n either is in *backward-slice* n_c (written $n_c,f \vdash (n,s) \ {-a\rightarrow}\ (n',s')$) or is not (called τ-move, written $n_c,f \vdash (n,s) \ {-a\rightarrow_\tau}\ (n',s')$), reaching a's target node n' and state $s' = transfer\ (f\ s)$. The parameter f is replaced with *kind* if we traverse the original, with *slice-kind* n_c if we traverse the sliced graph of n_c. An observable move then consists of arbitrary many τ-moves ($=as\Rightarrow_\tau$ is the reflexive transitive closure of $-a\rightarrow_\tau$), followed by a $-a\rightarrow$ move:

$$\frac{n_c,f \vdash (n,s) =as\Rightarrow_\tau (n',s') \qquad n_c,f \vdash (n',s') -a\rightarrow (n'',s'')}{n_c,f \vdash (n,s) =as\ @\ [a]\Rightarrow (n'',s'')}$$

The weak simulation. We define two (node,state) tuples to be weakly similar (i.e. in relation $WS\ n_c$), if both nodes are valid, the observable sets of both nodes in *backward-slice* n_c are equal, and the values of all relevant variables are equal in both states:

$$\frac{\begin{array}{c} valid\text{-}node\ n \qquad valid\text{-}node\ n' \\ obs\ n\ (backward\text{-}slice\ n_c) = obs\ n'\ (backward\text{-}slice\ n_c) \\ \forall\, V \in rv\ n_c\ n.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V \end{array}}{((n,s),(n',s')) \in WS\ n_c}$$

Relevant variables $rv\ n_c\ n$ are those variables that are used in some node in *backward-slice* n_c, reachable from n via a CFG path, and not redefined on this path. Simply put, only the values of the relevant variables of a node can influence other nodes in the slice. Thus, at some node n, states that have equal values in the relevant variables of n are observably equivalent for the slice; combined with the equal observable sets two (node,state) configuration tuples are in the weak simulation if they are not distinguishable by the slice.

Using this weak simulation, we can prove the theorem of the correctness of static intraprocedural slicing by showing that the correctness property holds for $WS\ n_c$:

THEOREM 1. *Correctness of Static Intraprocedural Slicing:*

$$\frac{\begin{array}{c} ((n_1,s_1),(n_2,s_2)) \in WS\ n_c \\ n_c,kind \vdash (n_1,s_1) =as\Rightarrow ({n_1}',{s_1}') \end{array}}{\begin{array}{l} \exists\, {n_2}'\ {s_2}'\ as'.\ (({n_1}',{s_1}'),({n_2}',{s_2}')) \in WS\ n_c\ \wedge \\ \qquad n_c,slice\text{-}kind\ n_c \vdash (n_2,s_2) =as'\Rightarrow ({n_2}',{s_2}') \end{array}}$$

Proof. *The proof uses two lemmas: if* (n_1,s_1) *and* (n_2,s_2) *are weakly similar and* (n_1,s_1) *makes (i) arbitrary* τ*-moves in the original graph, the resulting tuple* $({n_1}',{s_1}')$ *is still weakly similar to* (n_2,s_2)*, and (ii) a* $-a\rightarrow$ *move in the original graph to* $({n_1}',{s_1}')$*,* (n_2,s_2) *can make an observable move in the sliced graph, resulting in the tuple* $({n_1}',(transfer\ (slice\text{-}kind\ n_c\ a)\ s_2))$ *which is weakly similar to* $({n_1}',{s_1}')$*.* □

With this correctness property, we can infer another theorem, which is very similar to the correctness property of dynamic slicing as stated in [40]:

THEOREM 2. *Correctness of Slicing with Paths:*

$$\frac{n\ {-as\rightarrow*}\ n' \qquad preds\ (kinds\ as)\ s}{\begin{array}{c} \exists\, as'.\ n\ {-as'\rightarrow*}\ n' \wedge preds\ (slice\text{-}kinds\ n'\ as')\ s\ \wedge \\ (\forall\, V \in Use\ n'.\ state\text{-}val\ (transfers\ (slice\text{-}kinds\ n'\ as')\ s)\ V = \\ state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V)\ \wedge \\ slice\text{-}edges\ n'\ as = slice\text{-}edges\ n'\ as' \end{array}}$$

Take a path as from n to slicing node n' s.t. all predicates on this path are fulfilled using s as initial state. Then there is a path as' in the sliced graph whose predicates are fulfilled using initial state s, and the values of the variables used in n' are the same, no matter if we traverse as in the original or as' in the sliced graph starting in initial state s. *slice-edges* n' as filters from list as all edges whose source node is not in the backward slice. Hence, *slice-edges* n_c as = *slice-edges* n_c as'

$$\begin{array}{l}\textbf{locale } \textit{PDG} = \textit{CFGExit-wf} + \\ \quad \textbf{fixes } \textit{control-dependence} :: {}'node \Rightarrow {}'node \Rightarrow bool \\ \quad\quad \text{— written - } \textit{controls} \text{ -} \\ \quad \textbf{assumes } \textit{Exit-not-cdep}: n \textit{ controls } n' \Longrightarrow n' \neq (\textit{-Exit-}) \\ \quad \textbf{and } \textit{control-dependence-path}: \\ \quad\quad n \textit{ controls } n' \Longrightarrow \exists\, as.\ n -as\rightarrow* n' \wedge as \neq [] \end{array}$$

Figure 5. Locale describing a PDG

states that paths as and as' visit the same nodes in the backward slice of n_c in the same order; thus, as' in the sliced graph matches as in the original graph.

