Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing

Daniel Wasserrab, Denis Lohner*
Karlsruhe Institute of Technology (KIT), Germany
daniel.wasserrab@kit.edu, denis.lohner@kit.edu

Abstract

We present a machine-checked correctness proof for information flow noninterference based on interprocedural slicing. It reuses a correctness proof of the context-sensitive interprocedural slicing algorithm of Horwitz, Reps, and Binkley. The underlying slicing framework is modular in the programming language used; by instantiating this framework the correctness proofs hold for the respective language, without reproving anything in the correctness proofs for slicing and noninterference. We present instantiations with two different languages to show the applicability of the framework, and thus a verified noninterference algorithm for these languages. The formalization and proofs are conducted in the proof assistant Isabelle/HOL.

1 Introduction

Today, mobile code is ubiquitous. But even if we trust the identity of the code sender, do we trust the actual code? The idea of *information flow control* (IFC) is to guarantee that a program does not leak secret information to untrusted entities. Various algorithms for IFC exist, which claim to ensure this security, where the latter is usually expressed as some form of noninterference. But again, can we trust such algorithms? Hence, their correctness is a topic of burning pressure.

However, of the many IFC algorithms published so far, most come with a manual proof only, some even without any proof. There has been some effort to employ theorem provers for this task, e.g. see [10, 7, 3], yet almost exclusively in the area of IFC type systems. Such type systems are the standard technique for IFC nowadays, as they are easy to handle, modular and efficient.

Most type systems are not flow-sensitive, context-sensitive, let alone object sensitive, which can lead to a loss of precision and false alarms. The slicing-based approach to IFC developed in our group, see [12] for an overview, can reduce the amount of false alarms significantly. Up to now, we were only able to provide a formal correctness property that assumes the correctness of slicing (see Appendix 1 in [22]) and a machine-checked correctness proof was out of reach for the last years. To eliminate this deficit, we initiated the project "Quis custodiet", which aims for the verification of such slicing-based IFC algorithms in a theorem prover.

In this contribution, we show how a correctness proof for the interprocedural context-sensitive slicing algorithm of Horwitz, Reps, and Binkley [13] – the basic program analysis used in the slicing-based approach – can be leveraged to provide a first result on relating interprocedural slicing and IFC noninterference. The formalization is realized in Isabelle/HOL [19] and all the proofs are machine-checked. As the framework on which the slicing correctness proof bases uses a graph structure which abstracts from a concrete programming language, the noninterference proof immediately inherits this modularity. Hence, to show noninterference for any language, we just need to instantiate the slicing framework, without reproving anything in the correctness proofs. We show how to lift any graph in the framework such that it fulfils the requirements for the noninterference proof and provide two language instantiations of the framework, thus showing its applicability.

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

```
E int main(){
1   if(l1 == 42){
2    swap(h1,h2);
    } else {
3    swap(l1,l2);
    }
4   print l1;
5 }

e void swap(int x, int y){
i   int temp = y;
ii  y = x;
iii x = temp;
x }
```

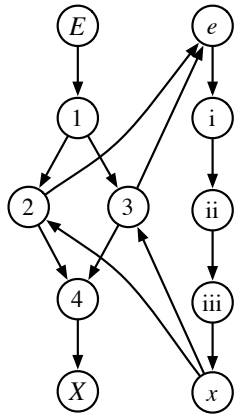


Figure 1: Running example: source code and control flow graph

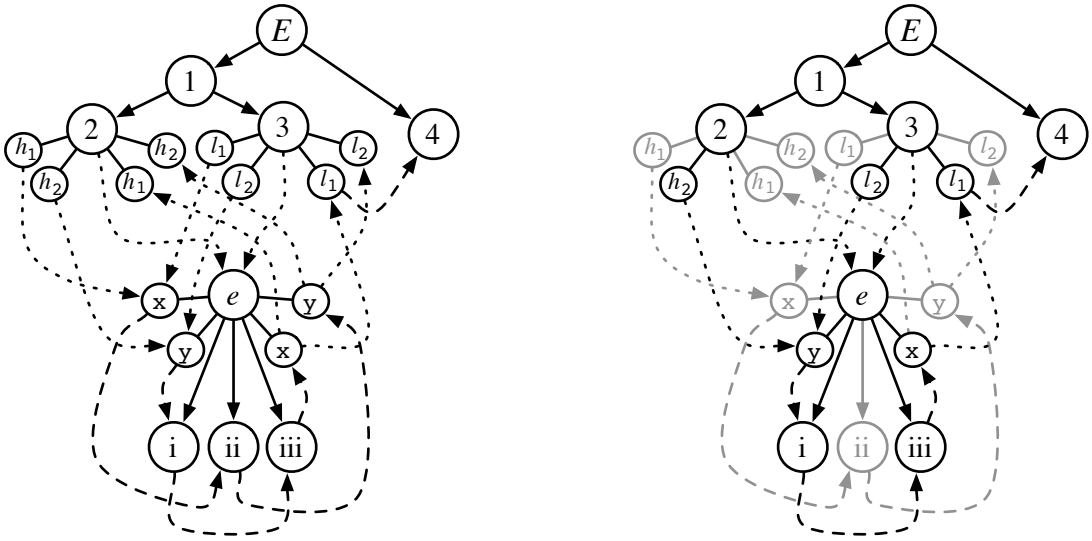
The remainder of the paper is structured as follows: Sec. 2 gives an overview of slicing, Sec. 3 sketches the slicing framework, the formalization of the Horwitz-Reps-Binkley slicer and its correctness proof. In Sec. 4, we show how the results from the section before can be used to guarantee information flow noninterference with slicing. Sec. 5 presents two instantiations of the framework, whereas Sec. 6 compares the sizes of the framework with that of the noninterference proof and the instantiations, as well as with previous work. Sec. 7 discusses related work, and Sec. 8 concludes and gives future prospects.

2 Program Slicing

Given a certain point in a program, a *backward slice* (or short: slice) is a conservative approximation of all statements that can influence this point. Slicing proved to be useful for many applications, e.g. debugging [23], testing [5], and software security algorithms [12, 22]. Today, commercial slicing tools such as Codesurfer [1] are routinely used for some of these tasks. Hundreds of papers on slicing have been published in the last two decades, for an overview, see Tip [25] and Krinke [16].

