Types vs. PDGs in Information Flow Analysis

Heiko Mantel and Henning Sudbrock

Computer Science Department, TU Darmstadt, Germany
{mantel,sudbrock}@mais.informatik.tu-darmstadt.de

Abstract Type-based and PDG-based information flow analysis techniques are currently developed independently in a competing manner, with different strengths regarding coverage of language features and security policies. In this article, we study the relationship between these two approaches. One key insight is that a type-based information flow analysis need not be less precise than a PDG-based analysis. For proving this result we establish a formal connection between the two approaches which can also be used to transfer concepts from one tradition of information flow analysis to the other. The adoption of rely-guarantee-style reasoning from security type systems, for instance, enabled us to develop a PDG-based information flow analysis for multi-threaded programs.

Keywords: Information flow security, Security type system, Program dependency graph

1 Introduction

When giving a program access to confidential data one wants to be sure that the program does not leak any secrets to untrusted sinks, like, e.g., to untrusted servers on the Internet. Such confidentiality requirements can be characterized by information flow properties. For verifying that a program satisfies an information flow property, a variety of program analysis techniques can be employed.

The probably most popular approach to information flow analysis is the use of security type systems. Starting with [25], type-based information flow analyses were developed for programs with various language features comprising procedures (e.g., [24]), concurrency (e.g., [23]), and objects (e.g., [16]). Security type systems were proposed for certifying a variety of information flow properties, including timing-sensitive and timing-insensitive properties (e.g., [22] and [2]) and properties supporting declassification (e.g., [14]).

Besides type systems, one can also employ other program analysis techniques for certifying information flow security. For instance, it was proposed in [10] to use program dependency graphs (PDGs) for information flow analysis. A PDG [4] is a graph-based program representation that captures dependencies caused by the data flow and the control flow of a program. PDG-based information flow analyses recently received new attention, resulting in, e.g., a PDG-based information flow analysis for object-oriented programs and a PDG-based information flow analysis supporting declassification [8,7].

To appear in:
E. Albert (Ed.): LOPSTR 2012, LNCS 7844, pp. 106–121, 2013.

Type-based and PDG-based information flow analyses are currently developed independently. The two sub-communities both see potential in their approach, but the pros and cons of the two techniques have not been compared in detail. In this article, we compare type-based and PDG-based information flow analyses with respect to their precision. Outside the realm of information flow security there already exist results that compare the precision of data-flow oriented and type-based analyses, for instance, for safety properties [18,17]. Here, we clarify the relation between type-based and PDG-based analyses in the context of information flow security. We investigate whether (a) one approach has superior precision, (b) both have pros and cons, or (c) both are equally precise. To be able to establish a precise relation, we consider two prominent analyses that are both fully formalized for a simple while language, namely the type-based analysis from Hunt and Sands [11] and the PDG-based analysis from Wasserrab, Lohner, and Snelting [27].

Our main result is that the two analyses have exactly the same precision. This result was surprising for us, because one motivation for using PDGs in an information flow analysis was their precision [8]. We derive our main result based on a formal connection between the two kinds of security analyses, which we introduce in this article. It turned out that this connection is also interesting in its own right, because it can be used for transferring ideas from type-based to PDG-based information flow analyses and vice versa. In this article, we illustrate this possibility in one direction, showing how to derive a novel PDG-based information flow analysis that is suitable for multi-threaded programs by exploiting our recently proposed solution for rely-guarantee-style reasoning in a type-based security analysis [15]. The resulting analysis is compositional and, thereby, enables a modular security analysis. This is an improvement over the analysis from [5], the only provably sound PDG-based information flow analysis for multi-threaded programs developed so far. Moreover, in contrast to [5] our novel analysis supports programs with nondeterministic public output.

In summary, the main contributions of this article are

1. the formal comparison of the precision of a type-based and a PDG-based information flow analysis, showing that they have the same precision;
2. the demonstration that our formal connection between the type-based and the PDG-based analysis can be used to transfer concepts from one approach to the other (by transferring rely-guarantee-style reasoning as mentioned above); and
3. a provably sound PDG-based information flow analysis for multi-threaded programs that is compositional with respect to the parallel composition of threads and compatible with nondeterministic public output.

We believe that the connection between type- and PDG-based information flow analysis can serve as a basis for further mutual improvements of the analysis techniques. Such a transfer is desirable because there are other relevant aspects than an analysis' precision like, e.g., efficiency and availability of tools. Moreover, we hope that the connection between the two approaches to information flow analysis fosters mutual understanding and interaction between the two communities.

2 Type-based Information Flow Analyses

If one grants a program access to secret information, one wants to be sure that the program does not leak secrets to untrusted sinks like, e.g., untrusted servers in a network. A secure program should not only refrain from directly copying secrets to untrusted sinks (as, e.g., with an assignment "$sink{:=}secret$"), but also should not reveal secrets indirectly (as, e.g., by executing "$\mathsf{if}\ (secret > 0)\ \mathsf{then}\ sink{:=}1$").

It is popular to formalize information flow security by the property that values written to public sinks do not depend on secrets, the probably best known such property being *Noninterference* [6,13]. In the following, we define information flow security for programs by such a property, and we present a security type system for certifying programs with respect to this property.

2.1 Execution Model and Security Property

We consider a set of *commands Com* that is defined by the grammar

$$c ::= \mathsf{skip} \mid x{:=}e \mid c; c \mid \mathsf{if}\ (e)\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od},$$

where $x \in Var$ is a variable and $e \in Exp$ is an expression. *Expressions* are terms built from variables and from operators that we do not specify further. The set of *free variables* in expression $e \in Exp$ is denoted with $fv(e)$. A *memory* is a function $mem : Var \to Val$ that models a snapshot of a program's memory, where Val is a set of *values* and $mem(x)$ is the *value of* x. Judgments of the form $\langle c, mem\rangle \Downarrow mem'$ model program execution, with the interpretation that command c, if executed with initial memory mem, terminates with memory mem'. The rules for deriving the judgments are as usual for big-step semantics.[1]

To define the security property, we consider a *security lattice* $\mathcal{D} = \{l, h\}$ with two *security domains* where $l \sqsubseteq h$ and $h \not\sqsubseteq l$. This models the requirement that no information flows from domain h to domain l. This is the simplest policy capturing information flow security.[2] A *domain assignment* is a function $dom : Var \to \mathcal{D}$ that associates a security domain with each program variable. We say that variables in the set $L = \{x \in Var \mid dom(x) = l\}$ are public or *low*, and that variables in the set $H = \{x \in Var \mid dom(x) = h\}$ are secret or *high*. The resulting security requirement is that the final values of low variables do not depend on the initial values of high variables. This requirement captures security with respect to an attacker who sees the initial and final values of low variables, but cannot access values of high variables (i.e., access control works correctly).