For semantically well-formed CFGs, we can lift this theorem from graphs to the semantics. Instead of a path in the original graph, we assume a semantic evaluation $\langle c,s\rangle \Rightarrow \langle c',s'\rangle$ and identify the nodes n and n' with c and c', respectively. The conclusion is pretty much the same as before:

THEOREM 3. *Correctness of Slicing Semantically:*

$$\frac{n \textit{ identifies } c \qquad \langle c,s\rangle \Rightarrow \langle c',s'\rangle}{\begin{array}{c}\exists\, n'\, as.\ n -as\rightarrow* n' \wedge \textit{preds } (\textit{slice-kinds } n'\, as)\ s \wedge n' \textit{ identifies } c' \\ (\forall\, V \in \textit{Use } n'.\ \textit{state-val } (\textit{transfers } (\textit{slice-kinds } n'\, as')\ s)\ V = \\ \textit{state-val } s'\ V)\end{array}}$$

5.2 Applying Control Dependences

The correctness proof in the previous section was parametrized by a function from nodes to node sets called *backward-slice* and certain constraints. In this section we show how one can use the three different control dependences presented in §3.2 to formalize the respective backward slice and that each of these slices is a valid parameter of the *BackwardSlice* locale, i.e it fulfills the assumptions made in the locale. Showing the correctness property for any further control dependences is analogously done by proving these assumptions, so no insight of the concrete correctness proof formalization is needed.

Program Dependence Graph. For binary control dependences, the backward slice is defined using a program dependence graph. Thus, we use a locale defining a PDG (see Fig. 5) as "middle layer" between backward slice and control dependence definition. We extend the CFG well-formedness locale stipulating (*-Exit-*), fix a binary control dependence relation and assume that (*-Exit-*) is not control dependent on anything and that there exists a nonempty CFG path between control dependent nodes. Then we define the PDG's control and data flow edges via:

$$\begin{array}{c}\textit{If } n \textit{ controls } n' \textit{ then } n \longrightarrow_{cd} n' \\ \textit{If } n \textit{ influences } V \textit{ in } n' \textit{ then } n -V\rightarrow_{dd} n'\end{array}$$

PDG paths are the reflexive transitive closure of PDG edges and denoted $n \longrightarrow_{d}* n'$. The backward slice of node n_c is then defined straightforward via

$$\textit{PDG-BS } n_c \equiv \textit{if valid-node } n_c \textit{ then } \{n' \mid n' \longrightarrow_{d}* n_c\} \textit{ else } \emptyset$$

$$\begin{array}{l}\textbf{locale } \textit{Postdomination} = \textit{CFGExit} + \\ \quad \textbf{assumes } \textit{Entry-path}: \textit{valid-node } n \Longrightarrow \exists\, as.\ (\textit{-Entry-}) -as\rightarrow* n \\ \quad \textbf{and } \textit{Exit-path}: \textit{valid-node } n \Longrightarrow \exists\, as.\ n -as\rightarrow* (\textit{-Exit-}) \\ \textbf{locale } \textit{StrongPostdomination} = \textit{Postdomination} + \\ \quad \textbf{assumes } \textit{successor-set-finite}: \textit{valid-node } n \\ \quad \Longrightarrow \textit{finite } \{n'.\ \exists\, a'.\ \textit{valid-edge } a' \wedge \textit{sourcenode } a' = n \wedge \\ \qquad\qquad \textit{targetnode } a' = n'\}\end{array}$$

Figure 6. Locale with the constraints for postdomination and strong postdomination

Standard Control Dependence Standard control dependence (SCD) bases on the notion of postdomination. As postdomination requires as further constraints that every node is reachable from (*-Entry-*) and can reach (*-Exit-*) via CFG paths, we define a new locale *Postdomination*, extending locale *CFGExit* (as we need an (*-Exit-*) node) with these assumptions, see Fig. 6. Then, postdomination is defined as:

$$\begin{array}{c}n' \textit{ postdominates } n \equiv \textit{valid-node } n \wedge \textit{valid-node } n' \wedge \\ (\forall\, as.\ n -as\rightarrow* (\textit{-Exit-}) \longrightarrow n' \in \textit{set } (\textit{srcs } as))\end{array}$$

We define standard control dependence as in §3.2 and prove that this definition is equivalent to the widely used definition from [9]:

$$\begin{array}{c}\textit{scd } n\ n' \equiv \exists\, a\ a'\ as.\ n' \notin \textit{set}(\textit{srcs } (a{\cdot}as)) \wedge n -a{\cdot}as\rightarrow* n' \wedge \\ \textit{src } a = n \wedge n' \textit{ postdominates } (\textit{trg } a) \wedge \textit{valid-edge } a' \wedge \\ \textit{src } a' = n \wedge \neg\, n' \textit{ postdominates } (\textit{trg } a')\end{array}$$

