Assumptions and Guarantees for Compositional Noninterference

Heiko Mantel[1], David Sands[2], and Henning Sudbrock[1]

[1]Department of Computer Science
TU Darmstadt, Germany
{*mantel,sudbrock*}*@cs.tu-darmstadt.de*

[2]Department of Computer Science and Engineering
Chalmers University of Technology, Sweden
dave@chalmers.se

***Abstract*—The idea of building secure systems by plugging together "secure" components is appealing, but this requires a definition of security which, in addition to taking care of top-level security goals, is strengthened appropriately in order to be compositional. This approach has been previously studied for information-flow security of shared-variable concurrent programs, but the price for compositionality is very high: a thread must be extremely pessimistic about what an environment might do with shared resources. This pessimism leads to many intuitively secure threads being labelled as insecure.**

Since in practice it is only meaningful to compose threads which follow an agreed protocol for data access, we take advantage of this to develop a more liberal compositional security condition. The idea is to give the security definition access to the intended pattern of data usage, as expressed by assumption-guarantee style conditions associated with each thread. We illustrate the improved precision by developing the first flow-sensitive security type system that provably enforces a noninterference-like property for concurrent programs.

***Keywords*-Information Flow Security; Assumption-Guarantee; Compositional Verification; Flow-Sensitivity**

I. Introduction

Before granting a program read access to confidential data, one would like to know that such secrets will not be leaked to untrusted sinks, like, e.g., to users with insufficient clearance or to untrusted servers in a network. Motivated by this desire, research on information flow security has made substantial progress over the last 30 years regarding how information flow security requirements can be characterized formally as well as how they can be reliably certified by verification and program analysis techniques.

Noninterference is certainly the best known property that formally characterizes information flow security [1]. It requires that secret inputs cannot influence a program's output to untrusted sinks and, thereby, ensures that attackers cannot conclude information about secrets from the output that they might receive during the execution of the program. Noninterference could also be used to capture aspects of integrity, but in this article we focus on confidentiality only.

The interplay between concurrency and information flow security is intriguing and has attracted many researchers. Already in the eighties, Sutherland and McCullough proposed noninterference-like properties for abstract specifications of concurrent, distributed systems [2], [3]. Their work initiated a long line of research on noninterference-like properties, culminating in frameworks for comparing and engineering such security properties (see, e.g., [4], [5], [6]). Even earlier, Reitman and Andrews proposed an information flow analysis for concurrent programs [7], but they did not yet provide a soundness proof or precise claim about which security property their analysis enforces. These shortcomings were overcome 17 years later by Volpano and Smith [8].

Many further analyses were proposed since, but the problem of certifying information flow security in a concurrent setting is not yet satisfactorily solved. Some approaches do not support a compositional analysis [7], [9], although compositionality is essential for making the analysis scale. The existing approaches either are overly restrictive or do not have satisfactory semantic foundations (e.g., [7], [10], [11]). There are three main facets of the former deficiency: too conservative assumptions about the environment of the thread under analysis (e.g., [8], [12], [13], [14]), severe restrictions on the communication between threads (e.g., [15], [16], [17]), or assumptions about the run-time environment which do not match existing implementations (e.g., [18], [19]). Moreover, flow-sensitive information flow analyses that are semantically well founded only exist for sequential languages so far. This lack of satisfactory theoretical foundations constitutes a major obstacle for reliably certifying information flow security of concurrent programs in practice.

The overall goal of our research project is to overcome these limitations. In this article, we propose a solution for certifying the information flow security of multi-threaded programs in a compositional and flow-sensitive manner based on assumptions and guarantees. The use of explicit assumptions and guarantees enables us to avoid being overly pessimistic about the environment, introducing inflexible restrictions for communication between threads, or imposing nonstandard requirements on the run-time environment.

In summary, the main novel contributions of the article are

1) a novel information flow security property that is compositional and compatible with assumption-guarantee based reasoning, and
2) the first flow-sensitive security type system that provably enforces a noninterference-like security property for concurrent programs.

Published in Proceedings of the 24th IEEE Computer Security Foundations Symposium, Cernay-la-Ville, France, June 27-29, 2011. IEEE Computer Society, pages 218–232.

While compositional verification based on assumptions and guarantees is popular in other domains (see, e.g., [20], [21], [22], [23], [24]), we are not aware of any assumption-guarantee style reasoning to verify the information flow security of concurrent programs. We illustrate at several small, but realistic programs that our approach is not only a conceptual step forward, but indeed leads to a better precision of security analyses.