Definition 1. *Two memories mem and mem' are* low-equal *(written $mem =_L mem'$) if and only if $mem(x) = mem'(x)$ for all $x \in L$.*

A command c is noninterferent *if whenever $mem_1 =_L mem_2$ and $\langle c, mem_1\rangle \Downarrow mem_1'$ and $\langle c, mem_2\rangle \Downarrow mem_2'$ are derivable then $mem_1' =_L mem_2'$.*

[1] The rules and detailed proofs of theorems in this article are available on the authors' website (`http://www.mais.informatik.tu-darmstadt.de/Publications`).

[2] The results in this article can be lifted to other security lattices.

$$[\mathsf{exp}]\frac{}{\Gamma \vdash e : \bigsqcup_{x \in fv(e)} \Gamma(x)} \qquad [\mathsf{if}]\frac{\Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma\,\{c_1\}\,\Gamma_1' \quad pc \sqcup t \vdash \Gamma\,\{c_2\}\,\Gamma_2'}{pc \vdash \Gamma\,\{\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\,\Gamma_1' \sqcup \Gamma_2'}$$

$$[\mathsf{assign}]\frac{\Gamma \vdash e : t}{pc \vdash \Gamma\,\{x{:=}e\}\,\Gamma[x \mapsto pc \sqcup t]} \qquad [\mathsf{seq}]\frac{pc \vdash \Gamma\,\{c_1\}\,\Gamma' \quad pc \vdash \Gamma'\,\{c_2\}\,\Gamma''}{pc \vdash \Gamma\,\{c_1; c_2\}\,\Gamma''}$$

$$[\mathsf{skip}]\frac{}{pc \vdash \Gamma\,\{\mathsf{skip}\}\,\Gamma} \qquad [\mathsf{while}]\frac{\begin{array}{c}\Gamma_i' \vdash e : t_i \quad pc \sqcup t_i \vdash \Gamma_i'\,\{c\}\,\Gamma_i'' \quad 0 \le i \le k \\ \Gamma_0' = \Gamma \quad \Gamma_{i+1}' = \Gamma_i'' \sqcup \Gamma \quad \Gamma_{k+1}' = \Gamma_k'\end{array}}{pc \vdash \Gamma\,\{\mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od}\}\,\Gamma_k'}$$

Figure 1. Type system from [11]

```
1. if (z < 0) then
2.   while (y > 0) do
3.     y:=y + z od else
4.   skip fi;
5. x:=y
```

Example 1. Consider command c to the right and assume that $dom(x) = dom(y) = l$ and $dom(z) = h$. Consider furthermore low-equal memories mem_1 and mem_2 with $mem_1(x) = mem_2(x) = mem_1(y) = mem_2(y) = 1$, $mem_1(z) = -1$, and $mem_2(z) = 1$. Then $\langle c, mem_1\rangle \Downarrow mem_1'$ and $\langle c, mem_2\rangle \Downarrow mem_2'$ are derivable with $mem_1'(x) = 0$ and $mem_2'(x) = 1$. Since $mem_1' \neq_L mem_2'$ command c is not noninterferent.

2.2 The Type-based Information Flow Analysis by Hunt and Sands

A type-based analysis uses a collection of typing rules to inductively define a subset of programs. The intention is that every program in the subset satisfies an information flow property like, e.g., the one from Definition 1. Starting with [25], many type-based information flow analyses were developed (see [21] for an overview). Here, we recall the type system from Hunt and Sands [11] which is, unlike many other type-based security analyses, flow-sensitive (i.e., it takes the order of program statements into account to improve precision).

In [11], typing judgments have the form $pc \vdash \Gamma\,\{c\}\,\Gamma'$, where c is a command, $\Gamma, \Gamma' : Var \to \mathcal{D}$ are *environments*, and pc is a security domain. The interpretation of the judgment is as follows: For each variable $y \in Var$, $\Gamma'(y)$ is a valid upper bound on the security level of the value of y after command c has been run if (a) for each $x \in Var$, $\Gamma(x)$ is an upper bound on the security level of the value of x before c has been run and (b) pc is an upper bound on the security level of all information on which it might depend whether c is run.

The typing rules from [11] are displayed in Figure 1, where $\sqcup$ denotes the least upper bound operator on $\mathcal{D}$, which is extended to environments by $(\Gamma \sqcup \Gamma')(x) := \Gamma(x) \sqcup \Gamma'(x)$. The typing rules ensure that for any given c, Γ, and pc there is an environment Γ' such that $pc \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable. Moreover, this environment is uniquely determined by c, Γ, and pc [11, Theorem 4.1].

Definition 2. *Let $c \in Com$, $\Gamma(x) = dom(x)$ for all $x \in Var$, and Γ' be the unique environment such that $l \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable. Command c is* accepted by the type-based analysis *if $\Gamma'(x) \sqsubseteq dom(x)$ for all $x \in Var$.*

Example 2. Consider command c and domain assignment *dom* from Example 1. Let $\Gamma(x) = dom(x)$ for all $x \in Var$. Then the judgment $l \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable if and only if $\Gamma'(x) = \Gamma'(y) = \Gamma'(z) = h$ and $\Gamma'(x') = \Gamma(x')$ for all other $x' \in Var$. Since $\Gamma'(x) \not\sqsubseteq dom(x)$ command c is not accepted by the type-based analysis.

Theorem 1. *Commands accepted by the type-based analysis are noninterferent.*

The theorem follows from Theorem 3.3 in [11].