LEMMA 1. *SCD Definition Variant (Ferrante et al.):*

$$\begin{array}{c}\textit{scd } n\ n' = (\exists\, as.\ n -as\rightarrow* n' \wedge n \neq n' \wedge \neg\, n' \textit{ postdominates } n \wedge \\ n' \notin \textit{set } (\textit{srcs } as) \wedge (\forall\, n''{\in}\textit{set } (\textit{trgs } as).\ n' \textit{ postdominates } n''))\end{array}$$

To verify that the correctness proof holds for this control dependence, we have to show the following theorem:

THEOREM 4. *Correctness Proof for SCD:*
The standard control dependence *scd* is a valid control dependence for locale *PDG*. Using *scd* the resulting slice *PDG-BS* is a valid backward slice for locale *BackwardSlice*.

To prove this theorem, we had to verify that *scd* fulfills all assumptions in locale *PDG* concerning *control-dependence* and then that the PDG backward slice definition *PDG-BS* meets all the constraints imposed on *backward-slice* in locale *BackwardSlice*, in particular that the set of observable nodes in *PDG-BS* n_c is at most a singleton for every valid node.

Weak Control Dependence. Weak control dependence (WCD) is termination sensitive, thus we need a stronger postdomination notion, namely strong postdomination. Strong postdomination also has a further requirement, no edge may have infinitely many target nodes, hence we use a locale *StrongPostdomination* extending locale *Postdomination* as depicted in Fig. 6. As an infinite path (e.g. through a nonterminating loop) exists if there is a finite path longer than k for any fixed k, reaching n' on any path longer than a certain k means that there is no loop between the two nodes. Therefore, we define strong postdomination via:

$$n' \; strongly\text{-}postdominates \; n \equiv n' \; postdominates \; n \wedge (\exists k \geq 1. \forall as \; nx. \; n -as\rightarrow* nx \wedge length \; as \geq k \longrightarrow n' \in set(srcs \; as))$$

Then the definition of WCD and the correctness proof work analogously to standard control dependence:

$$wcd \; n \; n' \equiv \exists a \; a' \; as. \; n' \notin set \; (srcs \; a{\cdot}as) \wedge n -a{\cdot}as\rightarrow* n' \wedge src \; a = n \wedge n' \; strongly\text{-}postdominates \; (trg \; a) \wedge valid\text{-}edge \; a' \wedge src \; a' = n \wedge \neg \; n' \; strongly\text{-}postdominates \; (trg \; a')$$

THEOREM 5. *Correctness Proof for WCD:*
The weak control dependence *wcd* is a valid control dependence for locale *PDG*. Using *wcd* the resulting slice *PDG-BS* is a valid backward slice for locale *BackwardSlice*.

Weak Order Dependence. The definition of weak order dependence (WOD) neither needs a notion of an exit node nor any further assumptions on the CFG. Thus, we can include its definition in the *CFG* locale:

$$\begin{aligned} & wod \; n \; n_1 \; n_2 \equiv n_1 \neq n_2 \wedge \\ & (\exists as_1. \; n -as_1\rightarrow* n_1 \wedge n_2 \notin set \; (srcs \; as_1)) \wedge \\ & (\exists as_2. \; n -as_2\rightarrow* n_2 \wedge n_1 \notin set \; (srcs \; as_2)) \wedge \\ & (\exists a. \; valid\text{-}edge \; a \wedge n = src \; a \wedge \\ & \quad ((\exists as_1'. \; trg \; a -as_1'\rightarrow* n_1 \wedge \\ & \quad (\forall as'. \; (trg \; a -as'\rightarrow* n_2) \longrightarrow n_1 \in set(srcs \; as'))) \vee \\ & \quad ((\exists as_2'. \; trg \; a -as_2'\rightarrow* n_2 \wedge \\ & \quad (\forall as'. \; (trg \; a -as'\rightarrow* n_1) \longrightarrow n_2 \in set(srcs \; as'))))) \end{aligned}$$

As weak order dependence is not a binary relation, we cannot use the *PDG* locale to provide a backward slice. Hence, we have to define its backward slice from the scratch:

$$\frac{valid\text{-}node \; n_c}{n_c \in WOD\text{-}BS \; n_c} \qquad \frac{n' \; influences \; V \; in \; n'' \qquad n'' \in WOD\text{-}BS \; n_c}{n' \in WOD\text{-}BS \; n_c}$$

$$\frac{wod \; n' \; n_1 \; n_2 \qquad n_1 \in WOD\text{-}BS \; n_c \qquad n_2 \in WOD\text{-}BS \; n_c}{n' \in WOD\text{-}BS \; n_c}$$

To apply the correctness proof to weak order dependence, we need to instantiate locale *BackwardSlice*, thus the proof has to verify that *WOD-BS* meets its assumptions:

THEOREM 6. *Correctness Proof for WOD:*
The set *WOD-BS* is a valid backward slice for locale *BackwardSlice*.