We will clarify slicing using the example from Fig. 1, whose source code is shown on the left. The right hand side depicts the corresponding control flow graph (CFG), each node bears a label, which gives the line of the statement to which it corresponds. The program is quite easy: after checking a predicate, it just swaps two values with the help of a procedure call. Note that we assume call-by-reference, so that the procedure indeed performs the swap.

For slicing, we need the *system dependence graph* (SDG) for this program which models the dependences between statements. It reuses the nodes of the CFG, but introduces formal and actual parameter nodes at call sites and procedure entries. Its nodes are connected via call, parameter and dependence edges; the call edges are the same as in the CFG, the parameter-in and -out edges are responsible for argument passing. Data dependence edges are drawn from a node where a variable is defined (i.e. assigned) to a node where this variable is used (e.g. in a calculation or predicate), such that no other node on some CFG path between them redefines the variable. We draw a control dependence edge between two nodes if the computation at the source node of this edge influences if the control flow reaches the target node (consider if-branches and while-loops). Fig. 2(a) shows the SDG of the running example: The actual- and formal-in parameters are drawn on the left of call and procedure entry nodes, the out parameters are on their right. As we have call-by-reference, `x` and `y` are the formal-in and -out parameters



(a) SDG (b) Slice

Figure 2: System dependence graph and naive slice of node 3

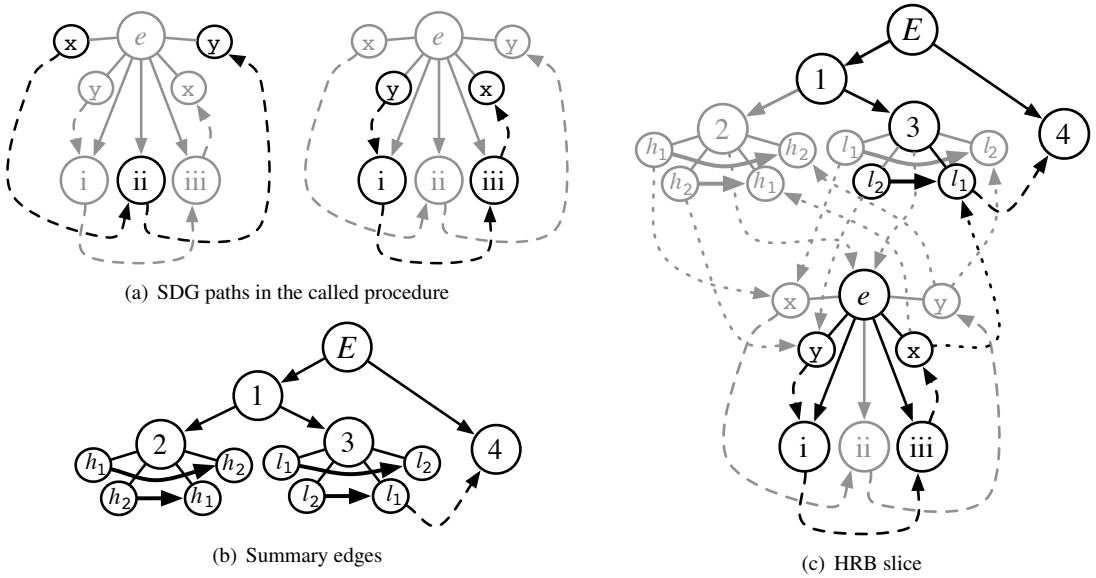
of `swap`. Control and data dependences are solid and dashed, call, return and parameter edges dotted arrows.

In the intraprocedural case, slicing is basically a reachability analysis, i.e. a slice collects all nodes that can reach the slicing node via an SDG path. In Fig. 2(b) the nodes that are greyed out are not part of the slice for node 4 (the **`print`** l_1 statement) if it were computed this way. Note that node 2 is in the slice, although a quick check of the source code in Fig. 1 reveals that no information can flow from this call to node 4. This is due to the fact that this algorithm is not context-sensitive, i.e. it does not distinguish between different call sites.

In [13], Horwitz, Reps, and Binkley present an algorithm for context-sensitive slicing, which eliminates such spurious nodes. It works in two phases: In the first phase, it collects all nodes that reach the slicing node via SDG edges except parameter-out edges; in the second phase, it collects all nodes which reach the nodes from the first phase via SDG edges, this time except call and parameter-in edges. While this prevents returns to "wrong" call sites, it also eliminates correct nodes before a call. To this end, they introduce *summary edges*: if there is a SDG path in a called procedure from a formal-in to a formal-out parameter node, we introduce such an edge between the respective actual parameter nodes at every call site; hence, we can access nodes prior to a call via this "shortcut". Fig. 3(a) shows two SDG paths in the called procedure of our example, which give rise to two different summary edges at each call site of this procedure, see Fig. 3(b). The resulting slice of the Horwitz-Reps-Binkley algorithm (HRB) if applied to node 4 can be seen in Fig. 3(c); note that the spurious node 2 is no longer in the slice, whereas the second actual-in parameter node of node 3 still is, due to the summary edge.