3 PDG-based Information Flow Analyses

PDG-based information flow analyses, firstly proposed in [10], exploit that the absence of certain paths in a *program dependency graph* (PDG) [4] is a sufficient condition for the information flow security of a program. In this section, we recall the PDG-based analysis from [27] which is sound with respect to the property from Definition 1. In order to make the article self-contained, we recall the construction of control flow graphs (CFGs) and PDGs in Sections 3.1 and 3.2.

3.1 Control Flow Graphs

Definition 3. *A* directed graph *is a pair* (N, E) *where* N *is a set of nodes and* $E \subseteq N \times N$ *is a set of edges. A* path p from node n_1 to node n_k *is a non-empty sequence of nodes* $\langle n_1, \ldots, n_k \rangle \in N^+$ *where* $(n_i, n_{i+1}) \in E$ *for all* $i \in \{1, \ldots, k-1\}$. *We call a path* trivial *if it is of the form* $\langle n \rangle$ *(i.e., a sequence of length* 1*), and* non-trivial *otherwise. Moreover, we say that node* n *is* on the path $\langle n_1, \ldots, n_k \rangle$ *if* $n = n_i$ *for some* $i \in \{1, \ldots, k\}$.

Definition 4. *A* control flow graph with def and use sets *is a tuple* (N, E, def, use) *where* (N, E) *is a directed graph,* N *contains two distinguished nodes start and stop, and* $def, use : N \to \mathcal{P}(Var)$ *are functions returning the def and use set, respectively, for a node. (The set* $\mathcal{P}(Var)$ *denotes the powerset of the set* Var*.)*

Nodes *start* and *stop* represent program start and termination, respectively, and the remaining nodes represent program statements and control conditions. An edge $(n, n') \in E$ models that n' might immediately follow n in a program run. Finally, the sets $def(n)$ and $use(n)$ contain all variables that are defined and used, respectively, at a node n. In the remainder of this article we simply write "CFG" instead of "CFG with def and use sets."

We recall the construction of the CFG for a command following [26], where statements and control conditions are represented by numbered nodes.

Definition 5. *We denote with* $|c|$ *the number of statements and control conditions of* $c \in Com$*, and define* $|c|$ *recursively by* $|\mathsf{skip}| = 1$, $|x{:=}e| = 1$, $|c_1; c_2| = |c_1| + |c_2|$, $|\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}| = 1 + |c_1| + |c_2|$*, and* $|\mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od}| = 1 + |c|$.

Definition 6. *For $c \in Com$ and $1 \leq i \leq |c|$ we denote with $c[i]$ the i^{th} statement or control condition in c, which we define recursively as follows: If $c = \mathsf{skip}$ or $c = x{:=}e$ then $c[1] = c$. If $c = c_1; c_2$ then $c[i] = c_1[i]$ for $1 \leq i \leq |c_1|$ and $c[i] = c_2[i - |c_1|]$ for $|c_1| < i \leq |c|$. If $c = \mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ then $c[1] = e$, $c[i] = c_1[i{-}1]$ for $1 < i \leq 1 + |c_1|$, and $c[i] = c_2[i{-}1{-}|c_1|]$ for $1 + |c_1| < i \leq |c|$. If $c = \mathsf{while}\ (e)\ \mathsf{do}\ c_1\ \mathsf{od}$ then $c[1] = e$ and $c[i] = c_1[i{-}1]$ for $1 < i \leq |c|$.*

Note that the ith statement or control condition, i.e., $c[i]$, is either an expression, an assignment, or a skip-statement.

Definition 7. *For $c \in Com$, $N_c = \{1, \ldots, |c|\} \cup \{start, stop\}$.*

We define an operator $\ominus : (\mathbb{N} \cup \{start, stop\}) \times \mathbb{N} \to \mathbb{Z} \cup \{start, stop\}$ by $n \ominus z = n - z$ if $n \in \mathbb{N}$ and $n \ominus z = n$ if $n \in \{start, stop\}$.

Definition 8. *For $c \in Com$ the set $E_c \subseteq N_c \times N_c$ is defined recursively by:*

- $E_{\mathsf{skip}} = E_{x:=e} = \{(start, 1), (1, stop), (start, stop)\}$,
- $E_{\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}} = \{(start, 1), (start, stop)\} \cup$
  $\{(1, n') \mid (start, n' \ominus 1) \in E_{c_1} \wedge n' \neq stop\} \cup$
  $\{(1, n') \mid (start, n' \ominus (1 + |c_1|)) \in E_{c_2} \wedge n' \neq stop\} \cup$
  $\{(n, n') \mid (n \ominus 1, n' \ominus 1) \in E_{c_1} \wedge n \neq start\} \cup$
  $\{(n, n') \mid (n \ominus (1 + |c_1|), n' \ominus (1 + |c_1|)) \in E_{c_2} \wedge n \neq start\}$,
- $E_{c_1;c_2} = \{(start, stop)\} \cup$
  $\{(n, n') \mid (n, n') \in E_{c_1} \wedge n' \neq stop\} \cup$
  $\{(n, n') \mid (n \ominus |c_1|, n' \ominus |c_1|) \in E_{c_2} \wedge n \neq start\} \cup$
  $\{(n, n') \mid (n, stop) \in E_{c_1} \wedge (start, n' \ominus |c_1|) \in E_{c_2} \wedge n \neq start \wedge n' \neq stop\}$, *and*
- $E_{\mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od}} = \{(start, 1), (start, stop)\} \cup$
  $\{(1, n') \mid (start, n' \ominus 1) \in E_c\} \cup$
  $\{(n', 1) \mid (n' \ominus 1, stop) \in E_c\} \cup$
  $\{(n, n') \mid (n \ominus 1, n' \ominus 1) \in E_c \wedge n \neq start \wedge n' \neq stop\}$.

Definition 9. *For $c \in Com$ we define $def_c : N_c \to \mathcal{P}(Var)$ by $def_c(n) = \{x\}$ if $n \in \{1, \ldots, |c|\}$ and $c[n] = x{:=}e$, and by $def_c(n) = \{\}$ otherwise. Moreover, we define $use_c : N_c \to \mathcal{P}(Var)$ by $use_c(n) = fv(e)$ if $n \in \{1, \ldots, |c|\}$ and $c[n] = x{:=}e$ or $c[n] = e$, and by $use_c(n) = \{\}$ otherwise.*

Definition 10. *The* control flow graph *of c is $CFG_c = (N_c, E_c, def_c, use_c)$.*

Note that, by definition, an edge from *start* to *stop* is contained in E_c. This edge models the possibility that c is not executed.