The main part of this proof is again to show that the set of observable nodes in *WOD-BS* n_c is at most a singleton.

6. Low-Deterministic Security with Slicing

The correctness results from the previous section are needed in proving that low-deterministic security and slicing comply. Low-deterministic security, a special case of a noninterference definition using partial equivalence relations (per) [30], partitions variables in two security levels, H for secret and L for public data. Basically, a program that is noninterferent w.r.t. low-deterministic security has to fulfill one basic property: executing the program in two different initial states that may differ in the values of their H-variables yields two final states that again only differ in the values of their H-variables; thus the values of the H-variables did not influence those of the L-variables. We will now show how slicing can guarentee that a program is low-deterministic secure[4].

locale *LowDeterministicGraph* = *BackwardSlice* +
fixes H :: $'var \; set$ **fixes** L :: $'var \; set$
fixes *High* :: $'node$ — written (*-High-*)
fixes *Low* :: $'node$ — written (*-Low-*)
assumes *HighLowDistinct*: $H \cap L = \{\}$
and *HighLowUNIV*: $H \cup L = UNIV$
and *Entry-edge-Exit-or-High*: $[\![valid\text{-}edge \; a; src \; a = (\text{-}Entry\text{-})]\!] \Longrightarrow trg \; a = (\text{-}Exit\text{-}) \vee trg \; a = (\text{-}High\text{-})$
and *High-target-Entry-edge*: $\exists a. \; valid\text{-}edge \; a \wedge src \; a = (\text{-}Entry\text{-}) \wedge trg \; a = (\text{-}High\text{-}) \wedge kind \; a = (\lambda s. \; True)_{\surd}$
and *Entry-predecessor-of-High*: $[\![valid\text{-}edge \; a; trg \; a = (\text{-}High\text{-})]\!] \Longrightarrow src \; a = (\text{-}Entry\text{-})$
and *Exit-edge-Entry-or-Low*: $[\![valid\text{-}edge \; a; trg \; a = (\text{-}Exit\text{-})]\!] \Longrightarrow src \; a = (\text{-}Entry\text{-}) \vee src \; a = (\text{-}Low\text{-})$
and *Low-source-Exit-edge*: $\exists a. \; valid\text{-}edge \; a \wedge src \; a = (\text{-}Low\text{-}) \wedge trg \; a = (\text{-}Exit\text{-}) \wedge kind \; a = (\lambda s. \; True)_{\surd}$
and *Exit-successor-of-Low*: $[\![valid\text{-}edge \; a; src \; a = (\text{-}Low\text{-})]\!] \Longrightarrow trg \; a = (\text{-}Exit\text{-})$
and *DefHigh*: $Def \; (\text{-}High\text{-}) = H$
and *UseHigh*: $Use \; (\text{-}High\text{-}) = H$
and *UseLow*: $Use \; (\text{-}Low\text{-}) = L$
and *Low-neq-Exit*: $(\text{-}Low\text{-}) \neq (\text{-}Exit\text{-})$

Figure 7. Locale fixing the assumptions needed for low-deterministic security using slicing

Assumptions. Every per-based approach implies certain assumptions: (i) all H-variables are defined at the beginning of the program, (ii) all L-variables observed (or used in our terms) at the end and (iii) every variable is either H or L. Thus, we have to extend the prerequisites of our framework accordingly. To this end, we define a new locale *LowDeterministicGraph* (see Fig. 7) which extends the locale *BackwardSlice* containing the correctness results.

First, we fix two variable sets H and L. Rules *HighLowDistinct* and *HighLowUNIV* guarantee that these sets partition the set of all variables. Second, we introduce two nodes, (*-High-*) and (*-Low-*). (*-High-*) is the node directly after (*-Entry-*), reached via a no-op edge, but before any other node in the graph. Rules *Entry-edge-Exit-or-High*, *Entry-predecessor-of-High* and *High-target-Entry-edge* make sure this holds. Analogously, rules *Exit-edge-Entry-or-Low*, *Exit-successor-of-Low* and *Low-source-Exit-edge* guarantee that (*-Low-*) is the node directly before (*-Exit-*). Remember that (*-Entry-*) and (*-Exit-*) may neither define nor use variables. Yet, requiring (*-High-*) to define all H- and (*-Low-*) to use all L-variables (via *DefHigh* and *UseLow*), we can still fulfill the per assumptions mentioned before. *UseHigh* and *Low-neq-Exit* are additional conditions necessary below.