3 Formalizing Context-Sensitive Slicing and its Correctness Proof

To achieve a modular framework for interprocedural slicing, we base our work on an abstract representation of a CFG. This representation assumes certain structural and well-formedness properties, but leaves the concrete definition of some functions unspecified. Instead, the instantiating language has to provide the latter; hence, the unspecified functions can be regarded as parameters of the instantiation. This is



(a) SDG paths in the called procedure

(b) Summary edges

(c) HRB slice

Figure 3: Summary edges and HRB slicing

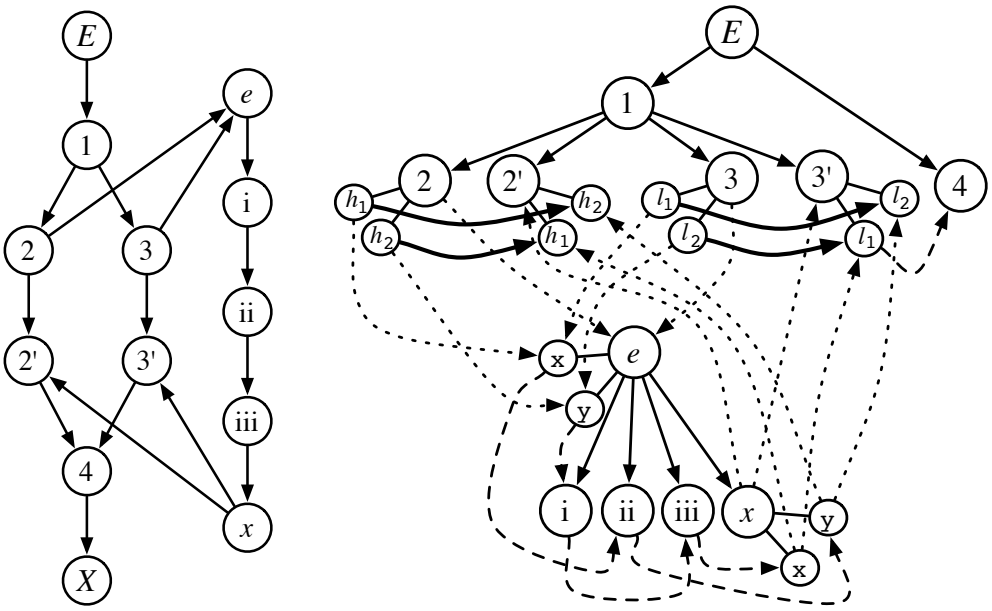
the key concept to make the proof modular and reusable. In this section, assuming a function means it is unspecified whereas defined functions are specified within the proof context of the assumptions.

3.1 The Abstract Control Flow Graph

The abstract CFG consists of nodes and edges of arbitrary type. A concrete edge or node is in the graph, if it fulfils the predicate *valid-edge* or *valid-node*, resp. As only the instantiation can say which edges are part of the graph, *valid-edge* is a parameter of the instantiation and therefore its definition unspecified. *valid-node* is defined using *valid-edge*. *Entry* and *Exit* are the global entry and exit nodes in the special procedure *Main*, in which execution starts. The instantiation also has to provide a list of the procedures in the program, each with its respective formal-in and -out parameters. Note that we do not restrict ourselves to one out parameter per procedure; this enables us e.g. to model call-by-reference. Each node is part of exactly one procedure.

To give the CFG a semantic meaning, so that we can argue about program executions, edges also carry semantic information. A state is basically a stack of call frames consisting of local variables and a return information. Traversing an intraprocedural edge either updates the local variables or checks if a certain predicate holds. Call and return edges also push and pop call frames, taking care of parameter passing and that leaving a procedure returns to the call site which was used for entering that procedure. We adopt a standard trick and split a call site in a call and a return node; this helps in distinguishing if a call site is visited before or after a procedure call.

Fig. 4(a) shows the adapted CFG of the running example so that its structure agrees to the framework. Note that the call site nodes are now split in two different nodes, e.g. 2 is the call, 2' the return node of the former call site node 2 of Fig. 1. We do not label the edges with their semantic effect, instead, we give some examples: the semantic effect of edge 1$\longrightarrow$2 checks that in the topmost call frame, the value of l_1 is 42. Traversing edge ii$\longrightarrow$iii updates the local variables in the topmost call frame by assigning `y` the value of `x`. Call edge 3$\longrightarrow e$ pushes a new call frame on the state, such that node 3' is saved as return information and that in the local variables, the formal parameters `x` and `y` of procedure `swap` get assigned the values of l_1 and l_2 at the call site; all other local variables stay undefined.



(a) CFG (b) SDG

Figure 4: Framework graphs of the running example

We may combine CFG edges to paths; however, not all paths agree to a program execution. We call those paths that do *valid paths* and write $n\ -as\rightarrow_{\surd}*\ n'$ for them, where n and n' are the source and target node of the path and as is the list of edges which constitute the path. For example in Fig. 4(a), path [1,3,e,i,ii,iii,x,3',4] is a valid path, whereas [2,e,i,ii,iii,x,3'] is not.

We also assume a *Def* and a *Use* set for every node. Their intuition is as usual: they contain all variables (or more specific: locations) that get assigned and are used, resp., in the statement that this node represents. Consider again Fig. 4(a) and the source code in Fig. 1: in node iii, `temp` is used and `x` defined, node 1 just uses l_1, call node 2 uses h_1 and h_2, whereas return node 3' defines l_1 and l_2.

To guarantee that the abstract graph defined is indeed a CFG, we assume some properties that have to hold: there are no multi-edges, the graph is deterministic – i.e. in any state and node, only one leaving edge can be traversed –, the *Def* and *Use* sets of a node agree to the semantic effects of the edges leaving that node, those of *Entry* and *Exit* are empty, etc. Due to space restriction, we cannot give a full list of all these conditions; they can be found in the proof scripts [28].

3.2 The System Dependence Graph

The SDG reuses the nodes of the CFG but also introduces new nodes for actual and formal in and out parameters. Fig. 4(b) shows the framework SDG of the running example. Note that now the formal-out parameter nodes belong to the procedure exit node x and the actual-out parameter nodes to the respective return nodes. We write fip, fop, aip and aop for formal-in, formal-out, actual-in, and actual-out parameter nodes; a subsequent tuple specifies the node to which the parameter node belongs and the name of the concrete parameter; e.g. aip(2,h_2) is the node of the second actual-in parameter of 2, which is h_2.

The edges in the SDG can be categorized into intraprocedural dependence edges and interprocedural call, return and parameter edges. Data dependence is defined as usual, a node n influences the value of

a location V in node n', if V is defined in n and used in n' and their respective CFG nodes are connected with an intraprocedural CFG path, on which no node redefines V. We define control dependence in the standard understanding as defined by Ferrante et al. [11], however in an equivalent version of Wolfe [30] (the proof of equivalence is part of the formalization): a node n controls the execution of node n', if n has two intraprocedural successors, the first one postdominated by n', the other one not. A node n' postdominates node n, if all intraprocedural paths from n to the method (or global) exit contain n'.