We show that our approach is sound given that assumptions are matched by valid guarantees.[1] That a program's pattern of data usage complies with given assumptions and guarantees can be proved using standard techniques. This problem is not the focus of the article, but we sketch how standard solutions can be applied.

We view our contribution as a significant step towards lifting the precision of information flow analyses for concurrent programs to a similar level as modern information flow analyses for sequential programs. We expect this to facilitate making the certification of information flow security for concurrent programs feasible.

II. The Approach at a Glance

Compositional reasoning is essential in the security analysis of complex systems because it reduces conceptual complexity and thereby contributes to making a program analysis scale. Compositional reasoning can also be exploited in software development, for instance, following the appealing idea that secure systems can simply be built by plugging together certified components. In either case, compositional reasoning requires a security condition that, in addition to expressing top-level security requirements, must also be compositional. Unfortunately, making a security condition compositional usually means to considerably strengthen it as otherwise soundness is endangered, leading to a condition that might be much more restrictive than prescribed by the top-level security requirements.

The underlying problem is that, while a non-compositional analysis of a thread can exploit the thread's actual environment (i.e. the other threads of the program), a compositional analysis must presuppose all environments in which the thread could possibly operate. This universal quantification over a large set of environments has led to fully compositional conditions (like, e.g., the strong security condition [12]), but at the cost of being rather restrictive.

We use two minimalistic examples to illustrate the problem of information flow security in multi-threaded programs occurring with traditional fully compositional reasoning.

Example 1. Consider the thread

$c_1 =$ debug:=False;
  if (debug) then log:=log + secret

where the value of the variable secret is secret and the attacker can read the final value of the variable log. When executing the thread c_1 in isolation, the final value of log is independent of the value of secret. In consequence, the thread c_1 has intuitively secure information flow. However, if c_1 runs in parallel with the thread $c_2 =$ debug:=True, the value of secret is appended to the variable log if c_2 is scheduled between the assignment to debug and the if-statement in c_1. Hence, a concurrent program executing the threads c_1 and c_2 might leak the secret to the attacker. In consequence, a security condition that is both adequate and fully compositional must classify at least one of the threads c_1 and c_2 as insecure. For instance, c_1 is classified as insecure by the fully compositional approaches in [25], [12] because c_1 contains an assignment of a secret to a variable whose final value is visible to the attacker, regardless whether the assignment is actually reachable. ◊

Example 2. Consider the thread

$c_3 =$ temp:=key1;
  key1:=key2;
  key2:=temp;
  temp:=0

that swaps the values of the variables key1 and key2, where key1 and key2 contain secret values and the final value of the variable temp is observable by the attacker. Although c_3 assigns a secret to the publicly visible variable temp, c_3 has intuitively secure information flow as it overwrites temp with a constant before it terminates. Nevertheless, executing c_3 in parallel with the thread $c_4 =$ public:=temp (where the final value of public is visible to the attacker) may result in an information leak: If c_4 is executed between the two assignments to the variable temp in c_3, then the final value of the variable public equals the value of the secret key1. Note that also c_4 does not leak any secrets when being executed in isolation. An adequate fully compositional analysis must nevertheless classify at least one of the two considered threads as insecure. Programs that, like c_3, store secrets in a variable whose final value is visible to the attacker are usually classified as insecure (as, e.g., in [25], [19]), even if the variable's final value is not secret. ◊

In both examples, reasoning about information flow security for single threads becomes unsound in a concurrent setting due to concurrent accesses to the shared memory, and, hence, a fully compositional security analysis must reject at least one of the threads in each example. Fortunately, multi-threaded programs usually share memory in a more coordinated fashion. Therefore, full compositionality is rarely needed. For instance, in Example 1 it is unlikely that a programmer would assign to the variable debug after the initial assignment, if his intention is that debug specifies once and for all at the beginning of the program whether debug messages should be output, and, hence, any later modification of debug would be erroneous. In consequence, a security analysis only needs to consider environments that do not assign to debug (in particular, the environment

[1]Proofs of all theorems in the article are available on the authors' website.

consisting of c_2 need not be taken into account). Concerning the thread c_3 in Example 2, it is unlikely that a programmer accesses the temporary variable temp in a concurrent thread while c_3 uses it for swapping the values of key1 and key2, as this typically constitutes a programming error. Therefore, a security analysis need only consider environments that do not access temp during the swapping operation. This excludes, in particular, the environment consisting of c_4.