[4] This extension of the framework is available online: http://pp.info.uni-karlsruhe.de/~lohner/Slicing/LDS/

Low equivalence. States that are equal in public values, i.e. those in L-variables, are non-distinguishable for an external observer. *Low equivalence* groups those states in an equivalence class using the relation $\approx_L$:

$$s \approx_L s' \equiv \forall\, V{\in}L.\ \textit{state-val}\ s\ V = \textit{state-val}\ s'\ V$$

We can easily prove that if (*-High-*) is not in the backward slice of n_c, the relevant variables of (*-Entry-*) in the sliced graph have equal values for two low-equivalent states:

LEMMA 2.
$$\frac{s \approx_L s' \qquad (\textit{-High-}) \notin \textit{backward-slice}\ n_c}{\forall\, V{\in}\textit{rv}\ n_c\ (\textit{-Entry-}).\ \textit{state-val}\ s\ V = \textit{state-val}\ s'\ V}$$

Another lemma regards the values of the variables used in (*-Low-*) after traversing paths in the sliced graph. Assume we have two paths *as* and *as*′ between n and (*-Low-*). Both paths fulfill all their predicates in the sliced graph of (*-Low-*) with initial states s and s', respectively. These two states agree on the values of all relevant variables of n in this sliced graph. Then the final states after traversing *as* and *as*′ agree in the values of the used variables in (*-Low-*):

LEMMA 3.
$$\frac{\begin{array}{cc} n\ {-as\to*}\ (\textit{-Low-}) & \textit{preds}\ (\textit{slice-kinds}\ (\textit{-Low-})\ as)\ s \\ n\ {-as'\to*}\ (\textit{-Low-}) & \textit{preds}\ (\textit{slice-kinds}\ (\textit{-Low-})\ as')\ s \\ \multicolumn{2}{c}{\forall\, V{\in}\textit{rv}\ (\textit{-Low-})\ n.\ \textit{state-val}\ s\ V = \textit{state-val}\ s'\ V}\end{array}}{\begin{array}{r}\forall\, V \in \textit{Use}\ (\textit{-Low-}).\ \textit{state-val}\ (\textit{transfers}\ (\textit{slice-kinds}\ (\textit{-Low-})\ as)\ s)\ V = \\ \textit{state-val}\ (\textit{transfers}\ (\textit{slice-kinds}\ (\textit{-Low-})\ as')\ s')\ V\end{array}}$$

Low-deterministic security. Assume we have a program and two low equivalent initial states $s \approx_L s'$. Executing the program results in two final states that are not low equivalent. Yet, a different value in a L-variable in the final states can only occur due to a different value in a H-variable in the initial states. Hence, we know that at least one initial H-variable influenced a result L-variable. As (*-Low-*) uses all L-variables and (*-High-*) defines all H-variables, there is a path in the PDG between those nodes due to this interference. Thus, the backward slice of (*-Low-*) contains (*-High-*).

A low-deterministic secure program executed in low equivalent initial states results yields low equivalent final states. The final state of executing a program with an initial state s is *transfers* (*kinds as*) s, if *as* is the path between (*-Entry-*) and (*-Exit-*) in the CFG and *preds* (*kinds as*) s holds. Following the argumentation in the last paragraph, assuring that the backward slice of (*-Low-*) does not contain (*-High-*) should suffice in proving low-deterministic security of a program. Hence, this correctness theorem is stated as follows:

THEOREM 7 (Low-Deterministic Security with Paths).
$$\frac{\begin{array}{cc} s \approx_L s' & (\textit{-High-}) \notin \textit{backward-slice}\ (\textit{-Low-}) \\ (\textit{-Entry-})\ {-as\to*}\ (\textit{-Exit-}) & \textit{preds}\ (\textit{kinds}\ as)\ s \\ (\textit{-Entry-})\ {-as'\to*}\ (\textit{-Exit-}) & \textit{preds}\ (\textit{kinds}\ as')\ s'\end{array}}{\textit{transfers}\ (\textit{kinds}\ as)\ s \approx_L \textit{transfers}\ (\textit{kinds}\ as')\ s'}$$

Proof. *The trick to prove this theorem is to argue in the sliced graph of* (-Low-). *First, we split the* (-Entry-)–(-Exit-) *paths into paths from* (-Entry-) *to* (-Low-) *and the no-op edges between* (-Low-) *and* (-Exit-). *We now apply the correctness results for slicing from the previous section to the trimmed paths. So the values of all variables that are used in* (-Low-) *are equal, regardless if we traversed the original or the sliced graph; this holds for both trimmed paths. Using lemmas 2 and 3 we know that these values also agree in the states after traversing both paths in the sliced graph. Thus, the values of* (-Low-)*'s used variables also agree in the final states after traversing the paths in the original graph. Since traversing the edges between* (-Low-) *and* (-Exit-) *has no influence on the states, we know that the same holds for the final states after executing the whole program. With the fact that the variables used in* (-Low-) *are exactly the L-variables we obtain the conclusion.* □

Low-deterministic security is usually defined via the semantics of a program. We assume that our CFG is semantically well-formed. Let n be the immediate successor node of (*-High-*); since (*-Entry-*) and (*-High-*) are mere auxiliary nodes without a corresponding statement, this is the node that starts the program. *final c* is a property that checks if statement c is fully evaluated. We assume that the node corresponding to a final statement is the immediate predecessor of (*-Low-*). Then, we can lift low-deterministic correctness as follows:

THEOREM 8 (Low-Deterministic Security Semantically).
$$\frac{\begin{array}{ccc} s_1 \approx_L s_2 & (\textit{-High-}) \notin \textit{backward-slice}\ (\textit{-Low-}) & \textit{final}\ c' \\ n\ \textit{identifies}\ c & \langle c,s_1\rangle \Rightarrow \langle c',{s_1}'\rangle & \langle c,s_2\rangle \Rightarrow \langle c',{s_2}'\rangle\end{array}}{{s_1}' \approx_L {s_2}'}$$