Instead of giving the rules, we describe the SDG edges informally and provide an example in the SDG of Fig. 4(b) for each edge type; their formalization is straightforward and can be found elsewhere [28]. We have the following edges:

- data dependence $n \longrightarrow_{dd} n'$, if n' is data dependent on n (e.g., aop(3,l_1) $\longrightarrow_{dd}$ 4),
- control dependence $n \longrightarrow_{cd} n'$, if (i) n' is control dependent on n (1 $\longrightarrow_{cd}$ 3), (ii) n' is a parameter node attached to n ($e \longrightarrow_{cd}$ fip(e,x)), or (iii) n is the method entry and n' a method exit of the same procedure ($e \longrightarrow_{cd} x$);
- call $n -p\rightarrow_{call} n'$, if n is a call node and n' the corresponding method entry of procedure p (2 $-$swap$\rightarrow_{call} e$);
- return $n -p\rightarrow_{ret} n'$, if n is a method exit of procedure p and n' a matching return node in the caller (x $-$swap$\rightarrow_{ret}$ 3');
- parameter-in $n -p{:}V\rightarrow_{in} n'$, if V is the ith formal-in parameter, n the ith actual-in and n' the ith formal-in parameter node of the respective call of p (aip(2,h_2) $-$swap:x$\rightarrow_{in}$ fip(e,x));
- parameter-out $n -p{:}V\rightarrow_{out} n'$, if V is the ith formal-out parameter, n the ith formal-out and n' the ith actual-out parameter node of the respective return from p (fop(x,y) $-$swap:y$\rightarrow_{out}$ aop(2',h_2)).

To determine the summary edges, which are indispensable for efficient context-sensitive interprocedural slicing, we need to formalize *realizable* paths in the SDG, i.e. paths on which a finished procedure call always returns to the site of the most recently executed unmatched call. Analogously to [21], where this is done using a grammar, we define a predicate *matched* describing *same-level realizable* SDG paths. *matched n ns n'* states that there is a context-sensitive SDG path from node n to n' (both in the same procedure), visiting nodes *ns* on its way. We add a summary edge $n \longrightarrow_{sum} n'$ between an actual-in and -out parameter at a call site, if the respective formal-in and -out parameters in the called procedure are connected via a *matched* path.

Consider again Fig. 4(b): since e and x are connected, we get the summary edges 2 $\longrightarrow_{sum}$ 2' and 3 $\longrightarrow_{sum}$ 3' between call and matching return node. Moreover, as there are also paths between fip(e,x) and fop(x,y) as well as between fip(e,y) and fop(x,x) (via [fip(e,x),ii,fop(x,y)] and [fip(e,y),i,iii,fop(x, x)], resp., review also Fig. 3(a)), we also get summary edges between the actual parameter nodes (the bold arrows in Fig. 4(b)): aip(2,h_1) $\longrightarrow_{sum}$ aop(2',h_2), aip(2,h_2) $\longrightarrow_{sum}$ aop(2',h_1), aip(3,l_1) $\longrightarrow_{sum}$ aop(3',l_2), and aip(3,l_2) $\longrightarrow_{sum}$ aop(3',l_1).

3.3 The Algorithm of Horwitz, Reps, and Binkley Formally

The slicing algorithm by Horwitz, Reps, and Binkley works in two phases: First, beginning at the slicing node n, we traverse backwards all intraprocedural dependence, call, parameter-in and summary edges, but no return or parameter-out edges, and insert all visited nodes into the slice. Second, beginning at all actual-out and return nodes visited in Phase 1, we traverse all intraprocedural dependence, return, parameter-out and summary edges, but no call or parameter-in edges, and again include all visited nodes in the slice. This approach guarantees a context-sensitive interprocedural slice; for more details see [13, 21]. We call this slice HRB slice and write *HRB-slice*.

We formalize this using the two sets $\mathit{sum\text{-}SDG\text{-}slice1}\ n = \{n'.\ n' \longrightarrow_{\{dd \cup cd \cup call \cup in \cup sum\}}* n\}$ and $\mathit{sum\text{-}SDG\text{-}slice2}\ n = \{n'.\ n' \longrightarrow_{\{dd \cup cd \cup ret \cup out \cup sum\}}* n\}$, collecting backwards the nodes of the transitive hull of the edges traversed in Phase 1 and 2, resp., beginning at node n. To finally get *HRB-slice n*, the HRB slice of slicing node n, we insert (i) all elements from $\mathit{sum\text{-}SDG\text{-}slice1}\ n$ in this set, and (ii) all elements from $\mathit{sum\text{-}SDG\text{-}slice2}\ n'$, if $n' \in \mathit{sum\text{-}SDG\text{-}slice1}\ n$ and n' is either a return or actual-out node.

3.4 The Correctness Proof

This section sketches the correctness proof for the slicing algorithm of Horwitz, Reps, and Binkley; a detailed description can be found in [27].Correctness is defined as a weak simulation between the original and sliced CFG, if they are regarded as labelled transition systems. In the sliced CFG, the semantic effect of all edges whose source node is not in the slice is replaced by a no-op; i.e. we do not eliminate the nodes and edges, just their effects on a traversal.

However, more understandable and useful is a corollary of this correctness theorem. Informally, it agrees to the wide-spread informal correctness requirement for slicing, which can be sketched like this: *"After performing the slicing algorithm, the observable effects at the slicing node are preserved."* These observable effects include which path was taken to reach the slicing node and which values the locations contain that are used in this node. More formally, we say that a program execution to the slicing node gives rise to a traversal of its sliced CFG (which we also call execution), such that the execution paths agree if we only consider the nodes in the slice, and such that the variables that are used in the slicing node contain the same values after both executions.