In this article, we tackle the problem how to exploit the coordinated way in which threads access the shared memory in a compositional security analysis. As a solution, we propose an assumption-guarantee style approach, using assumptions that threads make about how concurrent threads access the shared memory, and guarantees that threads provide about how they access the shared memory.

In a multi-threaded program, the assumptions of a thread express which concurrent memory accesses definitely do not occur at a given point during the thread's execution, if the remaining threads follow the intended way in which the program accesses the shared memory. More concretely, we consider assumptions of the form *"At this point, other threads do not read variable x"* and assumptions of the form *"At this point, other threads do not modify variable x."* This is sufficient to cover typical scenarios in which one thread exclusively accesses variables during the program execution.

We use the information provided by a thread's assumptions about its environment in the security analysis. Consider, for instance, the security analysis of the thread c_1 in Example 1. In this analysis, we can assume that the variable debug remains unchanged between any two execution steps of c_1 within the scope of a no-write assumption for debug. If the no-write assumption holds throughout the execution of c_1, then we can deduce that the insecure assignment log:=log + secret is unreachable even if other threads execute concurrently. Hence, we may classify c_1 as secure without losing compositionality for environments that correctly follow the intended access pattern for debug. Considering the thread c_3 from Example 2, in its security analysis we make the assumption that temp is not read concurrently during the swapping operation. Knowing that the value of temp is not read concurrently, it is safe that c_3 stores a secret in temp during the swapping operation.

The guarantees provided by a thread express that the thread refrains from certain accesses to the shared memory. We consider guarantees that correspond to the above assumptions, i.e., *"At this point, I do not read variable x"* and *"At this point, I do not modify variable x."* Based on such assumptions and guarantees, we develop a security analysis and show that this analysis is compositional – given that each assumption made within a thread is matched by the corresponding guarantee in all threads that may execute in parallel.

Where do the assumptions and guarantees come from? Assumptions and guarantees of the threads could, e.g., be specified using annotations in the program's source code (capturing the intended pattern of data usage), or be derived using a static program analysis (capturing some actual pattern of data usage). For defining security formally, we assume that information about assumptions and guarantees is captured in the program semantics. In Section IV, we provide examples that use annotations in program comments to specify assumptions and guarantees.

The challenge was to develop a formal security condition for concurrent programs that adequately deals with assumptions and guarantees about accesses to the shared memory and thereby enables a compositional security analysis that successfully classifies threads such as c_1 and c_3 in Examples 1 and 2 as secure. We provide a solution to this challenge in the following section.

III. A Novel Security Condition

In this section, we present our novel security condition. It is designed to adequately exploit assumptions and guarantees about accesses to the shared memory, and, as we illustrate in this article, thereby enables a compositional as well as flow-sensitive security analysis.

A. Assumptions and Guarantees

To formalize the assumptions and guarantees informally introduced in Section II, we allow threads to assign *modes* to each variable. Each mode represents an assumption or a guarantee with respect to the corresponding variable. This approach borrows from the idea of access modes in Unix systems, where a file is assigned modes representing how it may be accessed. Similar to files being assigned multiple access modes (e.g., both read and write access), a thread may assign multiple modes to a variable (representing, e.g., that the thread guarantees to neither read nor write the variable).

Formally, we define the set of modes $Mod = \{asm\text{-}noread, asm\text{-}nowrite, guar\text{-}noread, guar\text{-}nowrite\}$. We model a snapshot of the modes of variables during the execution of a thread with a *mode state* $mds \in Mds$. Mode states map each mode $m \in Mod$ to a set $mds(m) \subseteq Var$, where Var is a finite set of variable identifiers which we do not specify further. If $x \in mds(asm\text{-}noread)$ respectively $x \in mds(asm\text{-}nowrite)$, this represents the assumption that no other thread currently reads respectively writes the variable x. Moreover, if $x' \in mds(guar\text{-}noread)$ respectively $x' \in mds(guar\text{-}nowrite)$, this represents the guarantee that the thread itself currently does not read respectively write the variable x'. We assume that no modes are assigned to variables at the beginning of a thread's execution, but that threads acquire respectively release all modes dynamically at runtime. The *initial mode state* mds_0 is defined by $mds_0(m) = \{\}$ for all $m \in Mod$.