Proof. *This theorem is basically a corollary of theorem 7 and the semantically well-formedness definition.* □

CFG Lifting. A CFG constructed for a language would not naturally fulfill the properties assumed in Fig. 7. However, some small adjustments can be sufficient to tackle this problem. Assume we have a CFG that defines with the help of a control dependence definition a PDG. We relabel (*-Entry-*) as (*-High-*), (*-Exit-*) as (*-Low-*) and remove the edge between these nodes. Moreover we add a new entry and exit node, add no-op edges between those two, between the new entry and the old one and between the old exit and the new one. The *Def* and *Use* sets are only redefined for (*-High-*) and (*-Low-*) so that they fulfill the properties needed. This adjusted CFG together with a partitioning of the variables in H and L then fulfills the assumptions of locale *LowDeterministicGraph*.

7. Instantiation with Languages

Exploiting the above results, proving slicing correct for any language just boils down to formalizing a CFG for this language and proving that this formalization fulfills all the needed properties. Thus the correctness proof of static slicing for an instantiated language requires no insight into the slicing definitions or proof details; anyone familiar with formalizing languages can reprove it for a wide variety of languages (imperative and object-oriented).

In the following we show how to instantiate the framework with two different programming languages, a simple While language (without procedures) and Jinja VM byte

$$\frac{(\textit{instrs-of}\ P\ C\ M)_{[pc]} \in \{\textsc{Load}\ idx,\ \textsc{Store}\ idx,\ \textsc{Push}\ val,\ \textsc{Pop},\ \textsc{IAdd},\ \textsc{CmpEq}\} \quad f = (\lambda s.\ \textit{exec-instr}\ (\textit{instrs-of}\ P\ C\ M)_{[pc]}\ P\ s\ (\textit{length}\ cs)\ (\textit{stkLength}\ P\ C\ M\ pc)) \qquad \textit{valid-callstack}\ (P, C0, Main)\ ((C,\ M,\ pc)\cdot cs)}{(P, C0, Main) \vdash (\text{-}\ (C,\ M,\ pc)\cdot cs\ \text{-})\ {-}{\Uparrow}f{\rightarrow}\ (\text{-}\ (C,\ M,\ \textit{Suc}\ pc)\cdot cs\ \text{-})}$$

Figure 8. Example of Jinja CFG edges for simple instructions

code. Whereas static slicing with weak order dependence is correct for both languages (as we do not need additional assumptions), for the correctness of slicing using standard and weak control dependence the respective conditions (as described in §5.2) are shown to be valid.

While. This simple While language features integer and boolean variables, conditionals and while loops. We already showed in [40] how this language can be embedded in the framework to perform dynamic slicing. Note that the locales describing the abstract trace CFG there and the abstract CFG here are the same. This holds as for languages without procedures these CFGs coincide since the question of method inlining does not show up. Hence, we can refer to the details of the implementation in [40] and deduce that all the results shown in this paper hold for While.

By the construction rules of the CFG we can prove that every node is reachable from the entry node and can reach the unique exit node, and that every node has a finite number of successors (due to the fact that without recursive procedures the number of nodes in the graph is finite). Thus we can guarantee the correctness of static slicing for standard and weak control dependence.

Jinja byte code. Jinja [19] models a large subset of the Java language, including operational semantics for the source code and the virtual machine byte code, both with type safety proofs, a compiler from the former to the latter and a byte code verifier (BCV), both verified. Jinja is fully object-oriented and features exception throwing and catching. Slicing such languages is far from trivial. Though the framework is for intraprocedural slicing, it can still be instantiated with a large subset of Jinja, as non-recursive methods can be sliced by inlining method calls; for programs without method calls the intraprocedural slice is well-defined anyway.

Proving slicing correct with the framework requires instantiations of the locales *CFG-wf* and (if SCD or WCD shall be used) *CFGExit-wf*, *Postdomination* and *StrongPostdomination*, which all extend the *CFG* locale. Hence, the first step to prove slicing correct is to formalize an appropriate control flow graph for Jinja byte code.

The Jinja byte code language is, to put it simply, a goto-language using a stack machine with a program counter identifying the current statement in an instruction list. A program consists of a list of class declarations, each with its method declarations where the method bodies are the aforementioned instruction lists. We identify program points with call stacks (lists of triples consisting of class name, method name and program counter), as we use method inlining. Program execution starts at the first instruction of a given method, then proceeds as determined by the current instruction's control flow. The edges are drawn accordingly, taking exception handler delegation and dynamic dispatch into account. Fig. 8 shows a slightly simplified version of an edge formalization, where the function *exec-instr* models the state change and *valid-callstack* ensures some required well-formedness properties of the current program point. As you can see, this is the rule for non-branching instructions. Yet, some instructions can only be modeled with multiple edges (according to §4.3 edges model either a predicate or an update but not both), first predicate ones to determine the target program point, each followed by one edge updating the state accordingly; hence we need additional CFG nodes in between. A typical example for such a situation is method invocation, where first the dispatch target is determined before the appropriate state change is made.