We now augment CFGs with def and use sets by two nodes *in* and *out* to capture the program's interaction with its environment. Two sets of variables $I, O \subseteq Var$, respectively, specify which variables may be initialized by the environment before program execution and which variables may be read by the environment after program execution. This results in the following variant of CFGs:

Definition 11. *Let $CFG = (N, E, def, use)$ and $I, O \subseteq Var$. Then $CFG^{I,O} = (N', E', def', use')$ where $N' = N \cup \{in, out\}$, $E' = \{(start, stop), (start, in),$*

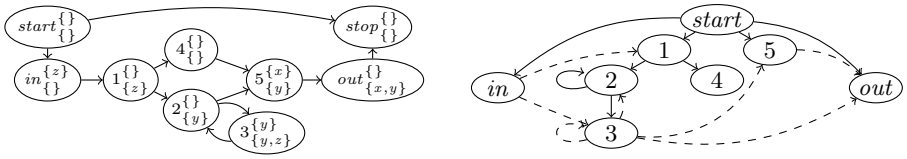

Figure 2. The CFG and the PDG for the command from Example 1

$(out, stop)\} \cup \{(in, n') \mid (start, n') \in E \wedge n' \neq stop\} \cup \{(n, out) \mid (n, stop) \in E \wedge n \neq start\} \cup \{(n, n') \in E \mid n \notin \{start, stop\} \wedge n' \notin \{start, stop\}\}$, $def'(in) = I$, $use'(in) = def'(out) = \{\}$, $use'(out) = O$, *and* $def'(n) = def(n)$ *and* $use'(n) = use(n)$ *for* $n \in N$.

Definitions 10 and 11 both augment the usual notion of control flow graphs (see Definition 4). In the remainder of this article, we use the abbreviation CFG for arbitrary control flow graphs (including those that satisfy Definition 10 or 11).

We use a graphical representation for displaying CFGs where we depict nodes with ellipses and edges with solid arrows. For each node n we label the corresponding ellipse with n^X_Y where $X = def(n)$ and $Y = use(n)$.

Example 3. Command c in Example 1 contains three statements and two control conditions (i.e., $|c| = 5$). Hence, $N_c = \{1, \ldots, 5, start, stop\}$. Nodes 1–5 represent the statements and control conditions in Lines 1–5 of the program, respectively. The control flow graph $CFG_c^{\{z\},\{x,y\}}$ is displayed at the left hand side of Figure 2.

3.2 The PDG-based Information Flow Analysis by Wasserrab et al

PDGs are directed graphs that represent dependencies in imperative programs [4]. PDGs were extended to programs with various languages features like procedures (e.g., [9]), concurrency (e.g., [3]), and objects (e.g., [8]). We recall the construction of PDGs from CFGs for the language from Section 2 based on the following notions of data dependency and control dependency.

Definition 12. *Let* (N, E, def, use) *be a CFG and* $n, n' \in N$. *If* $x \in def(n)$ *we say that* the definition of x at n reaches n' *if there is a non-trivial path* p *from* n *to* n' *such that* $x \notin def(n'')$ *for every node* n'' *on* p *with* $n'' \neq n$ *and* $n'' \neq n'$.

Node n' *is* data dependent *on node* n *if there exists* $x \in Var$ *such that* $x \in def(n)$, $x \in use(n')$, *and the definition of* x *at* n *reaches* n'.

Intuitively, a node n' is data dependent on a node n if n' uses a variable that has not been overwritten since being defined at n.

Example 4. Consider the CFG on the left hand side of Figure 2. The definition of y at Node 3 reaches Node 5 because $\langle 3, 2, 5\rangle$ is a non-trivial path and $y \notin def(2)$. Hence, Node 5 is data dependent on Node 3 because $y \in def(3)$, $y \notin def(2)$, and $y \in use(5)$. Note that Node 2 is also data dependent on Node 3, and that Node 3 is data dependent on itself.

Definition 13. *Let (N, E, def, use) be a CFG. Node n'* postdominates *node n if $n \neq n'$ and every path from n to stop contains n'.*

Node n' is control dependent *on node n if there is a non-trivial path p from n to n' such that n' postdominates all nodes $n'' \notin \{n, n'\}$ on p and n' does not postdominate n.*

Intuitively, a node n' is control dependent on a node n if n represents the innermost control condition that guards the execution of n'.

Example 5. Consider again the CFG in Figure 2. Node 5 postdominates Node 1 because Node 5 is on all paths from Node 1 to Node *stop*. Hence, Node 5 is not control dependent on Node 1. Nodes 2, 3, and 4 do not postdominate Node 1. Node 3 is not control dependent on Node 1 because Node 3 does not postdominate Node 2 and all paths from Node 1 to Node 3 contain Node 2. However, Node 3 is control dependent on Node 2. Moreover, Nodes 2 and 4 are control dependent on Node 1 because $\langle 1, 2\rangle$ and $\langle 1, 4\rangle$ are non-trivial paths in the CFG.

Definition 14. *Let $CFG = (N, E, def, use)$ be a control flow graph. The directed graph (N', E') is the* PDG of *CFG (denoted with $PDG(CFG)$) if $N' = N$ and $(n, n') \in E'$ if and only if n' is data dependent or control dependent on n in CFG.*

We use the usual graphical representation for displaying PDGs, depicting Node n with an ellipse labeled with n, edges that reflect control dependency with solid arrows, and edges that reflect data dependency with dashed arrows. Moreover, we do not display nodes that have neither in- nor outgoing edges.

Example 6. The PDG of the CFG at the left of Figure 2 is displayed right of the CFG. Node *stop* is not displayed because it has neither in- nor outgoing edges.

The PDG-based information flow analysis from Wasserrab et al [27] for a command c is based on the PDG of $CFG_c^{H,L}$ (cf. Definition 11).

