Using Dynamic Pushdown Networks to Automate a Modular Information-Flow Analysis

Heiko Mantel[1], Markus Müller-Olm[2], Matthias Perner[1(✉)], and Alexander Wenner[2]

[1] Computer Science Department, TU Darmstadt, Darmstadt, Germany
{mantel,perner}@cs.tu-darmstadt.de
[2] Institut Für Informatik, Westfälische Wilhelms-Universität Münster, Münster, Germany
{markus.mueller-olm,alexander.wenner}@wwu.de

Abstract. In this article, we propose a static information-flow analysis for multi-threaded programs with shared memory communication and synchronization via locks. In contrast to many prior analyses, our analysis does not only prevent information leaks due to synchronization, but can also benefit from synchronization for its precision. Our analysis is a novel combination of type systems and a reachability analysis based on dynamic pushdown networks. The security type system supports flow-sensitive tracking of security levels for shared variables in the analysis of one thread by exploiting assumptions about variable accesses by other threads. The reachability analysis based on dynamic pushdown networks verifies that these assumptions are sound using the result of an automatic guarantee inference. The combined analysis is the first automatic static analysis that supports flow-sensitive tracking of security levels while being sound with respect to termination-sensitive noninterference.

Keywords: Information-flow security · Concurrency · Static analysis

1 Introduction

Before giving a multi-threaded program access to sensitive information, one might want to know whether the program keeps this information secret. Static information-flow analyses are a solution for checking whether a program keeps sensitive information secret before running the program.

Information-flow security for sequential programs received a lot of attention in research and mature solutions exist, e.g. [2,5,7,12]. Analyzing information-flow security for concurrent programs is conceptually more difficult. In particular, analyses for sequential programs are not sufficient for analyzing concurrent programs [17], because further information leaks can occur. Consider, for instance, the program $o1{:=}s1; s1{:=}s2; s2{:=}o1; o1{:=}0$, which swaps the values stored in $s1$ and $s2$ via the variable $o1$. Assume the values of $s1$ and $s2$ shall be kept

M. Falaschi (Ed.): LOPSTR 2015, LNCS 9527, pp. 201–217, 2015.
DOI: 10.1007/978-3-319-27436-2_12

Fig. 1. Work flow of the proposed analysis

secret from an attacker who can only observe the variable *o1* after the program run. While the program does not leak the values of *s1* and *s2* if run in isolation, it might leak the value of *s1* to the attacker if the program $o2{:=}o1; o1{:=}o2$ is run concurrently. Synchronization adds further complexity to this problem, because it can introduce additional information leaks [14].

For verifying that multi-threaded programs have secure information flow, several security type systems were proposed and proven sound wrt. noninterference-like security properties (e.g., [16,17]). While some of this work addresses the danger of information leakage via synchronization (e.g., [14,19,20]), the potential positive effects of synchronization primitives for information-flow security have been neglected for some time. However, programmers use synchronization frequently to limit the possible interferences between threads. In particular, synchronization can be employed to prevent information leakage.

Mantel, Sands, and Sudbrock propose a framework for verifying information-flow security in a modular fashion such that the positive effects of synchronization can be exploited [10]. They present a flow-sensitive security type system that is suitable for rely-guarantee-style reasoning about information-flow security based on code annotations that capture a programmer's intentions and expectations by so called modes. A mode is either an assumption about a given thread's environment that the programmer expects to hold when the thread reaches some program point, or it is a guarantee that the programmer intends to provide to the thread's environment. In [10], the security type system is proven sound under the precondition that all assumptions made by a thread are justified by corresponding guarantees of other threads and that all such guarantees are, indeed, provided. In [3], this approach is adapted to a hybrid information-flow analysis, where monitors enforce the soundness of rely-guarantee-style reasoning by forcing threads to provide all guarantees that are needed to justify the assumptions made by other threads.

In this article, we propose a particular combination of security type systems with dynamic pushdown networks [9] (brief: DPNs). The purpose of this combination is to obtain a solution for rely-guarantee-style reasoning where DPNs are used to effectively check that all assumptions are justified. In addition, we present an inference that soundly computes the guarantees that are provided at each program point. That is, our solution statically ensures that modes are used soundly and our soundness result is unconditional, unlike in [10] where a sound use of modes is assumed. In contrast to [3], we present a solution for a static analysis, i.e. one only needs to verify the information flow security of a program once and no run-time overhead is imposed on the program. Another novelty of this article in comparison to [3,10] is that our security type system covers dynamic thread creation as well as lock-based synchronization.

Figure 1 illustrates how the different modules of our analysis interact. The guarantee inference takes a program annotated with assumptions as input and adds guarantee annotations. This program is input to the assumption verifier and the security type system. A program is then accepted as secure if and only if it is accepted by the assumption verifier as well as the security type system.

Overall, our analysis is the first completely automated, static information-flow analysis that soundly enforces termination-sensitive noninterference while permitting flow-sensitive tracking of security levels for shared variables.

2 Basic Notions and Notation

2.1 Model of Computation

We consider multi-threaded programs whose threads synchronize by locks and communicate via shared memory. We focus on interleaving concurrency (i.e., one thread performs a step at a time), non-deterministic scheduling (i.e., each thread could be chosen to perform a step next), and non-re-entrant locks (i.e., a lock can only be acquired if no thread, including the acquiring thread, holds this lock). To capture the behavior of multi-threaded programs, we use two transition systems: a local labeled transition system to capture the behavior of individual threads and a global transition system to capture the behavior of multiple threads.

We assume as given a *finite set of locks Lck* and define the *set of all memory configurations* by $Mem = Var \to Val$, where Var is a *finite set of variables* and Val is a *set of values*. We leave Var and Val both under-specified.

We refer to the states and labels of local, labeled transition systems as local configurations and events, respectively. Formally, a *local transition system* is a triple $(LCnf, Eve, \to)$ where $LCnf$ and Eve are sets and $\to \subseteq LCnf \times Eve \times LCnf$. We define the *set of local configurations* by $LCnf = CCnf \times Mem$, where $CCnf$ is a *set of control configurations* that we leave under-specified for now. An *event* is a term that captures the non-local effects of a thread's computation. We define the set of all events by $Eve = \{\epsilon, \nearrow_{ccnf}, l, \neg l \mid ccnf \in CCnf, l \in Lck\}$. We use the events $\nearrow_{ccnf}$, l, and $\neg l$ to capture the creation of a new thread with initial control configuration $ccnf$, the acquisition of lock l, and the release of l, respectively. The term ϵ signals that no non-local effect occurs. We assume that termination is captured by a predicate trm on control configurations.

A *global transition system* is a pair $(GCnf, \twoheadrightarrow)$, where $GCnf$ is a *set of global configurations* and $\twoheadrightarrow \subseteq GCnf \times GCnf$. We define $GCnf$ by $GCnf = CCnf^+ \times Mem$, i.e., a global configuration is a pair of a non-empty list of local control configurations and a memory configuration. A global configuration $\langle [ccnf_1, \ldots, ccnf_n], mem \rangle$ models a snapshot of a computation with n threads where the ith thread's state is captured by $(ccnf_i, mem)$ for $1 \leq i \leq n$. We say that a list of control configurations $[ccnf_1, \ldots, ccnf_n]$ *has terminated* (denoted $trm([ccnf_1, \ldots, ccnf_n])$) iff $trm(ccnf_i)$ holds for all $i \in \{1, \ldots, n\}$.

We assume the control configuration of a thread to capture which locks are held by this thread. To retrieve the set of acquired locks, we use a function

$locks : CCnf \to 2^{Lck}$ and inductively lift it to a function $locks : CCnf^* \to 2^{Lck}$ by $locks([]) = \emptyset$ and $locks(\overrightarrow{ccnf} +\!+ [ccnf]) = locks(\overrightarrow{ccnf}) \cup locks(ccnf)$. In a global configuration $\langle [ccnf_1, \ldots, ccnf_n], mem\rangle$, $locks(ccnf_i)$ is the set of locks acquired by the ith thread and $Lck \setminus locks([ccnf_1, \ldots, ccnf_n])$ is the set of available locks.