B. Execution Model

We consider multi-threaded programs that execute a fixed finite number of threads. A *local configuration* is a triple

$$\langle c, mds, mem\rangle$$

that models a snapshot during the execution of a single thread, where $c \in Com$ is the command that remains to be executed by the thread, $mds \in Mds$ is the mode state modeling the current assumptions and guarantees of the thread, and $mem \in Mem$ is the memory state modeling the current memory by mapping variable identifiers to values from a set Val. The operational small-step semantics for single threads is defined by a transition relation on local configurations, denoted $\rightarrowtriangle$. For being language-independent, we leave the set Com and the transition relation $\rightarrowtriangle$ generic. We only assume a command $\mathsf{stop} \in Com$ which cannot make a transition in any local configuration, representing a terminated thread, and that the mode state does not affect command and memory state in $\rightarrowtriangle$-transitions. The latter assumption captures that modes do not affect the program execution (modes shall solely provide additional information for the security analysis). We instantiate our approach with a concrete programming language in Section IV.

A *global configuration* is a pair

$$\langle\langle(c_1, mds_1), \ldots, (c_n, mds_n)\rangle, mem\rangle$$

that models a snapshot during the execution of a multi-threaded program with n threads. It comprises a list of pairs consisting of a command c_i and a mode state mds_i each modeling a single thread and its assumptions and guarantees, as well as a memory state mem modeling the shared memory. While we are generic in the transition relation for local configurations, we define a specific transition relation $\rightarrow$ for global configurations in Figure 1. This transition relation models the execution of thread pools via arbitrarily interleaved executions of the single threads.

As notational convention we denote local configurations with lc, global configurations with gc, elements of Com with c, of Mds with mds, of Mem with mem, of Var with x, of Val with v, and of Mod with m, all possibly with indices or primes. When $lc = \langle c, mds, mem\rangle$, we write $lc(m)$ for $mds(m)$ and $lc[x \mapsto v]$ for $\langle c, mds, mem[x \mapsto v]\rangle$. Finally, we denote the reflexive transitive closure of $\rightarrow$ with $\rightarrow^*$ and write $gc \rightarrow^1 gc'$ if $gc \rightarrow gc'$ and $gc \rightarrow^{k+1} gc'$ if $gc \rightarrow gc'' \rightarrow^k gc'$ for some gc''.

C. Security Property

We consider a security lattice with two security domains, low and $high$, where the requirement is that no information flows from $high$ to low. This is the simplest policy capturing information flow security, and the results in this article can be lifted to more complex security lattices in a standard way. We assume a *domain assignment* $\mathcal{L} : Var \rightarrow \{low, high\}$ that associates a security domain with each variable identifier. We assume that an attacker can observe the initial and the final values of low variables, but cannot directly access the values of high variables (i.e., access control works correctly). That means, an attacker cannot distinguish between initial respectively between final memory states that differ only in the values of high variables. We capture this upper bound on an attacker's observational capability by the following indistinguishability relation on memory states.

Definition 1. Memory states mem_1 and mem_2 are *low-equal* (denoted by $mem_1 =_{low} mem_2$) if and only if

$$\forall x \in Var : \mathcal{L}(x) = low \implies mem_1(x) = mem_2(x).$$

Many compositional security conditions not only require that two runs of a secure program starting in low-equal memory states result in low-equal final memory states (which captures the idea of noninterference), but require low equality also at intermediate execution points (compare, e.g., [12], [15], [26]). The motivation for this requirement is usually compositionality, as the security analysis of the program's environment might depend on the fact that low variables never store secrets. The requirement, however, may lead to inaccuracies if the attacker cannot access the memory during the program execution. We exploit the coordinated way in which threads access the shared memory to overcome such inaccuracies without losing compositionality. To this end, we allow threads to temporarily store secrets in a low variable as long as concurrent threads do not read that variable, which is expressed by the assumption represented by the mode $asm\text{-}noread$. To capture this formally, we introduce the following relaxed variant of low equality, requiring equality only on low variables without the mode $asm\text{-}noread$.

Definition 2. Memory states mem_1 and mem_2 are *low-equal modulo the mode state* mds (denoted by $mem_1 =^{mds}_{low} mem_2$) if and only if

$$\forall x \in Var : \big(\mathcal{L}(x) = low \ \wedge x \notin mds(asm\text{-}noread)\big) \\ \implies mem_1(x) = mem_2(x).$$