Next, the locale *CFG-wf* has to be instantiated. The problem here is that Jinja byte code uses a stack machine, thus keeping track of the variables is a bit tricky. For example, a program could LOAD a value onto the stack, then do some stack-involving computation where this variable is not used, and thereafter STORE the value again; then the STORE must be data dependent on the corresponding LOAD, which means the same variable must be in the LOAD's *Def* set and in the STORE's *Use* set. Therefore, we say every stack position corresponds to a variable (counted from bottom up); also the local variables are identified through their index positions. Additionally, to distinguish variables of different methods, stack and local variables are labeled with the appropriate call depth available from the CFG node. The heap is treated as a whole and thus instructions are regarded either to define or use the complete heap or to not define and use it at all. This is a conservative approximation, but the properties of *CFG-wf* are not violated. One could gain precision here by using points-to analysis.

The tricky part is to determine the index position of the stack variables that are defined or used in a given node. However, fixing the *Def* and *Use* set is no problem, if the index of the stack's top element is known. Jinja's BCV, which guarantees the stack length to be the same at any program point, no matter how one gets there, provides this index. The state is then defined as a pair of a mapping from the set of variables to appropriate values and a heap.

Using these formalizations we are finally able to instantiate the *CFG-wf* locale and to show that the assumed properties (see Fig. 3) hold. Except for *Entry-empty* (we simply define the *Def* and *Use* set of the entry node to be empty), these properties are shown by case analysis. Having the locale instantiated, we have done all to show slicing correct for Jinja byte code using weak order dependence.

We also proved our formalization of the Jinja byte code CFG to be semantically equivalent to Jinja's *exec* function, which defines the operational semantics of Jinja byte code. Furthermore, we have explicitly proven state conformance as stated by Jinja's BCV to be invariant under the *transfer* function from §4.3 for the CFG.

We can also use standard and weak control dependence, as we instantiated the locales *Postdomination* and *StrongPostdomination*; we omit the details.

The instantiation of the framework with Jinja byte code took about one fourth to one third of the total effort needed to formalize the correctness results. This means, using the framework to adapt the proof to another language can save about 70% of the work compared to starting from scratch.

8. Related Work

8.1 Correctness of Slicing

Reps and Yang [28] were the first to prove static intraprocedural slicing correct for a simple While language without procedures, using CFGs and PDGs. Some generalized frameworks for proving the correctness of slicing already exist. The approach of Gouranton and Le Métayer [11] is also language independent, but based on natural semantics instead of graph structures. It only covers dynamic slicing, the more challenging correctness proof of static slicing is not mentioned. In [38], Ward and Zedan model slicing as a program transformation, i.e. an operation on a program which generates a semantically equivalent program. As the definition of slicing in both works is quite distinct from the graph-based approach used in many program analyses, we think that our work, using the well-known notions of CFGs and PDGs, is more intuitive. Both works rely on pen-and-paper proofs whereas our framework is fully machine-checked.

8.2 Noninterference using Proof Assistants

Formalization of Goguen/Meseguer. In his work on noninterference, Rushby [29] focuses on security policies whose interference relation is intransitive. He formalizes the core of the Goguen/Meseguer to provide an "unwinding lemma", using notation that differs considerably from the original.

Von Oheimb [37] uses Isabelle/HOL to extend this work with nondeterminism. Furthermore, he adds a concept for confidentiality similar to IFC, called *nonleakage*. If a program is nonleaking, data from the initial states should not be leaked, whereas Goguen/Meseguer noninterference says that the occurrence of certain events should not be observable. The combination of both, *noninfluence*, is also formalized.

Verification of Information Flow Type Systems. Several authors proved the correctness of IFC type systems in a proof assistant; mostly, noninterference is defined as low-deterministic security.

Kammüller developed a framework for using the byte code verifier of a Java-like language to show non-interference [18]. This approach is related to ours, as he uses the module concept of Coq to abstract from a specific language syntax. His framework is restricted to byte code languages, whereas ours can handle source as well as byte code languages. Due to Coq, his proofs are executable as programs, i.e. they can actually run their non-interference check. The underlying information flow type system is not given explicitly, but seems inspired by the work of Barthe et al. [5].

The IFC type system of Banerjee and Naumann [4] covers the sequential core of Java. They prove their system to be sound using simulation and indistinguishability of states. This work (omitting access control) is formalized in two different proof assistants: in PVS by Naumann [24] and in Isabelle/HOL by Strecker [36].

Barthe and Nieto [6] formalize an information flow type system for a concurrent while language as defined from Boudol and Castellani [8], which is an extension of the Volpano/Smith system [31] . Using Isabelle/HOL, they define a bisimulation (which allows stuttering) over the semantic rules to show noninterference. Furthermore, they also verify noninterference for scheduling programs. The sequential subset of the Volpano/Smith system was also formalized in Isabelle/HOL by Snelting and Wasserrab [34], together with a proof that it preserves low-deterministic security.

9. Conclusion and Future Work

We presented a machine-checked correctness proof for static intraprocedural slicing, and a machine-checked correctness proof for a slicing-based IFC algorithm. The modular proof structure allows to plug in other language or dependence definitions easily.