We say that a local transition system $(LCnf, Eve, \to)$ *handles locks properly* iff (1) $(ccnf, mem) \xrightarrow{l} (ccnf', mem')$ implies $locks(ccnf') = locks(ccnf) \dot\cup \{l\}$,[1] (2) $(ccnf, mem) \xrightarrow{\neg l} (ccnf', mem')$ implies $locks(ccnf) = locks(ccnf') \dot\cup \{l\}$, (3) $(ccnf, mem) \xrightarrow{\alpha} (ccnf', mem')$ and $\alpha \notin \{l, \neg l \mid l \in Lck\}$ imply $locks(ccnf') = locks(ccnf)$, and (4) $(ccnf, mem) \xrightarrow{\nearrow_{ccnf^*}} (ccnf', mem')$ implies $locks(ccnf^*) = \emptyset$.

Let $(LCnf, Eve, \to)$ be a local transition system that handles locks properly. The global transition relation $\twoheadrightarrow \subseteq GCnf \times GCnf$ induced by this local transition system is the smallest relation that satisfies the following conditions:

1. If $(ccnf_i, mem) \xrightarrow{l} (ccnf_i', mem')$ and $l \notin locks(\overrightarrow{ccnf}_1 +\!+ \overrightarrow{ccnf}_2)$
   then
   $\langle \overrightarrow{ccnf}_1 +\!+ [ccnf_i] +\!+ \overrightarrow{ccnf}_2, mem\rangle \twoheadrightarrow \langle \overrightarrow{ccnf}_1 +\!+ [ccnf_i'] +\!+ \overrightarrow{ccnf}_2, mem'\rangle$.
2. If $(ccnf_i, mem) \xrightarrow{\nearrow_{ccnf}} (ccnf_i', mem')$
   then
   $\langle \overrightarrow{ccnf}_1 +\!+ [ccnf_i] +\!+ \overrightarrow{ccnf}_2, mem\rangle \twoheadrightarrow \langle \overrightarrow{ccnf}_1 +\!+ [ccnf, ccnf_i'] +\!+ \overrightarrow{ccnf}_2, mem'\rangle$.
3. If $(ccnf_i, mem) \xrightarrow{\alpha} (ccnf_i', mem')$ and $\alpha \notin \{\nearrow_{ccnf}, l \mid ccnf \in CCnf, l \in Lck\}$
   then
   $\langle \overrightarrow{ccnf}_1 +\!+ [ccnf_i] +\!+ \overrightarrow{ccnf}_2, mem\rangle \twoheadrightarrow \langle \overrightarrow{ccnf}_1 +\!+ [ccnf_i'] +\!+ \overrightarrow{ccnf}_2, mem'\rangle$.

The first item above captures the acquisition of a lock by the thread at position $i = 1 + \sharp(\overrightarrow{ccnf}_1)$. Since the local transition system handles locks properly, a lock can only be acquired if no thread – including thread i – holds this lock. The second item captures the creation of a thread by the ith thread. Due to the proper handling of locks, newly created threads hold no locks. Finally, the third item handles all other steps of the ith thread, including the release of a lock.

We inductively define a family of relations $(\twoheadrightarrow_k)_{k\in\mathbb{N}}$ by $gcnf \twoheadrightarrow_0 gcnf$ and if $gcnf \twoheadrightarrow_k gcnf'$ and $gcnf' \twoheadrightarrow gcnf''$ then $gcnf \twoheadrightarrow_{k+1} gcnf''$. The transitive, reflexive closure of $\twoheadrightarrow$ is defined by $gcnf \twoheadrightarrow^* gcnf'$ iff $\exists k \in \mathbb{N}.\, gcnf \twoheadrightarrow_k gcnf'$. If $gcnf \twoheadrightarrow^* gcnf'$ then $gcnf'$ is *reachable* from $gcnf$. We define the *set of all global configurations reachable from gcnf* by $gReach(gcnf) = \{gcnf' \mid gcnf \twoheadrightarrow^* gcnf'\}$.

In Sect. 2.5, we define a local transition system $(LCnf, Eve, \to)$ for a simple programming language and capture multi-threaded computations by the global transition system $(GCnf, \twoheadrightarrow)$, where $\twoheadrightarrow$ is induced by $(LCnf, Eve, \to)$.

2.2 Attacker Model and Definition of Security

We focus on confidentiality in this article. More concretely, we assume that certain variables store secrets, and we only classify a program as secure if it does

[1] We use $\dot\cup$ to denote the disjoint union of two sets, e.g., $locks(ccnf') = locks(ccnf) \dot\cup \{l\}$ is equivalent to $locks(ccnf') = (locks(ccnf) \cup \{l\}) \wedge l \notin locks(ccnf)$.

not reveal information about these secrets when it is run. We consider attackers that might be able to observe the values of all other variables both, before and after a program run. We refer to variables that initially store secrets as **high** and to variables that might be observable to the attacker as **low**.

We define a set of security levels by $Lev = \{\mathbf{low}, \mathbf{high}\}$ and use a function $lev : Var \to Lev$ to associate a security level with each variable. For the attacker, two memory configurations are indistinguishable if they agree on the values of all low variables. We say that $mem, mem' \in Mem$ are **low**-*equal* (denoted by $mem =^{lev}_{\mathbf{low}} mem'$) iff $\forall x \in Var.\, (lev(x) = \mathbf{low} \implies mem(x) = mem'(x))$ holds.

Definition 1. *A control configuration ccnf is* secure *for* $lev : Var \to Lev$ *iff*

$$\begin{aligned}
&\forall mem_1, mem_1', mem_2 \in Mem.\, \forall \overrightarrow{ccnf}_1 \in CCnf^+.\\
&\langle [ccnf], mem_1\rangle \twoheadrightarrow^* \langle \overrightarrow{ccnf}_1, mem_1'\rangle \wedge trm(\overrightarrow{ccnf}_1) \wedge mem_1 =^{lev}_{\mathbf{low}} mem_2\\
&\implies \exists mem_2' \in Mem.\, \exists \overrightarrow{ccnf}_2 \in CCnf^+.\\
&\qquad \langle [ccnf], mem_2\rangle \twoheadrightarrow^* \langle \overrightarrow{ccnf}_2, mem_2'\rangle \wedge trm(\overrightarrow{ccnf}_2) \wedge mem_1' =^{lev}_{\mathbf{low}} mem_2'
\end{aligned}$$

Our security definition captures possibilistic, termination-sensitive noninterference for a two-level security policy [15]. That is, if a program satisfies our security definition then the initial values of **high** variables do not influence the possibility of a **low** attacker's observations. In particular, programs that leak information via their termination behavior [4] do not satisfy Definition 1.

2.3 Dynamic Pushdown Networks

We briefly recall the result on analysis of dynamic pushdown networks (DPNs) from [9] exploited in the assumption verifier and describe the connection to our model of computation. A DPN consists of multiple instances of independent pushdown systems running in parallel. Additional instances can be created dynamically. Synchronisation is supported in the form of locks. Using finite data abstraction, DPNs can thus model concurrent programs with recursive procedures, dynamic thread creation, and synchronization with locks.

Formally, a DPN is a tuple (P, Γ, A, Δ) where P is a finite set of control states, Γ is a finite set of stack symbols, A is a finite set of actions, and $\Delta \subseteq P\Gamma \times A \times P\Gamma^*$ is a finite set of transitions. An action from $\{\nearrow_{p,\gamma} \mid p \in P, \gamma \in \Gamma\} \subseteq A$ indicates creation of a new pushdown instance with a control state p and stack symbol γ, and an action from $\{l, \neg l \mid l \in Lck\} \subseteq A$ indicates acquisition and release of a lock l. The set of acquired locks can be retrieved from a control state with the function $locks : P \to 2^{Lck}$. The set of acquired locks in a control state must be consistent with transitions, i.e. for all $(p\gamma, a, p'w') \in \Delta$ we have $locks(p') = \{l\} \dot\cup locks(p)$ if $a = l$, $locks(p) = \{l\} \dot\cup locks(p')$ if $a = \neg l$ and $locks(p) = locks(p')$ otherwise; in addition $locks(p'') = \emptyset$ if $a = \nearrow_{p'',\gamma''}$. Note that there is no re-entrant use of locks.