Definition 15. *The command $c \in Com$ is* accepted by the PDG-based analysis *if and only if there is no path from in to out in $PDG(CFG_c^{H,L})$.*

Example 7. For command c and domain assignment *dom* from Example 1 the graph $PDG(CFG_c^{H,L})$ is displayed at the right of Figure 2. It contains a path from Node in to Node out, (e.g., the path $\langle in, 3, out\rangle$). In consequence, c is not accepted by the PDG-based analysis.

Theorem 2. *Commands accepted by the PDG-based analysis are noninterferent.*

The theorem follows from [27, Theorem 8].

4 Comparing the Type- and the PDG-based Analysis

While both the type-based analysis from Section 2 and the PDG-based analysis from Section 3 are sound, both analyses are also incomplete. I.e., for both

analyses there are programs that are noninterferent (according to Definition 1), but that are not accepted by the analysis. A complete analysis is impossible, because the noninterference property is undecidable (this can be proved in a standard way by showing that the decidability of the property would imply the decidability of the halting problem for the language under consideration [21]). This raises the question if one of the two analyses is more precise than the other. In this section, we answer this question. As an intermediate step, we establish a relation between the two analyses:

Lemma 1. *Let $c \in Com$, $y \in Var$, and Γ be an environment. Let Γ' be the unique environment such that $l \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable in the type system from Section 2. Moreover, let X be the set of all $x \in Var$ such that there exists a path from in to out in $PDG(CFG_c^{\{x\},\{y\}})$. Then $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$ holds.*

Proof sketch. We argue that the following more general statement holds: If the judgment $pc \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable, then the equality $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$ holds if there is no path from *start* to *out* in $PDG(CFG_c^{\{\},\{y\}})$ that contains a node $n \notin \{start, out\}$, and the equality $\Gamma'(y) = pc \sqcup \left(\bigsqcup_{x \in X} \Gamma(x)\right)$ holds if there is such a path. Intuitively, the absence of a path from *start* to *out* with more than two nodes guarantees that y is not changed during any execution of c, while y might be changed during an execution of c if a path from *start* to *out* with more than two nodes exists. Hence, the security domain pc (determined by the type-based analysis as an upper bound on the security level of all information on which it depends whether c is executed) is included in the formula for $\Gamma'(y)$ only if such a path exists in the PDG. Formally, the more general statement is proven by induction on the structure of the command c. A detailed proof is available on the authors' website. Lemma 1 follows from this more general statement by instantiating pc with the security level l. □

Lemma 1 is the key to establishing the following theorem that relates the precision of the type-based analysis to the precision of the PDG-based analysis, showing that the analyses have exactly the same precision.

Theorem 3. *A command $c \in Com$ is accepted by the type-based analysis if and only if it is accepted by the PDG-based analysis.*

Proof. Our proof is by contraposition. Let $\Gamma(x) = dom(x)$ for all $x \in Var$, and let Γ' be the unique environment such that $l \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable in the type system from Section 2. If c is not accepted by the type-based analysis then $dom(y) = l$ and $\Gamma'(y) = h$ for some $y \in Var$. Hence, by Lemma 1 there exists $x \in Var$ with $dom(x) = h$ and a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{\{x\},\{y\}})$. Hence, there is a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{H,L})$. Thus, c is not accepted by the PDG-based analysis. If c is not accepted by the PDG-based analysis then there is a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{H,L})$. But then there exist variables x, y with $dom(x) = h$ and $dom(y) = l$ such that there is a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{\{x\},\{y\}})$. Hence, by Lemma 1, $\Gamma'(y) = h$. Since $dom(y) = l$ it follows that $\Gamma'(y) \not\sqsubseteq dom(y)$. Thus, c is not accepted by the type-based analysis. □

Theorem 3 shows that the information flow analyses from [11] and [27] have exactly the same precision. More generally this means that, despite their conceptual simplicity, type-based information flow analyses need not be less precise than PDG-based information flow analyses.

Given that both analyses have equal precision, the choice of an information flow analysis should be motivated by other aspects. For instance, if a program's environment is subject to modifications, one might desire a compositional analysis, and, hence, choose a type-based analysis. On the other hand, if a program is not accepted by the analyses one could use the PDG-based analysis to localize the source of potential information leakage by inspecting the path in the PDG that leads to the rejection of the program.

Beyond clarifying the connection between type-based and PDG-based information flow analyses, Theorem 3 also provides a bridge that can be used to transfer concepts from the one tradition of information flow analysis to the other. In the following section, we exploit this bridge to transfer the concept of rely-guarantee-style reasoning for the analysis of multi-threaded programs from type-based to PDG-based information flow analysis.

5 Information Flow Analysis of Multi-threaded Programs

Multi-threaded programs may exhibit subtle information leaks that do not occur in single-threaded programs. Such a leak is illustrated by the following example.

Example 8. Consider two threads with shared memory that execute commands $c_1 = \mathsf{if}\ (x)\ \mathsf{then\ skip;\ skip\ else\ skip\ fi};\ y{:=}\mathsf{True}$ and $c_2 = \mathsf{skip;\ skip};\ y{:=}\mathsf{False}$, respectively, and that are run under a Round-Robin scheduler that selects them alternately starting with the first thread and rescheduling after each execution step. If initially $x = \mathsf{True}$ then c_1 assigns True to y after c_2 assigns False to y. Otherwise, c_1 assigns True to y prior to the assignment to y in c_2. I.e., the initial value of x is copied into y. Such leaks are also known as *internal timing leaks*.

Many type-based analyses detect such leaks (for instance, [23,22,28,15]). Regarding PDG-based analyses, this is only the case for a recently proposed analysis [5]. However, this analysis has serious limitations: It forbids publicly observable nondeterminism, and it is not compositional (cf. Section 6 for a more detailed comparison). This motivated us to choose this domain for illustrating how the connection between type-based and PDG-based information flow analysis (from Section 4) can be exploited to transfer ideas from the one analysis style to the other. More concretely, we show how rely-guarantee-style reasoning can be transferred from a type-based to a PDG-based information flow analysis. The outcome is a sound PDG-based information flow analysis for multi-threaded programs that is superior to the one in [5] in the sense that it supports publicly observable nondeterminism.