In our formalization, *kinds as* identifies the semantic effects of edges *as* in the original, *slice-kinds n as* those in the sliced graph of node n. *transfers* computes the effect traversing these edges has on a state, *preds* guarantees that all predicates w.r.t. an initial state hold. We identify a program execution via a valid path $n\ -as\rightarrow_{\surd}*\ n'$ on which all predicates in the initial state hold. *slice-edges n as* keeps all edges of *as* whose source nodes are in the HRB slice of node n. Then, we can prove the following theorem, which formalizes above correctness intuition:

Theorem 1. *Fundamental Property of Static Interprocedural Slicing:*

$$\frac{n\ -as\rightarrow_{\surd}*\ n' \qquad \mathit{preds}\ (\mathit{kinds}\ as)\ [cf]}{\begin{array}{c}\exists as'.\ n\ -as'\rightarrow_{\surd}*\ n' \wedge \mathit{preds}\ (\mathit{slice\text{-}kinds}\ n'\ as')\ [cf] \wedge \mathit{slice\text{-}edges}\ n'\ as = \mathit{slice\text{-}edges}\ n'\ as' \wedge \\ (\forall V \in \mathit{Use}\ n'.\ \mathit{state\text{-}val}\ (\mathit{transfers}\ (\mathit{slice\text{-}kinds}\ n'\ as')\ [cf])\ V = \mathit{state\text{-}val}\ (\mathit{transfers}\ (\mathit{kinds}\ as)\ [cf])\ V)\end{array}}$$

4 Slicing Guarantees Information Flow Noninterference

In this section, we employ the correctness result from the previous section to prove that context-sensitive interprocedural slicing guarantees classical noninterference in the area of information flow control (IFC). The notation is slightly simplified w.r.t. the proof scripts, so that the reader does not get confused with concepts orthogonal to the desired results, e.g. conversion between SDG and CFG nodes, initial states often consist of only one call frame, etc. Previous work [29] verified this connection between intraprocedural slicing and noninterference.

4.1 Noninterference

IFC noninterference is used as a security criterion stating that, given a program (or any other information processing system), secret information cannot leak to public outputs. Classical language based IFC

noninterference, as used e.g. by Volpano et. al [26], therfore assigns a security level (from a given security lattice) to every variable in the program. This security label is fixed for a variable and can not be altered during a program run.

For simplicity we only consider security levels H and L for highly classified or secret information and public information respectively. In our framework we assume the variable sets H and L that partition the set of all variables.

Further, two program states s and s' are said to be *low equal* ($s \approx_L s'$), if they only differ in their H-parts. Formally, using the notion of our framework:

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

Then, a program c is noninterferent (in the classical understanding), iff for every pair of (terminating) program runs starting in low equal states, the resulting final states are again low equal:

$$\forall\, s_1\ s_2\ {s_1}'\ {s_2}'.\ (s_1 \approx_L s_2 \wedge \langle c, s_1\rangle \Downarrow {s_1}' \wedge \langle c, s_2\rangle \Downarrow {s_2}') \longrightarrow {s_1}' \approx_L {s_2}'$$

This definition makes certain (implicit) assumptions: (i) all H-variables are defined at the beginning of the program, (ii) all L-variables are observed at the end and (iii) every variable has a security label attached (i.e. every variable is either H or L).

4.2 Noninterference via Slicing

In our framework a naive approach to guarantee that these assumptions hold would require *Entry* to have all H-variables in its *Def* set and *Exit* to have all L-variables in its *Use* set. Yet, remember that our framework claimed that the *Def* and *Use* sets of both nodes are empty. Hence, we assume two special nodes, *High* and *Low*, where the former is the only immediate successor of *Entry*, the latter the only immediate predecessor of *Exit*; both connecting edges are labelled with intraprocedural no-ops, hence both *Low* and *High* are in procedure *Main*. All H-variables are contained in the *Def* set of *High* (for conformance reasons, they also have to be in its *Use* set), all L-variables in the *Use* set of *Low*.

But how does this help to guarantee low equality? Assume we have a program and two low equal initial states and executing the program results in two final states that are not low equal. As we assume all H-variables to be defined at the beginning of the program[1] and the framework CFG has to be deterministic (cf. Sec. 3), 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 valid path in the SDG between them due to this interference (this correlation is an implication from the correctness result in the previous section). Thus, the HRB slice computed for *Low* contains *High*. So, to show that there is no such influence, there may not be a valid SDG path, hence *High* $\notin$ *HRB-slice Low* has to hold.

Example. Consider our running example: let the variables h_1, h_2 as well as `temp`, `x` and `y` be labelled *H*. l_1 and l_2 contain low information. Slicing for *Low* in the augmented graph yields *sum-SDG-slice*1 *Low* = $\{E$, 1, 3, aip(3,l_1), aip(3,l_2), 3', aop(3',l_1), aop(3',l_2), *Low*$\}$. As 3', aop(3',l_1) and aop(3',l_2) are return or actual-out nodes, we have to include the respective *sum-SDG-slice*2 sets to the slice. However, these sets won't include node 2 or 2' nor one of their attached parameter nodes, because Phase 2 excludes the traversal of call and parameter-in edges. As node *High* only has outgoing dependence edges to the parameter nodes of node 2[2], it is not included in the slice and our program is considered secure.

Note, that type systems like [26] reject this program as unsecure because they can not distinguish between the different calling contexts. Hence, they register a (spurious) flow from h_1 to l_2 (via `temp`).

[1] Therefore, intermediate input to the program must already be fixed in the starting state.

[2] There are no data dependencies from *High* to e, i, ii, iii or x, as fip(e,x) and fip(e,y) redefine `x` and `y`, and `temp` is (re)defined in i. Also, we do not allow data dependences across procedure boundaries.

4.3 The Proof

First we state a lemma, that connects low equal states with the concept of *relevant variables*, which play an important role in the definition of the weak simulation for the correctness proof of slicing. The set of relevant variables $rv\ n_c\ n$ collects for a node n all those variables, which are defined at some node n', such that $n' \in$ *HRB-slice* n_c and there is an intraprocedural path between n and n' without redefinition. To put it simply, only the values of the relevant variables of a node can influence subsequent nodes in the slice. If no other node on any path between n and the slicing node n_c is in the slice, then $rv\ n_c\ n = Use\ n_c$.