Configurations of a DPN are lists of pushdown instances represented as words from $DCnf = (P\Gamma^*)^+$. Let $locks(p_1w_1 \dots p_nw_n) = \bigcup_{i\in\{1,\dots,n\}} locks(p_i)$.

A step of the semantics of the DPN rewrites the control state and topmost stack-symbol of one pushdown instance according to a transition rule, if allowed by the state of locks. On thread creation, a new pushdown instance is added to the left of the current instance in the configuration. Formally, the transition relation $\rightarrow\!\!\triangleright$ is the smallest relation such that $s\, p\gamma w\, s' \rightarrow\!\!\triangleright s\, s''\, p'w'w\, s'$ holds for all $s, s' \in DCnf, w \in \Gamma^*$, $(p\gamma, a, p'w') \in \Delta$ provided $l \notin locks(sp\gamma ws')$ if $a = l$ and $s'' = p''\gamma''$ if $a = \nearrow_{p'',\gamma''}$ and $s'' = \varepsilon$ otherwise.

We say that a thread uses locks in a well-nested fashion if it releases all locks in opposite order of their acquisition. Given a DPN whose threads use locks in a well-nested fashion and a regular set $B \subseteq (P \cup \Gamma)^*$, we can check effectively, whether a configuration in B is reachable from initial configuration s_0 or not, i.e., whether $\exists s \in B : s_0 \rightarrow\!\!\triangleright^* s$ (see [9]).

In order to analyze a program from an initial configuration $\langle [ccnf], mem\rangle$, we consider a DPN $\mathcal{M}_{ccnf} = (P_{ccnf}, \Gamma_{ccnf}, A_{ccnf}, \Delta_{ccnf})$ with $P_{ccnf} \subseteq CCnf$, $ccnf \in P_{ccnf}$ and $\Gamma_{ccnf} = \{\#\}$ that satisfies the following condition: if $ccnf' \in P_{ccnf}$ and $(ccnf', mem) \xrightarrow{\alpha} (ccnf'', mem')$ then $ccnf'' \in P_{ccnf}$, $\alpha' \in A_{ccnf}$ and $(ccnf'\#, \alpha', ccnf''\#) \in \Delta_{ccnf}$, where $\alpha' = \alpha$ for $\alpha \notin \{\nearrow_{ccnf} |\ ccnf \in CCnf\}$, and $ccnf''' \in P_{ccnf}$ and $\alpha' = \nearrow_{ccnf''',\#}$ for $\alpha = \nearrow_{ccnf'''}$. Elements of P_{ccnf} abstract local configurations in the sense that they do not carry information about memory configurations. Correspondingly, the transitions in Δ_{ccnf} abstract steps in the local semantics. However, labelling and hence synchronisation and thread creation is preserved. We reuse the function *locks* defined for control configurations.

The DPN $\mathcal{M}_{ccnf}$ can be used to approximate reachability of configurations starting from $\langle [ccnf], mem\rangle$ respecting synchronisation via locks and thread creation, since $\langle [ccnf], mem\rangle \twoheadrightarrow^* \langle [ccnf_1, \ldots, ccnf_n], mem'\rangle$ implies that $ccnf\# \rightarrow\!\!\triangleright^* ccnf_1\# \ldots ccnf_n\#$. Hence, an unreachable configuration in the DPN translates to an unreachable configuration in the program. Since we abstract from the shared global memory, the converse direction does not hold in general.

The above approach is fitted to non-recursive programs but can easily be extended to recursive programs by using a larger stack alphabet.

2.4 Control Configurations and Modes

We specialize control configurations to triples of the form $(c, lkst, mdst)$, where c is a *command*, *lkst* is a *lock state*, and *mdst* is a *mode state*. In the control configuration of a thread, the command specifies how the thread's computation will continue, the lock state specifies which locks the thread currently holds, and the mode state specifies the thread's current assumptions about its environment as well as the guarantees that the thread currently provides to its environment.

We use *Com*, *LkSt*, and *MdSt* to denote the set of all commands, the set of all lock states, and the set of all mode states, respectively, i.e., $CCnf = Com \times LkSt \times MdSt$. We leave *Com* under-specified and define *LkSt* and *MdSt* below. In Sect. 2.5, we specialize *Com* for the syntax of a concrete programming language and formalize the language's semantics by a local transition system.

Formally, a lock state is a set of locks, i.e., $LkSt = 2^{Lck}$. In a control configuration $(c, lkst, mdst)$ of a thread, the lock state $lkst$ specifies which locks this thread holds. Hence, we define the function $locks$ by $locks((c, lkst, mdst)) = lkst$.

We define mode states to be functions from modes to sets of variables, i.e., $MdSt = Md \to 2^{Var}$, where $Md = \{\mathsf{A\text{-}NR}, \mathsf{A\text{-}NW}, \mathsf{G\text{-}NR}, \mathsf{G\text{-}NW}\}$ is the *set of modes*. The modes A-NR (for *no-read assumption*) and A-NW (for *no-write assumption*) represent assumptions, while the modes G-NR (for *no-read guarantee*) and G-NW (for *no-write guarantee*) represent guarantees. If $x \in mdst(\mathsf{A\text{-}NW})$ then it is assumed that the thread's environment does not write x. Similarly, if $y \in mdst(\mathsf{A\text{-}NR})$ then it is assumed that the thread's environment does not read the variable y. If $x \in mdst(\mathsf{G\text{-}NW})$ and $y \in mdst(\mathsf{G\text{-}NR})$, then the thread guarantees to not write x and to not read y, respectively. We say *a mode state*$mdst$*is consistent with a mode state*$mdst'$ iff $mdst(\mathsf{A\text{-}NW}) \subseteq mdst'(\mathsf{G\text{-}NW})$ and $mdst(\mathsf{A\text{-}NR}) \subseteq mdst'(\mathsf{G\text{-}NR})$, i.e., if all assumptions made by $mdst$ are matched by corresponding guarantees of $mdst'$.

We say that a local configuration $((c, lkst, mdst), mem)$ *provides its no-write guarantees* iff for all $x \in mdst(\mathsf{G\text{-}NW})$ and $(ccnf', mem') \in LCnf$ the implication

$$((c, lkst, mdst), mem) \xrightarrow{\alpha} (ccnf', mem') \implies mem'(x) = mem(x) \tag{1}$$

holds. Moreover, we say $((c, lkst, mdst), mem)$ *provides its no-read guarantees* iff for all $y \in mdst(\mathsf{G\text{-}NR})$, $v \in Val$, and $(ccnf', mem') \in LCnf$ the implication

$$\begin{aligned} &((c, lkst, mdst), mem) \xrightarrow{\alpha} (ccnf', mem') \\ &\implies ((c, lkst, mdst), mem[y \mapsto v]) \xrightarrow{\alpha} (ccnf', mem') \\ &\quad \vee ((c, lkst, mdst), mem[y \mapsto v]) \xrightarrow{\alpha} (ccnf', mem'[y \mapsto v]) \end{aligned} \tag{2}$$

holds. The two disjuncts on the right hand side of the implication cover the case where the variable y is written and not written, respectively, in the step. Finally, we say that a local configuration *provides its guarantees* if it provides both, its no-write guarantees and its no-read guarantees.