5.1 A Type-based Analysis for Multi-threaded Programs

We consider multi-threaded programs executing a fixed number of threads that interact via shared memory, i.e., configurations have the form $\langle (c_1, \ldots, c_k), mem \rangle$ where the commands c_i model the threads and *mem* models the shared memory.

In the following, we recall the type-based analysis from [15] that exploits rely-guarantee-style reasoning, where typing rules for single threads exploit assumptions about when and how variables might be accessed by other threads. Assumptions are modeled by *modes* in the set $Mod = \{asm\text{-}noread, asm\text{-}nowrite\}$, where *asm-noread* and *asm-nowrite* are interpreted as the assumption that no other thread reads and writes a given variable, respectively.[3] The language for commands from Section 2 is extended as follows with a notation for specifying when one starts and stops making assumptions for a thread, respectively:

$$ann ::= \mathsf{acq}(m, x) \mid \mathsf{rel}(m, x) \qquad c ::= \ldots \mid /\!/ann/\!/\ c,$$

where $m \in Mod$ and $x \in Var$. The annotations $/\!/\mathsf{acq}(m, x)/\!/$ and $/\!/\mathsf{rel}(m, x)/\!/$, respectively, indicate that an assumption for x is acquired or released.

Typing judgments for commands have the form $\vdash \Lambda\, \{c\}\, \Lambda'$ where $\Lambda, \Lambda' : Var \rightharpoonup \mathcal{D}$ are *partial environments*. Partial environments provide an upper bound on the security level only for low variables for which a no-read and for high variables for which a no-write assumption is made, respectively. For other variables, the typing rules ensure that $dom(x)$ is an upper bound on the security level of the value of x. We write $\Lambda\langle x \rangle$ for the resulting upper bound (defined by $\Lambda\langle x \rangle = \Lambda(x)$ if Λ is defined for x and by $\Lambda\langle x \rangle = dom(x)$ otherwise). This reflects that (a) low variables that might be read by other threads must not store secrets because the secrets might be leaked in other threads (i.e., the upper bound for low variables without no-read assumption must be l), and that (b) other threads might write secrets into high variables without no-write assumption (and, hence, the upper bound for high variables without no-write assumption cannot be l).

The security type system contains two typing rules for assignments:

$$\frac{\Lambda(x) \text{ is defined} \quad \Lambda' = \Lambda[x \mapsto (\bigsqcup_{y \in fv(e)} \Lambda\langle y \rangle)]}{\vdash \Lambda\, \{x{:=}e\}\, \Lambda'} \qquad \frac{(\bigsqcup_{y \in fv(e)} \Lambda\langle y \rangle) \sqsubseteq dom(x) \quad \Lambda(x) \text{ is not defined} \quad \Lambda' = \Lambda}{\vdash \Lambda\, \{x{:=}e\}\, \Lambda'}$$

The left typing rule is like in the type system from Section 2. The rule applies if Λ is defined for the assigned variable. The right typing rule reflects that if Λ is not defined for the assigned variable x then $dom(x)$ must remain an upper bound on the security level of the value of x, and, hence, only expressions that do not contain secret information may be assigned to x if $dom(x) = l$.

A slightly simplified[4] variant of the typing rule for conditionals is as follows:

$$\frac{\vdash \Lambda\, \{c_1\}\, \Lambda' \quad \vdash \Lambda\, \{c_2\}\, \Lambda' \quad l = \bigsqcup_{x \in fv(e)} \Lambda\langle x \rangle}{\vdash \Lambda\, \{\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\, \Lambda'}$$

[3] We omit the modes *guar-noread* and *guar-nowrite* representing guarantees from [15], because they are irrelevant for the security type system.

[4] The original rule from [15] permits that guards depend on secrets (i.e., $h = \bigsqcup_{x \in fv(e)} \Lambda\langle x \rangle$) if the branches are in a certain sense indistinguishable.

In contrast to the corresponding typing rule in Section 2, the guard is required to be low. This ensures that programs with leaks like in Example 8 are not typable.

For the complete set of typing rules we refer to [15].

Remark 1. In contrast to the type system in Section 2 the security level *pc* is not considered here, because the typing rules ensure that control flow does not depend on secrets (permitting some exceptions as indicated in Footnote 4).

Definition 16. *A multi-threaded program consisting of commands* $c_1, \ldots, c_k$ *is* accepted by the type-based analysis for multi-threaded programs *if the judgment* $\vdash \Lambda_0\ \{c_i\}\ \Lambda_i'$ *is derivable for each* $i \in \{1, \ldots, k\}$ *for some* Λ_i' *(where* Λ_0 *is undefined for all* $x \in Var$*) and the assumptions made are valid for the program.*

Validity of assumptions is formalized in [15] by *sound usage of modes*, a notion capturing that the assumptions made for single threads are satisfied in any execution of the multi-threaded program. Theorem 6 in [15] ensures that the type-based analysis for multi-threaded programs is sound with respect to *SIFUM-security*, an information flow security property for multi-threaded programs. We refer the interested reader to [15] for the definitions of sound usage of modes and of SIFUM-security.

5.2 A Novel PDG-based Analysis for Multi-threaded Programs

We define a PDG-based analysis for multi-threaded programs by transferring rely-guarantee-style reasoning from the type-based analysis (Definition 16) to PDGs. To this end, we augment the set of edges of the program dependency graph $PDG(CFG_c^{H,L})$, obtaining a novel program dependency graph $PDG^{\parallel}(CFG_c^{H,L})$. Using this graph, the resulting analysis for multi-threaded programs is as follows:

Definition 17. *A multi-threaded program consisting of commands* $c_1, \ldots, c_k$ *is* accepted by the PDG-based analysis for multi-threaded programs *if there is no path from in to out in* $PDG^{\parallel}(CFG_{c_i}^{H,L})$ *for each* $i \in \{1, \ldots, k\}$ *and the assumptions made are valid for the program.*

It follows from Definition 17 that the analysis is compositional with respect to the parallel composition of threads.