If we have two low equal states and *High* is not in the HRB slice of n_c, then the values of the relevant variables of *Entry* are equal in both states:

Lemma 2.

$$\frac{s \approx_L s' \qquad High \notin HRB\text{-}slice\ n_c}{\forall V \in rv\ n_c\ Entry.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V}$$

The next lemma is the key lemma in our proof; its proof takes up the majority of the whole theory. It regards the values of the variables used in *Low*, which is also the slicing node, after traversing paths in the sliced graph. Assume we have two paths *as* and *as'* between *n*, a node in the *Main* procedure, and *Low*. Both paths fulfil 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.

$$\frac{\begin{array}{c} n\ -as\rightarrow_{\surd}*\ Low \qquad n\ -as'\rightarrow_{\surd}*\ Low \\ get\text{-}proc\ n = Main \qquad \forall V \in rv\ Low\ n.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V \\ preds\ (slice\text{-}kinds\ Low\ as)\ s \qquad preds\ (slice\text{-}kinds\ Low\ as')\ s' \end{array}}{\begin{array}{l} \forall V \in Use\ Low.\ state\text{-}val\ (transfers\ (slice\text{-}kinds\ Low\ as)\ s)\ V = \\ \qquad state\text{-}val\ (transfers\ (slice\text{-}kinds\ Low\ as')\ s')\ V \end{array}}$$

Proof. As *n* as well as *Low* are in *Main*, both *as* and *as'* are *same level path*s. Such paths are defined via a predicate *same-level-path as*, a special case of *same-level-path-aux cs as*, which describes the situation that we already traversed some edges of the same level path: *as* is the remaining path and *cs* is a stack of all call edges previously visited, for which no matching return edge has been encountered. It is inductively defined on the first edge of *as*: if it is a call edge, we push it on top of *cs* and continue with this new stack and *tl as*. If it is a return edge which belongs to the call edge *hd cs*, we continue with *tl cs* and *tl as*; if it is not or *cs* is empty, the predicate is false. If *hd as* is an intraprocedural edge, we just eliminate this edge. The above lemma is a corollary of a more general one, which draws the same conclusion but only requires suffixes of same level paths with an appropriate call edge stack. The latter can be proved by induction on *same-level-path-aux*, where in each nonempty case we have to make sure that the following conditions on the path suffixes from the assumptions hold: (i) their lengths are equal, (ii) the leading edge of both paths is the same, and (iii) for all relevant variables in the target node of the leading edge, the values after traversing the leading edge in state s and s' agree. Combining this with the induction hypotheses, the conclusion follows directly. □

In our notion of paths, the final state of executing a program in an initial state s is *transfers* (*kinds as*) s, if *as* is the corresponding valid path between *Entry* and *Exit* in the CFG and *preds* (*kinds as*) s holds. As we argued above, assuring that *HRB-slice* of *Low* does not contain *High* suffices in proving noninterference of a program. Thus, we state the noninterference theorem for HRB slicing as follows:

Theorem 4. *Path Noninterference Theorem for HRB slicing:*

$$\frac{s \approx_L s' \qquad \mathit{High} \notin \mathit{HRB\text{-}slice}\ \mathit{Low} \qquad \mathit{Entry} -as\rightarrow_{\surd}* \mathit{Exit} \qquad \mathit{preds}\ (\mathit{kinds}\ as)\ s \qquad \mathit{Entry} -as'\rightarrow_{\surd}* \mathit{Exit} \qquad \mathit{preds}\ (\mathit{kinds}\ as')\ s'}{\mathit{transfers}\ (\mathit{kinds}\ as)\ s \approx_L \mathit{transfers}\ (\mathit{kinds}\ as')\ s'}$$

Proof. The trick to prove this theorem is to argue in the sliced graph of *Low*. First, we split the paths as and as' into paths from *Entry* to *Low* and the no-op edges between *Low* and *Exit*. Then, we apply Thm. 1 to both trimmed paths. Thus, 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. Via Lem. 2 and 3, the first two premises and the paths from *Entry* to *Low*, we know that these values also agree 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. As the variables used in *Low* are exactly the L-variables, we obtain the conclusion. □

For some CFGs it is possible to prove that they agree to an operational semantics: program c evaluates in initial state s to a final statement c' and state s', written $\langle c,s\rangle \Rightarrow \langle c',s'\rangle$, if the CFG for this program has a path from *Entry* to *Exit* such that traversing this path in s also yields s'. Then, we obtain a theorem that connects HRB slicing to the standard semantic definition for noninterference as a corollary from Thm. 4:

Theorem 5. *Noninterference Theorem for HRB slicing:*

$$\frac{s_1 \approx_L s_2 \qquad \mathit{High} \notin \mathit{HRB\text{-}slice}\ \mathit{Low} \qquad \langle c,s_1\rangle \Rightarrow \langle c',{s_1}'\rangle \qquad \langle c,s_2\rangle \Rightarrow \langle c',{s_2}'\rangle}{{s_1}' \approx_L {s_2}'}$$

4.4 Lifting Arbitrary Framework Graphs

In general, few CFGs that are in the framework will fulfil the assumptions used in the proof above, as they do not have the required *High* and *Low* nodes. However, we can lift any arbitrary CFG from the framework such that the above requirements hold. Note that the partition in H- and L-variables must still be provided. First, we relabel the *Entry* node to *High* and the *Exit* node to *Low*. Next, we adapt its *Def* and *Use* set, so that *High* defines and uses all H-variables, *Low* uses all L-variables. We add two additional nodes, *NewEntry* and *NewExit*, the new entry and exit nodes of the program, with empty *Def* and *Use* sets. They connect to *High* and *Low* with the required intraprocedural no-op edges. We proved that this lifted graph fulfils the assumptions for the correctness result from above if its original graph is a valid CFG in the framework.

5 Language Instantiations

To instantiate a locale, we have to provide formal function definitions, which match the fixed signatures, and prove that these functions fulfil the requirements posed in the locale. To show that the framework from Sec. 3 is indeed a valid abstraction, and that the chosen modularization is useful, we instantiated it with two formalizations of different languages, a simple while language with procedures called *Proc* and the quite sophisticated object oriented byte code language of *Jinja*. Performing these instantiations often helped in identifying wrong or too restricted assumptions on the abstract CFG; we even found a new necessary requirement we forgot to include in the first place.