We say that a global configuration $\langle [ccnf_1, \ldots, ccnf_n], mem \rangle$ with $ccnf_i = (c_i, lkst_i, mdst_i)$ for each $i \in \{1, \ldots, n\}$ *justifies its assumptions* iff $mdst_j$ is consistent with $mdst_k$ for all $j, k \in \{1, \ldots, n\}$, $j \neq k$. Intuitively, this means that if one thread makes an assumption about a variable then all other threads must provide the corresponding guarantee.

Modes and mode states were introduced in [10] as a basis for rely-guarantee-style reasoning about information-flow security. The approach enables one to verify the security of multi-threaded programs in a modular fashion, based on security guarantees for each individual thread. More concretely, one statically verifies that steps of each thread only cause flows of information that comply with a given security policy. Rely-guarantee-style reasoning frees one from having to reason about arbitrary environments, one only needs to consider environments that satisfy the thread's current assumptions. Such rely-guarantee-style reasoning is sound if at each step of a computation the assumptions of all threads are justified and the guarantees of all threads are provided.

Definition 2. *A global configuration gcnf* ensures a locally sound use of modes *iff for each* $gcnf' \in gReach(gcnf)$*, where* $gcnf' = \langle [ccnf'_1, \ldots, ccnf'_n], mem' \rangle$*, and each* $i \in \{1, \ldots, n\}$*, the local configuration* $(ccnf'_i, mem')$ *provides its guarantees.*

A global configuration gcnf ensures a globally sound use of modes *iff each* $gcnf' \in gReach(gcnf)$ *justifies its assumptions.*

A global configuration gcnf ensures a sound use of modes *iff gcnf ensures both, a locally sound use of modes and a globally sound use of modes.*

Our semantics of modes is similar to the one in [3,10]. One original extension of rely-guarantee-style reasoning about information-flow security in this article is that we cover dynamic thread creation and synchronization with locks, which are two language features not supported by this prior work.

2.5 A Concrete Programming Language with Modes

We define an example programming language with annotations for acquiring and releasing modes. The set of *annotations* is $Ann = \{\mathsf{acq}(md, \overline{x}), \mathsf{rel}(md, \overline{x}) \mid md \in Md \wedge \overline{x} \subseteq Var\}$. An annotation $\mathsf{acq}(md, \overline{x})$ acquires the mode md for all variables in $\overline{x}$, and an annotation $\mathsf{rel}(md, \overline{x})$ releases the mode md for all variables in $\overline{x}$. To capture this formally, we define the function $updMds : MdSt \times Ann \to MdSt$ by $updMds(mdst, \mathsf{acq}(md, \overline{x})) = mdst[md \mapsto mdst(md) \cup \overline{x}]$ and $updMds(mdst, \mathsf{rel}(md, \overline{x})) = mdst[md \mapsto mdst(md) \setminus \overline{x}]$, and lift it to lists of annotations by $updMds(mdst, []) = mdst$ and $updMds(mdst, [a]{+}{+}\overrightarrow{a}) = updMds(updMds(mdst, a), \overrightarrow{a})$.

We define the special mode state $mdst_\perp$ by $mdst_\perp(\mathsf{A\text{-}NR}) = mdst_\perp(\mathsf{A\text{-}NW}) = \emptyset$ and $mdst_\perp(\mathsf{G\text{-}NR}) = mdst_\perp(\mathsf{G\text{-}NW}) = Var$. It is minimal in the sense that it imposes no constraints on assumptions and guarantees of its environment.

We assume as given a set *Exp* of *expressions*, a function $eval : Exp \times Mem \to Val$ that returns the value to which an expression evaluates in a given memory, and a function $vars : Exp \to 2^{Var}$ that returns the set of all variables that appear syntactically in an expression.

The set Com_p of syntactically correct programs is defined by the grammar:

$$\odot := \epsilon \mid @\overrightarrow{a}$$

$$c_p := \mathbf{skip} \mid x{:=}e \mid \mathbf{if}\ e\ \mathbf{then}\ c_p\ \mathbf{else}\ c_p\ \mathbf{fi} \mid \mathbf{while}\ e\ \mathbf{do}\ c_p\ \mathbf{od} \mid c_p; c_p \mid \mathbf{spawn}(c_p) \mid \mathbf{lock}(l)\odot; c_p; \mathbf{unlock}(l) \odot \mid c_p\odot$$

where $\overrightarrow{a} \in Ann^*$, $x \in Var$, $e \in Exp$, and $l \in Lck$. The syntax ensures a well-nested use of locks. The set *Com* of commands is defined by the grammar:

$$c := \mathbf{stop} \mid \mathbf{lock}(l) \odot \mid \mathbf{unlock}(l) \odot \mid c; c \mid c_p$$

We define that $trm((c, lkst, mdst))$ holds iff $c = \mathbf{stop}$. That is, the symbol **stop** indicates that the computation of a thread has terminated.

The local transition system for our programming language is defined by the calculus in Fig. 2. For the rules SK, AS, SQ1, SQ2, IFT, IFF, WHT, and WHF, SP, the lock state as well as the mode state is irrelevant for the premises and both remain

$$\text{SK}\ \frac{}{(\mathbf{skip}, lkst, mdst, mem) \xrightarrow{\epsilon} (\mathbf{stop}, lkst, mdst, mem)}$$

$$\text{AS}\ \frac{eval(e, mem) = v \qquad mem' = mem[x \mapsto v]}{(x{:=}e, lkst, mdst, mem) \xrightarrow{\epsilon} (\mathbf{stop}, lkst, mdst, mem')}$$

$$\text{SQ1}\ \frac{(c_1, lkst, mdst, mem) \xrightarrow{\alpha} (c_1', lkst', mdst', mem') \quad c_1' \neq \mathbf{stop}}{(c_1; c_2, lkst, mdst, mem) \xrightarrow{\alpha} (c_1'; c_2, lkst', mdst', mem')}$$

$$\text{SQ2}\ \frac{(c_1, lkst, mdst, mem) \xrightarrow{\alpha} (\mathbf{stop}, lkst', mdst', mem')}{(c_1; c_2, lkst, mdst, mem) \xrightarrow{\alpha} (c_2, lkst', mdst', mem')}$$

$$\text{SP}\ \frac{}{(\mathbf{spawn}(c), lkst, mdst, mem) \xrightarrow{\nearrow(c, \emptyset, mdst_\perp)} (\mathbf{stop}, lkst, mdst, mem)}$$

$$\text{IFT}\ \frac{eval(e, mem) = \mathbf{true}}{(\mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \mathbf{fi}, lkst, mdst, mem) \xrightarrow{\epsilon} (c, lkst, mdst, mem)}$$

$$\text{IFF}\ \frac{eval(e, mem) = \mathbf{false}}{(\mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \mathbf{fi}, lkst, mdst, mem) \xrightarrow{\epsilon} (c', lkst, mdst, mem)}$$

$$\text{WHT}\ \frac{eval(e, mem) = \mathbf{true}}{(\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}, lkst, mdst, mem) \xrightarrow{\epsilon} (c; \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}, lkst, mdst, mem)}$$

$$\text{WHF}\ \frac{eval(e, mem) = \mathbf{false}}{(\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}, lkst, mdst, mem) \xrightarrow{\epsilon} (\mathbf{stop}, lkst, mdst, mem)}$$

$$\text{LK}\ \frac{lkst \dot\cup \{l\} = lkst'}{(\mathbf{lock}(l), lkst, mdst, mem) \xrightarrow{l} (\mathbf{stop}, lkst', mdst, mem)}$$

$$\text{ULK}\ \frac{lkst = lkst' \dot\cup \{l\}}{(\mathbf{unlock}(l), lkst, mdst, mem) \xrightarrow{\neg l} (\mathbf{stop}, lkst', mdst, mem)}$$

$$\text{AN1}\ \frac{(c, lkst, mdst, mem) \xrightarrow{\alpha} (\mathbf{stop}, lkst', mdst', mem') \quad mdst'' = updMds(mdst', \overrightarrow{a})}{(c@\overrightarrow{a}, lkst, mdst, mem) \xrightarrow{\alpha} (\mathbf{stop}, lkst', mdst'', mem')}$$

$$\text{AN2}\ \frac{(c, lkst, mdst, mem) \xrightarrow{\alpha} (c', lkst', mdst', mem') \qquad c' \neq \mathbf{stop}}{(c@\overrightarrow{a}, lkst, mdst, mem) \xrightarrow{\alpha} (c'@\overrightarrow{a}, lkst', mdst', mem')}$$

Fig. 2. Semantics of the programming language

unchanged. The rules LK and ULK realize acquiring and releasing a lock, respectively. The rule AN1 updates the mode state according to an annotation if the annotated command is reduced to **stop**. The rule AN2 preserves the annotation if the command is not reduced to **stop**.