However, the underlying framework does not yet handle methods and interprocedural slicing. In order to extend our proof to the context-sensitive, object-sensitive interprocedural IFC with declassification as described in [15], more work will be needed. Threads and concurrency will pose an even greater challenge. While we have devised, implemented and evaluated sophisticated slicing algorithms for concurrent programs [20, 10], and extended Jinja with threads [23], we have not yet extended our IFC to Java programs with threads. A machine-checked correctness proof for this will probably require several years of work.

Let us finally point out a limitation of standard noninterference, which has to be overcome. As mentioned above, low-deterministic security, as well as related noninterference definitions, treat a program as a black box and cannot express security-related properties for interior statements or intermediate states. Such properties could be annotations (as required in some type-based IFC systems), or interior security levels or dependences (as in [15]). Future definitions of noninterference must maintain the overall security properties, but must allow to argue about interior details of programs; otherwise the correctness proofs will not be able to handle the high degree of precision in modern program analysis and IFC.

References

[1] Torben Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Information Processig Letters*, 106(2):45–51, 2008.

[2] Paul Anderson, Thomas Reps, and Tim Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE TSE*, 29(8):721–733, 2003.

[3] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES 2003*, volume 3085 of *LNCS*, pages 34–50. Springer, 2004.

[4] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. of CSFW 2002*, pages 239 – 253. IEEE, 2002.

[5] Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In *VMCAI 2004*, volume 2937 of *LNCS*, pages 2–15. Springer, 2004.

[6] Gilles Barthe and Leonor Prensa Nieto. Secure information flow for a concurrent language with scheduling. *Journal of Computer Security*, 15(6):647–689, 2007.

[7] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *Proc. of POPL 1993*, pages 384–396. ACM, 1993.

[8] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[10] Dennis Giffhorn and Christian Hammer. Precise slicing of concurrent programs - an evaluation of static slicing algorithms for concurrent programs. *Journal of Automated Software Engineering*, 2009.

[11] Valerie Gouranton and Daniel Le Métayer. Dynamic slicing: a generic analysis based on a natural semantics format. *Journal of Logic and Computation*, 9(6):835–871, 1999.

[12] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *Proc. of ISSSE 2006*, pages 87–96. IEEE, March 2006.

[13] Christian Hammer. *Information Flow Control for Java*. PhD thesis, Universität Karlsruhe (TH), 2009. Submitted.

[14] Christian Hammer, Rüdiger Schaade, and Gregor Snelting. Static path conditions for java. In *Proc. of PLAS 2008*, pages 55–66. ACM, 2008.

[15] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Technical Report 2008-16, Universität Karlsruhe (TH), November 2008. http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/574047.

[16] Christian Hammer and Gregor Snelting. An improved slicer for Java. In *Proc. of PASTE 2004*, pages 17–22. ACM, June 2004.

[17] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[18] Florian Kammüller. Formalizing non-interference for a simple bytecode language in Coq. *Formal Aspects of Computing*, 20(3):259–275, 2008.

[19] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM TOPLAS*, 28(4):619–695, 2006.

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

[21] Jens Krinke. Program slicing. *HandBook of Software Engineering and Knowledge Engineering, Vol. 3: Recent Advances*, pages 307–332, 2004.

[22] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20(6):563–595, 2008.

[23] Andreas Lochbihler. Type safe nondeterminism - a formal semantics of Java threads. In *Proc. of FOOL 2008*, 2008.

[24] David A. Naumann. Machine-checked correctness of a secure information flow analyzer (preliminary report). Technical Report SIT Report CS-2004-10, Stevens Institute of Technology, March 2004.

[25] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[26] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE TSE*, 16(9):965–979, 1990.

[27] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM TOPLAS*, 29(5):27, 2007.

[28] Thomas W. Reps and Wuu Yang. The semantics of program slicing and program integration. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT 1989, Vol.2*, volume 352 of *LNCS*, pages 360–374. Springer, 1989.

[29] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, dec 1992.

[30] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher Order Symbolic Computation*, 14(1):59–91, 2001.

[31] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of POPL 1998*, pages 355–364. ACM, 1998.

[32] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *SAS 1996*, volume 1145 of *LNCS*, pages 332–348. Springer, 1996.

[33] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM TOSEM*, 15(4):410–457, 2006.

[34] Gregor Snelting and Daniel Wasserrab. A correctness proof for the Volpano/Smith security typing system. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. September 2008. Formal proof development, http://afp.sf.net/entries/VolpanoSmith.shtml.

[35] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proc. of PLDI 2007*, pages 112–122. ACM, 2007.

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

[37] David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *ESORICS 2004*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004.

[38] Martin Ward and Hussein Zedan. Slicing as a program transformation. *ACM TOPLAS*, 29(2):7, 2007.

[39] Daniel Wasserrab. Towards certified slicing. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. September 2008. Formal proof development, http://afp.sf.net/entries/Slicing.shtml.

[40] Daniel Wasserrab and Andreas Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In Outmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLS 2008*, volume 5170 of *LNCS*, pages 294–309. Springer, 2008.

[41] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.