Proc is a simple language with boolean and integer expressions, variable assignment, sequential composition, if-then-else, while-loops and procedure calls. The CFG for a concrete program consists of nodes, represented by a pair of a procedure name and a unique identifier within the procedure, and the edges connecting these nodes. Those are defined in two stages: first, we define all intraprocedural edges and connect call and matching return nodes with dummy edges. In the second step, we add the interprocedural edges. We replace the dummy edges from the first step with call and return edges to the corresponding procedure, taking care that the nodes get tuples with the right procedure. The edges we obtain from this construction are the *valid-edge*s.

The state in the framework is a stack (i.e. a list) of call frames. Each call frame consists (i) of a mapping from variables, identified via *string*s, to values, and (ii) of the return information, i.e. the return node of the respective call. Variables that get assigned are in the *Def*, those used in assignments, predicates or calls in the *Use* set.

In Proc, the procedure list is a list of 4-tuples, each contains a procedure name, the names of its formal in and out parameters, and the method body. To instantiate the framework, we require the program and its procedure list to be well-formed: the main procedure may not be in the procedure list, its first entries have to be distinct, the length of the actual and formal in as well as out parameters have to match, etc.

The most complicated part in this instantiation was to find the right structures and definitions, e.g. for the CFG, the *Def* and *Use* sets, or which programs are considered well-formed. With the right structures and definitions, the proofs that they fulfil the requirements of the abstract framework CFG were mostly straightforward; however, complete automatisation was not possible, as some fine details in the induction cases differed.

Jinja [15] models a large subset of the Java language, including an 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, both verified. Jinja is fully object oriented and features exception throwing and catching.

We were surprised how few adaptations we had to make w.r.t. the instantiation of the intraprocedural framework as presented in [29]. The main difference is that nodes are no longer identified via whole call stack (which was necessary for method inlining), but a mere call frame is sufficient. Call frames are triples $(cname, mname, pc)$, where *cname* and *mname* are the names of the class and method, and *pc* identifies a position in the method's instruction list. Hence, every node identifies exactly one instruction.

Changes to the edges were minimal, with the exception of method calls, of course. Instead of simulating them with update and predicate edges, call and return edges are now modelled directly. Late binding is modelled using the call edge predicate, which determines the dynamic callee. As return information, we save the node that identifies the next instruction after the call. INVOKE may throw an exception either because the receiver object is `null` or the callee throws an exception; hence, exception handling and/or propagation need careful modelling.

The actual-in parameters of a method with n parameters are always the top $n + 1$ elements of the local stack (including the callee's **`this`** pointer), formal-in parameters are local variables. The actual- and formal-out parameter of a method is the top element of the stack (note that in Jinja a **`void`** method returns the dummy element *Unit*). Besides these parameters every method also has an implicit in and out parameter for the heap as well as an out parameter signalling an exception. Always passing the whole heap makes the computed slice and thus the noninterference criterion object insensitive. Again, adding a points-to analysis would improve precision here.

Defining the *Def* and *Use* sets is completely analogous to [29], for *ParamDefs* and *ParamUses* it is quite straightforward. Most of the proofs which guarantee that a Jinja byte code CFG conforms to an abstract CFG as required in the framework are done by case distinction, some require an induction on

	Framework	**Noninterference**			**Instantiation**			
		Proof		Lifting	While		Proc	Jinja byte code
Intraproc. ([29])	5,200	500		1,500	2,400		–	2,900
Interproc. (this work)	19,200	1,400		2,000	–		6,700	3,400

Figure 5: Comparing the sizes of the intra- and interprocedural formalization parts

the CFG's structure.

6 Evolution of the Formalization

Fig. 5 compares the sizes of the formalization, the noninterference results and the instantiations in lines of code with our previous work, which focused on static intraprocedural slicing [29]. While we could reuse some parts of that work, the size of the framework grew by a factor of 4. This is mainly due to the complicated formalizations of SDGs with summary edges and the algorithm of Horwitz, Reps, and Binkley (together with a precision proof, not presented in this work, see [27]), as well as the lifting of the correctness proof to the interprocedural case.

The proof that slicing can guarantee noninterference grew by a factor of 3, the lifting only by 30%. In both cases, intra- and interprocedural, the proof is quite short compared to the framework size. We hope that the correctness results of the framework turns out to be a benefit in proving other slicing related issues, similar to IFC noninterference as shown here.

As for the instantiations, some surprising facts can be seen. Note that Proc is essentially While plus procedures, hence we instantiated the framework presented here with the former, the one presented in [29] with the latter. Adding procedures to While – thus gaining Proc – resulted in an instantiation size increase by factor 2.8. However, for Jinja the migration from intra- to interprocedural led to an increase of the instantiation size of only $\frac{1}{6}$. This is likely due to the fact that byte code languages reflect the corresponding CFG structure very well in the sense that nodes correspond to instruction positions in a rather natural way.

Is the approach of using a language independent framework to prove the correctness of slicing actually wise? In the intraprocedural case, each instantiation we showed (While and Jinja) took about 50% of the size of the framework. Now, in the interprocedural case, we were able to push this factor down to $\frac{1}{3}$ for Proc and even to $\frac{1}{5}$ for Jinja; hence, instantiating the framework with a new language is indeed much easier than redoing the whole proof.

7 Related Work

Modularized Proofs. The ability to modularize proofs is not unique to locales in Isabelle, other theorem provers provide comparable tools, cf. parametric theories in PVS [20] and modules in Coq [9]. Similar to our work, other approaches use these means to prove properties abstracting from concrete program instances.

The Coq part in the tool KRAKATOA [17] – a tool for verifying that a Java program meets its JML specification – uses the module concept to abstract from a concrete Java program. KRAKATOA ensures that the properties proven for the abstract instance, i.e. a signature representing the class structure of an arbitrary Java program, also hold for any special program.