Given a program c_p, we say that c_p *is secure for lev* iff $(c_p, \emptyset, mdst_\perp)$ is secure for *lev*, that c_p *ensures a locally sound use of modes* iff $\langle[(c_p, \emptyset, mdst_\perp)], mem\rangle$ ensures a locally sound use of modes for all $mem \in Mem$, that c_p *ensures a globally sound use of modes* iff $\langle[(c_p, \emptyset, mdst_\perp)], mem\rangle$ ensures a globally sound

use of modes for all $mem \in Mem$, and that c_p *ensures a sound use of modes* iff $\langle[(c_p, \emptyset, mdst_\perp)], mem\rangle$ ensures a sound use of modes for all $mem \in Mem$.

3 A DPN-based Analysis for Sound Assumptions

We propose a two-step approach for ensuring a globally sound use of modes for a given program c_p. First, we construct a DPN that simulates c_p in the sense of Sect. 2.3. Second, we build an automaton that accepts all DPN configurations that contain a pair of inconsistent mode states. By the connection between DPN and program executions, c_p uses modes globally sound, if no such configuration is reachable in the DPN from a particular initial configuration. The techniques from [9] then enable us to determine whether this is the case.

We construct a DPN $\mathcal{M}_{ccnf}$ for the control configuration $ccnf = (c_p, \emptyset, mdst_\perp)$ as follows: Starting with $ccnf$, we collect all reachable control configurations, actions, and transitions using the rules from Fig. 2, ignoring the memory configurations. The resulting sets P_{ccnf}, A_{ccnf} and Δ_{ccnf} of control states, actions, and transitions satisfy all requirements from Sect. 2.3. Due to the syntax of programs locks are used well-nested in the DPN $\mathcal{M}_{ccnf}$ and mode states are preserved in its configurations.

For the second step, we first introduce a function that checks the mutual consistency of two mode states and returns a summary mode state.

Definition 3. *Let* $MdSt_\top = MdSt \cup \{\top\}$. *The function* $\oplus : MdSt_\top \times MdSt_\top \to MdSt_\top$ *is defined by* $mdst \oplus mdst' = mdst''$ *where*

- $mdst''(md) = mdst(md) \cup mdst'(md)$ *for* $md \in \{\mathsf{A\text{-}NR}, \mathsf{A\text{-}NW}\}$ *and* $mdst''(md) = mdst(md) \cap mdst'(md)$ *for* $md \in \{\mathsf{G\text{-}NR}, \mathsf{G\text{-}NW}\}$ *if* $mdst \neq \top$, $mdst' \neq \top$, $mdst$ *is consistent with* $mdst'$, *and* $mdst'$ *is consistent with* $mdst$.
- $mdst'' = \top$ *otherwise.*

If the two parameter mode states are mutually consistent, the function $\oplus$ returns a regular mode state that imposes the same constraints on concurrent threads as the combination of the original mode states. That is, it makes all assumptions that at least one of the mode states makes and provides only those guarantees that both mode states provide. If one of the parameter mode states makes an assumption that the other mode state does not match with a corresponding guarantee, the function returns the special symbol $\top$.

We are now ready to define the automaton that characterizes DPN configurations containing inconsistent mode states using the function $\oplus$.

Definition 4. *For a DPN* $\mathcal{M}_{ccnf} = (P_{ccnf}, \Gamma_{ccnf}, A_{ccnf}, \Delta_{ccnf})$ *as described above, we define* $\mathcal{A}_{ccnf} = (MdSt_\top, P_{ccnf} \cup \Gamma_{ccnf}, \delta, mdst_\perp, \{\top\})$ *as the conflict automaton, where* $\delta = \{(q, (c, lkst, mdst), q \oplus mdst) \mid q \in MdSt_\top, (c, lkst, mdst) \in P_{ccnf}\} \cup \{(q, \#, q) \mid q \in MdSt_\top\}$. *We denote the language accepted by the automaton by* $\mathcal{L}(\mathcal{A}_{ccnf})$.

$$\text{ISK}\ \frac{\overrightarrow{a} = anno(\overline{x}, \emptyset, \overline{x}_r, \overline{x}_w)}{\overline{x} \vdash \emptyset, \emptyset\{\mathbf{skip}\}\overline{x}_r, \overline{x}_w : \mathbf{skip}@\overrightarrow{a}} \qquad \text{IAS}\ \frac{\overrightarrow{a} = anno(vars(e) \cup \overline{x}, \{x\}, \overline{x}_r, \overline{x}_w)}{\overline{x} \vdash vars(e), \{x\}\{x{:=}e\}\overline{x}_r, \overline{x}_w : x{:=}e@\overrightarrow{a}}$$

$$\text{ILO}\ \frac{\overrightarrow{a} = anno(\overline{x}, \emptyset, \overline{x}_r, \overline{x}_w)}{\overline{x} \vdash \emptyset, \emptyset\{\mathbf{lock}(l)\}\overline{x}_r, \overline{x}_w : \mathbf{lock}(l)@\overrightarrow{a}} \qquad \text{IUL}\ \frac{\overrightarrow{a} = anno(\overline{x}, \emptyset, \overline{x}_r, \overline{x}_w)}{\overline{x} \vdash \emptyset, \emptyset\{\mathbf{unlock}(l)\}\overline{x}_r, \overline{x}_w : \mathbf{unlock}(l)@\overrightarrow{a}}$$

$$\text{IIF}\ \frac{\overline{x} \cup vars(e) \vdash \emptyset, \emptyset\{\mathbf{skip}; c_i\}\overline{x}_r, \overline{x}_w : c_i' \text{ for all } i \in \{1, 2\}}{\overline{x} \vdash vars(e), \emptyset\{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\}\overline{x}_r, \overline{x}_w : \mathbf{if}\ e\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}}$$

$$\text{IWH}\ \frac{\overline{x} \cup vars(e) \vdash \emptyset, \emptyset\{\mathbf{skip}; c\}vars(e), \emptyset : c' \quad \overrightarrow{a} = anno(\overline{x} \cup vars(e), \emptyset, \overline{x}_r, \overline{x}_w)}{\overline{x} \vdash vars(e), \emptyset\{\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}\}\overline{x}_r, \overline{x}_w : \mathbf{while}\ e\ \mathbf{do}\ c'\ \mathbf{od}@\overrightarrow{a}}$$

$$\text{ISQ}\ \frac{\overline{x} \vdash \overline{x}_r', \overline{x}_w'\{c_1\}\overline{x}_r'', \overline{x}_w'' : c_1' \quad \overline{x}' \vdash \overline{x}_r'', \overline{x}_w''\{c_2\}\overline{x}_r, \overline{x}_w : c_2'}{\overline{x} \vdash \overline{x}_r', \overline{x}_w'\{c_1; c_2\}\overline{x}_r, \overline{x}_w : c_1'; c_2'} \qquad \text{IAN}\ \frac{\overrightarrow{a}' = \overrightarrow{a} \restriction_{\mathsf{A}} \quad \overline{x} \vdash \overline{x}_r', \overline{x}_w'\{c\}\overline{x}_r, \overline{x}_w : c'}{\overline{x} \vdash \overline{x}_r', \overline{x}_w'\{c@\overrightarrow{a}\}\overline{x}_r, \overline{x}_w : c'@\overrightarrow{a}'}$$

$$\text{ISP}\ \frac{\emptyset \vdash \emptyset, \emptyset\{\mathbf{skip}; c\}\emptyset, \emptyset : c' \quad \overrightarrow{a} = anno(\overline{x}, \emptyset, \overline{x}_r, \overline{x}_w)}{\overline{x} \vdash \emptyset, \emptyset\{\mathbf{spawn}(c)\}\overline{x}_r, \overline{x}_w : \mathbf{spawn}(c')@\overrightarrow{a}}$$

with $anno(\overline{x}_1, \overline{x}_2, \overline{x}_3, \overline{x}_4) = [\mathsf{acq}(\text{G-NR}, \overline{x}_1), \mathsf{acq}(\text{G-NW}, \overline{x}_2), \mathsf{rel}(\text{G-NR}, \overline{x}_3), \mathsf{rel}(\text{G-NW}, \overline{x}_4)]$