We now define the graph $PDG^{\parallel}(CFG_c^{H,L})$, where the additional edges in $PDG^{\parallel}(CFG_c^{H,L})$ model dependencies for nodes *in* and *out* that result from the concurrent execution of threads that respect the assumptions made for c.

Definition 18. *If command* c *is not of the form* $/\!/ann/\!/ c'$ *we say that* c does not acquire $m \in Mod$ for $x \in Var$ *and that* c does not release $m \in Mod$ for $x \in Var$*. Moreover, if an arbitrary command* c *does not release* m *for* x *then the command* $/\!/\mathsf{acq}(x, m)/\!/ c$ acquires m for x*, and if* c *does not acquire* m *for* x *then the command* $/\!/\mathsf{rel}(x, m)/\!/ c$ releases m for x*.*

For $c \in Com$ *we define the function* $modes_c : (N_c \times Mod) \to \mathcal{P}(Var)$ *by* $x \in modes_c(n, m)$ *if and only if for all paths* $p = \langle start, \ldots, n\rangle$ *in* CFG_c *there is a node* n' *on* p *such that* $c[n']$ *acquires* m *for* x*, and if* n'' *follows* n' *on* p *then* $c[n'']$ *does not release* m *for* x*.*

Definition 19. *Let $c \in Com$. Then $PDG^{\parallel}(CFG_c^{H,L}) = (N, E \cup E')$ for $(N, E) = PDG(CFG_c^{H,L})$ and $(n, n') \in E'$ if and only if one of the following holds:*

1. *$n = in$ and there exist a variable $x \in H \cap use_c(n')$, a node $n'' \in N$ with $x \notin modes_c(n'', \textit{asm-nowrite})$, and a path p from n'' to n' in $CFG_c^{H,L}$ with $x \notin def_c(n''')$ for every node n''' on p with $n''' \neq n''$ and $n''' \neq n'$,*
2. *$n' = out$ and there exist a variable $x \in L \cap def_c(n)$, a node $n'' \in N$ with $x \notin modes_c(n'', \textit{asm-noread})$, and a path p from n to n'' in $CFG_c^{H,L}$ such that $x \notin def_c(n''')$ for every node n''' on p with $n''' \neq n$ and $n''' \neq n''$, or*
3. *$n \in \{1, \ldots, |c|\}$, $c[n] \in Exp$, and $n' = out$.*

The edges defined in Items 1 and 2 are derived from the typing rules for assignments: The edge (in, n) in Item 1, where n uses a high variable whose value might have been written by another thread, captures that the high variable might contain secrets when being used at n. The edge (n, out) in Item 2, where n defines a low variable whose value might be eventually read by another thread, ensures that the command is rejected if the definition at n might depend on secret input (because then there is a path from in to n). The edges defined in Item 3 are derived from the typing rule for conditionals: If the guard represented by Node n depends on secrets (i.e., there is a path from in to n) then the command is rejected because together with the edge (n, out) there is a path from in to out.

Example 9. Consider the following command c where the security domains of variables are $dom(x) = l$ and $dom(y) = h$:

$$/\!\!/\mathsf{acq}(\textit{asm-noread}, x)/\!\!/; x{:=}y; x{:=}0; /\!\!/\mathsf{rel}(\textit{asm-noread}, x)/\!\!/$$

Then $PDG^{\parallel}(CFG_c^{H,L}) = PDG(CFG_c^{H,L})$, and c is accepted by the PDG-based analysis for multi-threaded programs. Let furthermore $c' = x{:=}y; x{:=}0$. Then $PDG^{\parallel}(CFG_{c'}^{H,L}) \neq PDG(CFG_c^{H,L})$, because the graph $PDG^{\parallel}(CFG_{c'}^{H,L})$ contains an edge from the node representing $x{:=}y$ to Node out (due to Item 2 in Definition 19). Hence, c' is not accepted, because $PDG^{\parallel}(CFG_{c'}^{H,L})$ contains a path from Node in to Node out via the node representing the assignment $x{:=}y$.

Not accepting c' is crucial for soundness because another thread executing $x'{:=}x$ could copy the intermediate secret value of x into a public variable x'.

Theorem 4. *If a multi-threaded program is accepted by the PDG-based analysis for multi-threaded programs then the program is accepted by the type-based analysis for multi-threaded programs.*

The proof is by contradiction; it exploits the connection between PDG-based and type-based analysis stated in Lemma 1. A detailed proof is available on the authors' website.[5]

Soundness of the PDG-based analysis follows directly from Theorem 4 and the soundness of the type-based analysis (see [15, Theorem 6]).

[5] The reverse direction of Theorem 4 does not hold, because the type-based analysis for multi-threaded programs classifies some programs with secret control conditions as secure that are not classified as secure by our PDG-based analysis.

6 Related Work

We focus on related work covering flow-sensitive type-based analysis, PDG-based analysis for concurrent programs, and connections between analysis techniques. For an overview on language-based information flow security we refer to [21].

Flow-sensitivity of type-based analyses. In contrast to PDG-based information flow analyses, many type-based information flow analyses are not flow-sensitive. The first flow-sensitive type-based information flow analysis is due to Hunt and Sands [11]. Based on the idea of flow-sensitive security types from [11], Mantel, Sands, and Sudbrock developed the first sound flow-sensitive security type system for concurrent programs [15].

PDG-based analyses for concurrent programs. Hammer [7] presents a PDG-based analysis for concurrent Java programs, where edges between the PDGs of the individual threads are added following the extension of PDGs to concurrent programs from [12]. However, there is no soundness result. In fact, since the construction of PDGs from [12] does not capture dependencies between nodes in the PDG that result from internal timing (cf. Example 8), the resulting PDG-based information flow analysis fails to detect some information leaks.

Giffhorn and Snelting [5] present a PDG-based information flow analysis for multi-threaded programs that does not accept programs with internal timing leaks. The analysis enforces an information flow property defined in the tradition of observational determinism [20,28], and, therefore, does not accept any programs that have nondeterministic public output. Hence, the analysis forbids useful nondeterminism, which occurs, for instance, when multiple threads append entries to the same log file. Our novel analysis (from Section 5) permits concurrent writes to public variables and, hence, accepts secure programs that are not accepted by the analysis from [5]. Moreover, in contrast to the analysis from [5] our novel analysis is compositional with respect to the parallel composition of threads.