Information Flow Noninterference using Proof Assistants. Most works that formalize information flow noninterference in a proof assistant that we are aware of are information flow type systems; usually, they use the term noninterference in the same sense as we do. We focus on interprocedural works.

The IFC type system of Banerjee and Naumann [2] 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 [18] and in Isabelle/HOL by Strecker [24].

Using Isabelle/HOL, Beringer and Hofmann [7] formalize and prove correct different type systems, starting with the Volpano-Smith-Irvine type system [26] for a while language, which they extended with an additional simple call rule to procedures without parameters. They augment this type system with Hunt's and Sand's flow types [14] and objects as covered in [4]. Instead of the usual noninterference definition, which compares two program runs, they use a security property which captures noninterference formally with only a single execution.

In [3], Barthe et al. define a noninterference byte code verifier for a Java-like language. They base on an existing formalization of a Java-like byte code language in Coq, called Bicolano, which is quite similar to the one formalized in Jinja, but features arrays. The noninterference verifier consists of three parts: (i) a pre-analyzer which computes information to reduce the control flow graph, (ii) an analyzer for control dependence regions, and (iii) the actual information flow analyzer, which leverages Kildall's data flow algorithm to compute for every program point its security environment. The latter two are proved correct in Coq, whereas the correctness of the pre-analyzer is assumed; integrating a machine-checked correctness proof for it, for which parts already exist [8], is left for future work.

In [10], Darvas et al. formalize noninterference in dynamic logic and include it in the theorem prover KeY [6], which handles JAVA CARD programs. Noninterference is encoded in two formulas, first a formalization of classical noninterference as presented in this paper, and second, an equivalent formula $\forall \mathtt{l}.\exists r.\forall \mathtt{h}.\langle \mathtt{p}\rangle r \doteq \mathtt{l}$. This formula states that "when starting (terminating program) p with arbitrary values l, then the value r after executing p is independent of the choice of h." Then, they were able to prove programs like $\mathtt{l} = \mathtt{h}; \mathtt{l} = \mathtt{l} - \mathtt{h};$ or $\mathtt{if}\ (\mathtt{false})\ \mathtt{l} = \mathtt{h};$ secure, which are rejected by most type systems and even by slicing based approaches. They present extensions for proving insecurity and handling exceptions and (forms of) declassification. However, the examples presented in this work are very small, the authors are aware of the problem if this approach scales to more complex programs with several high and low variables.

8 Conclusion and Future Work

We presented a machine-checked correctness proof that interprocedural slicing is able to guarantee IFC noninterference. It reuses a correctness proof of the interprocedural slicing algorithm of Horwitz, Reps, and Binkley. The modularity employed in the underlying framework allows instantiations with different languages; we showed this for a simple While language with procedures and a sophisticated object-oriented byte code language.

As we now have verified context-sensitive interprocedural slicing and were even able to provide first noninterference results, we made significant progress for our ultimate aim, the verification of the slicing based IFC algorithm of [12]. However, to fully guarantee its flow-, object-, and context-sensitivity, and to be able to argue about interior security level changes, we currently work on a new definition of noninterference which is able to capture these effects.

References

[1] Anderson, P., Reps, T., Teitelbaum, T.: Design and implementation of a fine-grained software inspection tool. IEEE Transactions on Software Engineering 29(8), 721–733 (2003)

[2] Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a Java-like language. In: Proc. of CSFW '02, pp. 239 – 253. IEEE (2002)

[3] Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. In: ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer (2007)

[4] Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: Proc. of TLDI '05, pp. 103–112. ACM (2005)

[5] Bates, S., Horwitz, S.: Incremental program testing using program dependence graphs. In: Proc. of POPL '93, pp. 384–396. ACM (1993)

[6] Beckert, B., Hähnle, R., Schmitt, P. H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS, vol. 4334. Springer (2007)

[7] Beringer, L., Hofmann, M.: Secure information flow and program logics. In: Proc. of CSF '07, pp. 133–148. IEEE (2007)

[8] Besson, F., Jensen, T., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. Theoretical Computer Science 364(3), 273–291 (2006)

[9] Chrzaszcz, J.: Implementing Modules in the Coq System. In Basin, D. Wolff, B. (eds.) TPHOLS 2003. LNCS, vol. 2758, pp. 270–286. Springer (2003)

[10] Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer (2005)

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

[12] Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 8(6), 399–422 (2009)

[13] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM TOPLAS 12(1), 26–60 (1990)

[14] Hunt, S., Sands, D.: On flow-sensitive security types. In: Proc. of POPL '06, pp. 79–90. ACM (2006)

[15] Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM TOPLAS 28(4), 619–695 (2006)

[16] Krinke, J.: Program Slicing. In: Handbook of Software Engineering and Knowledge Engineering, Vol. 3: Recent Advances, pp. 307–332. World Scientific Publishing (2005)

[17] Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. Journal of Logic and Algebraic Programming 58, 89–106 (2004)

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

[19] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer (2002)

[20] Owre, S., Shankar, N.: Theory Interpretation in PVS. Technical Report SRI-CSL-01-01, SRI International (2001)

[21] Reps, T., Horwitz, S., Sagiv, M., Rosay, G.: Speeding up slicing. In: Proc. of SIGSOFT '94, pp. 11–20. ACM (1994)

[22] Snelting, G., Robschink, T., Krinke, J.: Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. ACM TOSEM 15(4), 410–457 (2006)

[23] Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: Proc. of PLDI '07, pp. 112–122. ACM (2007)

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

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

[26] Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. Journal of Computer Security 4(2-3), 167–187 (1996)

[27] Wasserrab, D.: Formalizing context-sensitive slicing: Verifying the Horwitz-Reps-Binkley algorithm in Isabelle/HOL. Submitted for publication.

[28] Wasserrab, D.: Backing up slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley slicer. In: The Archive of Formal Proofs. `http://afp.sf.net/entries/HRB-Slicing.shtml` (2009) Formal proof development.

[29] Wasserrab, D., Lohner, D., Snelting, G.: On PDG-based noninterference and its modular proof. In: Proc. of PLAS '09, pp. 31–44. ACM (2009)

[30] Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison-Wesley (1995)