Fig. 3. Inference of guarantee annotations

The states of the automaton record the summary mode state of the partial configuration already read. Thus the initial state is the minimal mode state and transitions accepting a control state add the mode state of the process to the summary using the $\oplus$ operation. Since we are interested in the configurations with inconsistent mode states, $\top$ is the only accepting state.

DPN-reachability and globally sound use of modes are connected as follows:

Theorem 1. *Let $ccnf = (c_p, \emptyset, mdst_\perp)$. If $\mathcal{L}(\mathcal{A}_{ccnf})$ is not reachable from $ccnf\#$ in DPN $\mathcal{M}_{ccnf}$, then c_p ensures a globally sound use of modes.*

4 An Inference for Sound Guarantees

We propose an inference to automatically annotate a command with guarantees. Recall that the initial mode state provides all guarantees, and that mode states are updated based on annotations after the annotated command terminates. With this in mind, the intuition of our inference is that a command requests the release of guarantees that it cannot provide from the preceding command and vouches to re-acquire said guarantees. Hence, the inference propagates sets of variables which may be read or written by a command backwards.

A judgment $\overline{x} \vdash \overline{x}_r', \overline{x}_w'\{c\}\overline{x}_r, \overline{x}_w : c'$ with $\overline{x}, \overline{x}_r, \overline{x}_r', \overline{x}_w, \overline{x}_w' \subseteq \mathit{Var}$ and $c, c' \in \mathit{Com}$ of the inference is derivable with the rules in Fig. 3. The set $\overline{x}$ comprises variables for which a conditional requests that a no-read guarantee shall be re-acquired in the body of the conditional. The sets $\overline{x}_r'$ and $\overline{x}_w'$ comprise variables for which c does not provide a no-read and no-write guarantee, respectively.

The sets $\overline{x}_r$ and $\overline{x}_w$ comprise variables for which a release of the respective guarantees is requested. The resulting command c' is annotated with guarantees.

All rules, except IIF and IAN, annotate a command to re-acquire guarantees that this command cannot provide before releasing requested guarantees. The rule IIF requests that its branches re-acquire and release all guarantees. The rule IAN removes existing guarantee annotations to avoid conflicts with inferred guarantees using a projection to assumption annotations The projection $\restriction_{\mathsf{A}}$ is defined by $[] \restriction_{\mathsf{A}} = []$, $([a]{+}{+}\overrightarrow{a}) \restriction_{\mathsf{A}} = [a]{+}{+}(\overrightarrow{a} \restriction_{\mathsf{A}})$ if $a \in \{\mathsf{acq}(md, \overline{x}), \mathsf{rel}(md, \overline{x}) \mid md \in \{\mathsf{A\text{-}NR}, \mathsf{A\text{-}NW}\} \wedge \overline{x} \subseteq \mathit{Var}\}$ and $([a]{+}{+}\overrightarrow{a}) \restriction_{\mathsf{A}} = \overrightarrow{a} \restriction_{\mathsf{A}}$ otherwise.

Theorem 2. *If* $\emptyset \vdash \emptyset, \emptyset\{\mathbf{skip}; c'_p\}\emptyset, \emptyset : c_p$ *is derivable, then* c_p *ensures a locally sound use of modes.*

Note that some rules add **skip** commands. These additional commands do not influence which final memories are reachable. We do this as a lightweight measure to support pre-annotations without further complicating our formalism.

5 A Type System for Information-Flow Security

We extend the security type system from [10,18]. To this end, we define a total, reflexive order $\sqsubseteq$ on *Lev* such that $\mathbf{low} \sqsubseteq \mathbf{high}$. To support flow-sensitive tracking of security levels for shared variables, we use *partial level assignments*, i.e. partial functions from $\mathit{Var} \rightharpoonup \mathit{Lev}$. For a given level assignment *lev* and a given partial level assignment Λ, a lookup $\Lambda_{lev}\langle x \rangle$ is defined by $\Lambda_{lev}\langle x \rangle = \Lambda(x)$ if $x \in \mathit{pre}(\Lambda)$ and $\Lambda_{lev}\langle x \rangle = \mathit{lev}(x)$ otherwise. Moreover, the partial type environment $\Lambda' = \Lambda \oplus_{lev} a$ is defined by $\Lambda'(x) = \Lambda_{lev}\langle x \rangle$ for all $x \in \mathit{pre}(\Lambda')$ and

$$\mathit{pre}(\Lambda') = \begin{cases} \mathit{pre}(\Lambda) \cup \{x \mid x \in \overline{x} \wedge \mathit{lev}(x) = \mathbf{low}\} \text{ if } a = \mathsf{acq}(\mathsf{A\text{-}NR}, \overline{x}) \\ \mathit{pre}(\Lambda) \cup \{x \mid x \in \overline{x} \wedge \mathit{lev}(x) = \mathbf{high}\} \text{ if } a = \mathsf{acq}(\mathsf{A\text{-}NW}, \overline{x}) \\ \mathit{pre}(\Lambda) \setminus \{x \mid x \in \overline{x} \wedge \mathit{lev}(x) = \mathbf{low}\} \text{ if } a = \mathsf{rel}(\mathsf{A\text{-}NR}, \overline{x}) \\ \mathit{pre}(\Lambda) \setminus \{x \mid x \in \overline{x} \wedge \mathit{lev}(x) = \mathbf{high}\} \text{ if } a = \mathsf{rel}(\mathsf{A\text{-}NW}, \overline{x}) \\ \mathit{pre}(\Lambda) \text{ otherwise.} \end{cases}$$

For **low**-variables, acquiring a no-read assumption enables floating of security levels. This allows tracking when a **low**-variable possibly stores sensitive information. For **high**-variables, acquiring a no-write assumption enables floating of security levels. This allows tracking when a **high**-variable definitely stores public information. Releasing the respective assumptions disables floating of security levels again. We lift the definition of $\oplus_{lev}$ to lists of annotations as follows $\Lambda \oplus_{lev} [] = \Lambda$ and $\Lambda \oplus_{lev} ([a]{+}{+}\overrightarrow{a}) = (\Lambda \oplus_{lev} a) \oplus_{lev} \overrightarrow{a}$.