Connections between different analysis techniques. Hunt and Sands show in [11] that the program logic from [1] is in fact equivalent to the type-based analysis from [11]. Rehof and Fähndrich [19] exploit concepts from PDGs (the computation of so-called summary edges for programs with procedures) in a type-based flow analysis. In this article, we go a step further by establishing and exploiting a formal connection between a type-based and a PDG-based analysis.

7 Conclusion

While security type systems are established as analysis technique for information flow security, information flow analyses based on program dependency graphs (PDGs) have only recently received increased attention. In this article, we investigated the relationship between these two alternative approaches.

As a main result, we showed that the precision of a prominent type-based information flow analysis is not only roughly similar to the precision of a prominent

PDG-based analysis, but that the precision is in fact exactly the same. Moreover, our result provides a bridge for transferring techniques and ideas from one tradition of information flow analysis to the other. This is an interesting possibility because there are other relevant attributes than the precision of an analysis (e.g., efficiency and the availability of tools). We showed at the example of rely-guarantee-style information flow analysis for multi-threaded programs that this bridge is suitable to facilitate learning by one sub-community from the other.

We hope that our results clarify the relationship between the two approaches. The established relationship could be used as a basis for communication between the sub-communities to learn from each other and to pursue joint efforts to make semantically justified information flow analysis more practical. For instance, our results give hope that results on controlling declassification with security type systems can be used to develop semantic foundations for PDG-based analyses that permit declassification. Though there are PDG-based analyses that permit declassification (e.g., [8]), all of them yet lack a soundness result, and, hence, it is unclear which noninterference-like property they certify.

Acknowledgment. We thank the anonymous reviewers for their valuable suggestions. This work was funded by the DFG under the project FM-SecEng in the Computer Science Action Program (MA 3326/1-3).

References

1. Amtoft, T., Banerjee, A.: Information Flow Analysis in Logical Form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer (2004)
2. Boudol, G., Castellani, I.: Noninterference for Concurrent Programs and Thread Systems. Theoretical Computer Science 281(1–2), 109–130 (2002)
3. Cheng, J.: Slicing Concurrent Programs - A Graph-Theoretical Approach. In: Fritszon, P. (ed.) AADEBUG 1993. LNCS, vol. 749, pp. 223–240. Springer (1993)
4. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The Program Dependence Graph and its Use in Optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)
5. Giffhorn, D., Snelting, G.: Probabilistic Noninterference Based on Program Dependence Graphs. Tech. Rep. 6, Karlsruher Institut für Technologie (KIT) (2012)
6. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: IEEE Symposium on Security and Privacy. pp. 11–20 (1982)
7. Hammer, C.: Information Flow Control for Java. Ph.D. thesis, Universität Karlsruhe (TH) (2009)
8. 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)
9. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Languages and Systems 12(1), 26–60 (1990)
10. Hsieh, C.S., Unger, E.A., Mata-Toledo, R.A.: Using Program Dependence Graphs for Information Flow Control. Journal of Systems and Software 17(3), 227–232 (1992)

11. Hunt, S., Sands, D.: On Flow-Sensitive Security Types. In: ACM Symposium on Principles of Programming Languages. pp. 79–90 (2006)
12. Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs. Ph.D. thesis, Universität Passau (2003)
13. Mantel, H.: Information Flow and Noninterference. In: Encyclopedia of Cryptography and Security (2nd Ed.), pp. 605–607. Springer (2011)
14. Mantel, H., Sands, D.: Controlled Declassification based on Intransitive Noninterference. In: Chin, W.N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 129–145. Springer (2004)
15. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and Guarantees for Compositional Noninterference. In: IEEE Computer Security Foundations Symposium. pp. 218–232 (2011)
16. Myers, A.C.: JFlow: Practical Mostly-Static Information Flow Control. In: ACM Symposium on Principles of Programming Languages. pp. 228–241 (1999)
17. Naik, M., Palsberg, J.: A Type System Equivalent to a Model Checker. ACM Transactions on Programming Languages and Systems 30(5), 1–24 (2008)
18. Palsberg, J., O'Keefe, P.: A Type System Equivalent to Flow Analysis. In: ACM Symposium on Principles of Programming Languages. pp. 367–378 (1995)
19. Rehof, J., Fähndrich, M.: Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In: ACM Symposium on Principles of Programming Languages. pp. 54–66 (2001)
20. Roscoe, A.W., Woodcock, J.C.P., Wulf, L.: Non-interference through Determinism. In: Gollmann, D. (ed.) ESORICS 1994. LNCS, vol. 875, pp. 33–53 (1994)
21. Sabelfeld, A., Myers, A.C.: Language-based Information-Flow Security. IEEE Journal on Selected Areas in Communication 21(1), 5–19 (2003)
22. Sabelfeld, A., Sands, D.: Probabilistic Noninterference for Multi-threaded Programs. In: IEEE Computer Security Foundations Workshop. pp. 200–215 (2000)
23. Smith, G., Volpano, D.: Secure Information Flow in a Multi-threaded Imperative Language. In: ACM Symposium on Principles of Programming Languages. pp. 355–364 (1998)
24. Volpano, D., Smith, G.: A Type-Based Approach to Program Security. In: Bidoit, M., Dauchet, M. (eds.) TAPSOFT 1997. LNCS, vol. 1214, pp. 607–621. Springer (1997)
25. Volpano, D., Smith, G., Irvine, C.: A Sound Type System for Secure Flow Analysis. Journal of Computer Security 4(3), 1–21 (1996)
26. Wasserrab, D., Lochbihler, A.: Formalizing a Framework for Dynamic Slicing of Program Dependence Graphs in Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 294–309. Springer (2008)
27. Wasserrab, D., Lohner, D., Snelting, G.: On PDG-based Noninterference and its Modular Proof. In: ACM Workshop on Programming Languages and Analysis for Security. pp. 31–44 (2009)
28. Zdancewic, S., Myers, A.C.: Observational Determinism for Concurrent Program Security. In: IEEE Computer Security Foundations Workshop. pp. 29–43 (2003)