The type system in Fig. 4 allows to derive judgements of the form $\vdash_{lev} \Lambda\{c\}\Lambda' : c'$. If such a judgment is derivable and *lev* and Λ together approximate where secrets are stored initially, then *lev* and Λ' approximate where secrets are stored after running c, provided concurrent threads behave according to the assumptions. The command c' is a **low**-slice of c, i.e. an abstraction of c in which subcommands that do not contribute to the behaviour observable via **low**-variables

$$\text{TEX}\ \frac{}{\vdash_{lev,\Lambda} e : \bigsqcup_{x \in vars(e)} \Lambda_{lev}\langle x\rangle} \qquad \text{TAH}\ \frac{lev(x) = \mathbf{high} \quad x \notin pre(\Lambda)}{\vdash_{lev} \Lambda\{x{:=}e\}\Lambda : \mathbf{skip}}$$

$$\text{TSK}\ \frac{}{\vdash_{lev} \Lambda\{\mathbf{skip}\}\Lambda : \mathbf{skip}} \qquad \text{TAL}\ \frac{\vdash_{lev,\Lambda} e : \mathbf{low} \quad lev(x) = \mathbf{low} \quad x \notin pre(\Lambda)}{\vdash_{lev} \Lambda\{x{:=}e\}\Lambda : x{:=}e}$$

$$\text{TLO}\ \frac{}{\vdash_{lev} \Lambda\{\mathbf{lock}(l)\}\Lambda : \mathbf{lock}(l)} \qquad \text{TFL}\ \frac{\vdash_{lev,\Lambda} e : \mathbf{low} \quad x \in pre(\Lambda)}{\vdash_{lev} \Lambda\{x{:=}e\}\Lambda[x \mapsto \mathbf{low}] : x{:=}e}$$

$$\text{TUL}\ \frac{}{\vdash_{lev} \Lambda\{\mathbf{unlock}(l)\}\Lambda : \mathbf{unlock}(l)} \qquad \text{TFH}\ \frac{x \in pre(\Lambda)}{\vdash_{lev} \Lambda\{x{:=}e\}\Lambda[x \mapsto \mathbf{high}] : \mathbf{skip}}$$

$$\text{TWL}\ \frac{\Lambda \sqsubseteq \Lambda' \quad \Lambda'' \sqsubseteq \Lambda' \quad \vdash_{lev,\Lambda'} e : \mathbf{low} \quad \vdash_{lev} \Lambda'\{c\}\Lambda'' : c'}{\vdash_{lev} \Lambda\{\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{od}\}\Lambda' : \mathbf{while}\ e\ \mathbf{do}\ c'\ \mathbf{od}}$$

$$\text{TIL}\ \frac{\vdash_{lev,\Lambda} e : \mathbf{low} \quad \vdash_{lev} \Lambda\{c_1\}\Lambda'' : c_1' \quad \vdash_{lev} \Lambda\{c_2\}\Lambda''' : c_2' \quad \Lambda' = \Lambda'' \sqcup \Lambda'''}{\vdash_{lev} \Lambda\{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\}\Lambda' : \mathbf{if}\ e\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}}$$

$$\text{TIH}\ \frac{\vdash_{lev} \Lambda\{c_1\}\Lambda'' : c_1' \quad \vdash_{lev} \Lambda\{c_2\}\Lambda''' : c_2' \quad c_1' = c_2' \quad \Lambda' = \Lambda'' \sqcup \Lambda'''}{\vdash_{lev} \Lambda\{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\}\Lambda' : \mathbf{skip}; c_1'}$$

$$\text{TSQ}\ \frac{\begin{array}{c}\vdash_{lev} \Lambda\{c\}\Lambda'' : c'' \\ \vdash_{lev} \Lambda''\{c'\}\Lambda' : c'''\end{array}}{\vdash_{lev} \Lambda\{c; c'\}\Lambda' : c''; c'''} \qquad \text{TAN}\ \frac{\begin{array}{cc}\vdash_{lev} \Lambda\{c\}\Lambda' : c' & \Lambda'' = (\Lambda' \oplus_{lev} \overrightarrow{a}) \\ \forall x.\Lambda'_{lev}\langle x\rangle \sqsubseteq \Lambda''_{lev}\langle x\rangle & \overrightarrow{a}' = \overrightarrow{a} \restriction_{\text{A-NR,A-NW}}\end{array}}{\vdash_{lev} \Lambda\{c@\overrightarrow{a}\}\Lambda'' : c'@\overrightarrow{a}'}$$

$$\text{TSP}\ \frac{\vdash_{lev} c : c'}{\vdash_{lev} \Lambda\{\mathbf{spawn}(c)\}\Lambda : \mathbf{spawn}(c')} \qquad \text{TTH}\ \frac{\vdash_{lev} \Lambda\{c\}\Lambda : c' \quad pre(\Lambda) = \emptyset}{\vdash_{lev} c : c'}$$

with $\Lambda \sqsubseteq \Lambda'$ iff $pre(\Lambda) = pre(\Lambda')$ and $\Lambda(x) \sqsubseteq \Lambda'(x)$ for all $x \in pre(\Lambda)$

Fig. 4. Security type system

are replaced by **skip**. The rule TTH with judgment $\vdash_{lev} c : c'$ ensures that *lev* alone approximates where secrets are stored. If no such judgment is derivable for a command c, then a secret might influence a **low**-variable in c.

The rule TAN enables and disables flow-sensitivity for particular variables by updating the pre-image of the partial level assignment, and ensures that a secret written into a variable x with $lev(x) = \mathbf{low}$ must be overwritten before disabling flow-sensitivity for x. The rules TFL and TFH track the floating security level of a variable x by updating the level of x in the partial level assignment. The rule TIH permits branching on secrets. To avoid implicit information leaks due to such branchings, TIH requires that the **low**-slices of both branches are syntactically identical. The rules TAH, TFH, and TIH perform the **low**-slicing.

Theorem 3. *If c_p ensures a sound use of modes and $\vdash_{lev} c_p : c'$ is derivable, then c_p is secure for lev.*

Theorems 1, 2, and 3 establish the soundness result for our combined analysis:

Corollary 1. *If* $\emptyset \vdash \emptyset, \emptyset\{\mathbf{skip}; c_p'\}\emptyset, \emptyset : c_p$, *and* $\vdash_{lev} c_p : c'$ *are derivable and* $\mathcal{L}(\mathcal{A}_{ccnf})$ *is not reachable from* $ccnf\#$ *in DPN* $\mathcal{M}_{ccnf}$ *for* $ccnf = (c_p, \emptyset, mdst_\perp)$, *then* c_p *is secure for lev.*

6 Applying the Analysis

We illustrate how our type system gains precision from assumptions, while the DPN-based analysis ensures soundness of the combined analysis with the example program $c_1 = \mathbf{spawn}(o2{:=}o1; o1{:=}o2); o1{:=}s1; s1{:=}s2; s2{:=}o1; o1{:=}0$ and level assignment *lev* with $lev(o1) = lev(o2) = \mathbf{low}$ and $lev(s1) = lev(s2) = \mathbf{high}$. The program c_1 may leak the value of *s1* to an observer of *o1* due to concurrent execution of both threads.

Our security type system indeed rejects c_1, because no typing rule is applicable for $o1{:=}s1$: The rule TAH cannot be applied due to $lev(o1) \neq \mathbf{high}$, the rule TAL cannot be applied due to $lev(s1) \neq \mathbf{low}$, and the rules TFL as well as TFH cannot be applied due to $o1 \notin pre(\Lambda)$ (as the pre-image of the partial level assignment is initially empty and there are no annotations in the program). Using the assumption A-NR to enable flow-sensitivity for variable *o1*, $o1{:=}s1$ can be typed using TFH. To this end the program c_1 can be annotated as follows:

$$\mathbf{spawn}(o2{:=}o1; o1{:=}o2)@[\mathsf{acq}(\text{A-NR}, \{o1\})];$$
$$o1{:=}s1; s1{:=}s2; s2{:=}o1; o1{:=}0@[\mathsf{rel}(\text{A-NR}, \{o1\})]$$

However the program still contains the leak and the analysis detects this. The guarantee inference transforms the command $o2{:=}o1; o1{:=}o2$ of the spawned thread with the rules ISP, ISQ, ISK, and IAS into the following command:

$$\mathbf{skip}@[\mathsf{acq}(\text{G-NR}, \emptyset), \mathsf{acq}(\text{G-NW}, \emptyset), \mathsf{rel}(\text{G-NR}, \{o1\}), \mathsf{rel}(\text{G-NW}, \{o2\})];$$
$$o2{:=}o1@[\mathsf{acq}(\text{G-NR}, \{o1\}), \mathsf{acq}(\text{G-NW}, \{o2\}), \mathsf{rel}(\text{G-NR}, \{o2\}), \mathsf{rel}(\text{G-NW}, \{o1\})];$$
$$o1{:=}o2@[\mathsf{acq}(\text{G-NR}, \{o2\}), \mathsf{acq}(\text{G-NW}, \{o1\}), \mathsf{rel}(\text{G-NR}, \emptyset), \mathsf{rel}(\text{G-NW}, \emptyset)].$$

The annotation $\mathsf{rel}(\text{G-NR}, \{o1\})$ in the first line makes explicit that the thread cannot provide the guarantee to not read *o1* during its next step, i.e. during the step of $o2{:=}o1$ in the second line. By spawning the new thread and executing its annotated first **skip** step, we reach a configuration with two threads. We have $o1 \notin mdst_2(\text{G-NR})$ for the mode state of the spawned thread due to the annotation $\mathsf{rel}(\text{G-NR}, \{o1\})$. Furthermore, we have $o1 \in mdst_1(\text{A-NR})$ for the mode state of the original thread due to the annotation $\mathsf{acq}(\text{A-NR}, \{o1\})$. Hence we have a reachable configuration that does not justify its assumptions. The corresponding DPN configuration preserves the mode states and is thus accepted by our conflict automaton that accepts DPN configurations with inconsistent mode states. Since the DPN over-approximates reachablitiy of the semantics, the reachability analysis from [9] detects that this DPN configuration is reachable, i.e. it detects a possible violation of globally sound use of modes and, hence, the program is rejected.

Adding synchronization via locks to ensure mutual exclusion of the regions accessing variable *o1* finally makes the program secure and no configuration with

inconsistent mode states is reachable in the semantics anymore. Since the DPN models locking precisely, the DPN analysis also no longer detects reachability of any violation of globally sound use of modes. The following version of c_1 with additional synchronization has no leak and is accepted by our analysis:

$$c_2 = \mathbf{spawn}(\mathbf{lock}(l); o2{:=}o1; o1{:=}o2; \mathbf{unlock}(l)); \mathbf{lock}(l)@[\mathsf{acq}(\mathsf{A\text{-}NR}, \{o1\})];$$
$$o1{:=}s1; s1{:=}s2; s2{:=}o1; o1{:=}0; \mathbf{unlock}(l)@[\mathsf{rel}(\mathsf{A\text{-}NR}, \{o1\})]$$

Theorem 4. *Let lev be a domain assignment with* $lev(o1) = lev(o2) = \mathbf{low}$ *and* $lev(s1) = lev(s2) = \mathbf{high}$. *Then there are* c_2', c_2'' *such that* $\emptyset \vdash \emptyset, \emptyset\{\mathbf{skip}; c_2\}\emptyset, \emptyset : c_2'$ *and* $\vdash_{lev} c_2' : c_2''$ *are derivable, and* $\mathcal{L}(\mathcal{A}_{ccnf})$ *is not reachable from ccnf# in DPN* $\mathcal{M}_{ccnf}$ *for* $ccnf = (c_2', \emptyset, mdst_\perp)$. *Hence,* c_2' *is secure for lev.*

7 Related Work

Andrews and Reitman [1] were the first to propose a static information-flow analysis based on flow rules, yet without a soundness proof wrt. a semantic security property. In [17], Smith and Volpano proposed the first security type system with a soundness proof against termination-sensitive noninterference.

The focus for most security type systems with support for synchronization, e.g. [14,19,20], has been preventing information leaks via synchronization. To the best of our knowledge, only the analyses in [10,11,18] can exploit synchronization for their precision. In [11], barrier synchonization allows combining different proof techniques in an analysis. In [10], Mantel, Sands, and Sudbrock introduced the rely-guarantee-style reasoning and the first flow-sensitive security type system for concurrent programs. The relationship of this article to [10] has already been clarified in the introduction.

Beyond security type systems, model-checking, e.g. in [8,13], as well as program dependence graphs, e.g. in [6], have been used to verify information-flow security for concurrent programs. These techniques promise very precise results, but are not necessarily compositional. A compositional analysis reduces the conceptual complexity of the verification, opens up the possibility to re-use analysis results of components, and, thus, can contribute to the scalability of an analysis. Our type system and our guarantee inference are compositional, meaning they can be applied to individual threads. Only our DPN-based analysis, which verifies the assumptions exploited by the type system for the actual program composed of multiple threads, is a whole-program analysis.

8 Conclusion

We automated a modular information-flow analysis for multi-threaded programs with a novel combination of a security type system and a reachability analysis based on DPNs. The combined analysis is sound wrt. termination-sensitive noninterference. The security type system supports flow-sensitive tracking of security levels for shared variables in the analysis of a given thread by exploiting

assumptions about accesses to said variables by other threads. Using a conceptual example, we illustrated how the modules of our analysis interact and how synchronization with locks can contribute to the precision of our analysis.

Lifting the analysis to a realistic language with recursive procedure calls and dynamically allocated data structures is an open task for future work. Finally, we would like to implement our analysis and evaluate it in practice.

Acknowledgments. This work was funded by the DFG under the projects RSCP (MA 3326/4-1/2/3) and IFC4MC (MU 1508/2-1/2/3) in the priority program RS^3 (SPP 1496) and under project OpIAT (MU 1508/1-1/2).

References

1. Andrews, G., Reitman, R.: An axiomatic approach to information flow in programs. ACM Trans. Program. Lang. Syst. **2**(1), 56–76 (1980)
2. Arden, O., Chong, S., Liu, J., Myers, A.C., Nystrom, N., Vikram, K., Zdancewic, S., Zhang, D., Zheng, L.: Jif. Software release: http://www.cs.cornell.edu/jif/ (2014)
3. Askarov, A., Chong, S., Mantel, H.: Hybrid monitors for concurrent noninterference. In: 28th IEEE Computer Security Foundations Symposium, pp. 137–151 (2015)
4. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: 13th European Symposium on Research in Computer Security, pp. 333–348 (2008)
5. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 217–232. Springer, Heidelberg (2013)
6. Giffhorn, D., Snelting, G.: A new algorithm for low-deterministic security. International Journal of Information Security pp. 1–25 (2014)
7. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. **8**(6), 399–422 (2009)
8. Huisman, M., Blondeel, H.-C.: Model-checking secure information flow for multi-threaded programs. In: Mödersheim, S., Palamidessi, C. (eds.) TOSCA 2011. LNCS, vol. 6993, pp. 148–165. Springer, Heidelberg (2012)
9. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 525–539. Springer, Heidelberg (2009)
10. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: 24th IEEE Computer Security Foundations Symposium, pp. 218–232 (2011)
11. Mantel, H., Sudbrock, H., Kraußer, T.: Combining different proof techniques for verifying information flow security. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 94–110. Springer, Heidelberg (2007)
12. Myers, A.C.: JFlow: practical mostly-static information flow control. In: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 228–241 (1999)
13. Ngo, T.M., Stoelinga, M., Huisman, M.: Confidentiality for probabilistic multi-threaded programs and its verification. In: Jürjens, J., Livshits, B., Scandariato, R. (eds.) ESSoS 2013. LNCS, vol. 7781, pp. 107–122. Springer, Heidelberg (2013)

14. Sabelfeld, A.: The impact of synchronisation on secure information flow in concurrent programs. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) PSI 2001. LNCS, vol. 2244, pp. 225–239. Springer, Heidelberg (2001)
15. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
16. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: 13th IEEE Computer Security Foundations Workshop, pp. 200–214 (2000)
17. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 355–364 (1998)
18. Sudbrock, H.: Compositional and Scheduler-Independent Information Flow Security. Ph.D. thesis, Technische Universität Darmstadt, Germany (2013)
19. Terauchi, T.: A type system for observational determinism. In: 21st IEEE Computer Security Foundations Symposium, pp. 287–300 (2008)
20. Vaughan, J., Millstein, T.: Secure information flow for concurrent programs under total store order. In: 25th IEEE Computer Security Foundations Symposium, pp. 19–29 (2012)