Besides increasing the precision of the security definition by exploiting how threads coordinate their accesses to the shared memory, we also introduce a new approach to formally define the security condition using partial equivalence relations [27]. Following a line of security definitions ([12], [28], [18], [29], [30], [26], etc.), we define a bisimulation-based indistinguishability relation on single commands that is not reflexive, and that only relates commands to themselves that have secure information flow. Like in, e.g., [12], we use strong bisimulations aiming at a scheduler-independent notion of security, but this choice is not required for exploiting assumptions and guarantees and it should be straightforward to adapt the development to timing-insensitive versions such as that from [31]. In contrast to previous definitions, we use a two-step approach

$$\frac{\langle c_i, mds_i, mem\rangle \rightharpoondown \langle c_i', mds_i', mem'\rangle}{\langle\langle(c_1, mds_1), \ldots, (c_i, mds_i), \ldots, (c_n, mds_n)\rangle, mem\rangle \to \langle\langle(c_1, mds_1), \ldots, (c_i', mds_i'), \ldots, (c_n, mds_n)\rangle, mem'\rangle}$$

Figure 1. Transition relation for global configurations

to distinguish explicitly between execution steps of the thread's environment and execution steps of the thread itself: In the first step, we consider memory modifications by concurrently executed threads in a closure condition that exploits assumptions about which variables might actually be modified (Definition 3). In the second step, we consider the execution steps of a single thread by defining a bisimulation relation on local configurations, exploiting no-read assumptions to determine whether secrets may be stored in low variables (Definition 4).

Definition 3. A binary relation $\mathcal{R}$ on local configurations with equal mode states is *closed under globally consistent changes*, if whenever $\langle c_1, mds, mem_1\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2\rangle$ the following conditions are satisfied for all $x \in Var$:

(1) $(x \notin mds(asm\text{-}nowrite) \wedge \mathcal{L}(x) = high) \Rightarrow \forall v_1, v_2 \in Val:$ $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$

(2) $(x \notin mds(asm\text{-}nowrite) \wedge \mathcal{L}(x) = low) \Rightarrow \forall v \in Val:$ $\langle c_1, mds, mem_1[x \mapsto v]\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2[x \mapsto v]\rangle$

The closure conditions in Definition 3 characterize all possible memory modifications that might be performed by an environment that respects a thread's no-write assumptions and that does not store secrets in low variables. Firstly, all variables without a no-write assumption might be modified, while variables with a no-write assumption remain unchanged. Secondly, the new value of low variables must not contain a secret, which is captured by requiring that it is equally modified on both sides of the relation.

Definition 4. A symmetric binary relation $\mathcal{R}$ on local configurations with equal mode states that is closed under globally consistent changes is a *strong low bisimulation modulo modes*, if whenever $\langle c_1, mds, mem_1\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2\rangle$ then

(1) $mem_1 =^{mds}_{low} mem_2$ and

(2) if $\langle c_1, mds, mem_1\rangle \rightharpoondown \langle c_1', mds', mem_1'\rangle$ then there exist a command c_2' and a memory state mem_2' such that $\langle c_2, mds, mem_2\rangle \rightharpoondown \langle c_2', mds', mem_2'\rangle$ and $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}\ \langle c_2', mds', mem_2'\rangle$.

The relation $\approx$ is the union of all strong low bisimulations modulo modes; it is the largest such bisimulation.

Strong low bisimulations modulo modes characterize the following indistinguishability of snapshots during two executions of a thread (represented by local configurations): If local configurations are related by a strong low bisimulation modulo modes, then they are indistinguishable to an observer who sees the values of low variables (except those not read by assumption), and remain indistinguishable after one transition. This is due to Condition (1) in Definition 4, ensuring low equality of memories modulo the mode state, and to Condition (2), ensuring that the configurations after one transition are also in relation, and, hence, have memories low equal modulo their mode state. Repeating this argument shows that the configurations remain indistinguishable after any number of transitions. In consequence, by observing low variables that the environment may read one cannot distinguish the two executions. This remains true if the memory is modified between transitions, as long as variables with no-write assumption remain unchanged and low variables are changed equally in both memories (because strong low bisimulations modulo modes are closed under globally consistent changes).

Based on strong low bisimilarity modulo modes, we define an indistinguishability relation on commands and our novel security condition that classifies a command as secure if it is indistinguishable to itself.

Definition 5. Two commands c_1 and c_2 are *low indistinguishable for the mode state mds* (denoted by $c_1 \sim^{mds}_{low} c_2$) if $\langle c_1, mds, mem_1\rangle \approx \langle c_2, mds, mem_2\rangle$ for all memory states mem_1 and mem_2 with $mem_1 =^{mds}_{low} mem_2$.

A command c is *SIFUM-secure* if $c \sim^{mds_0}_{low} c$ (where mds_0 is the initial mode state defined in Section III-A).

SIFUM-security (expanding to *secure information flow using modes*) adequately captures security in the sense that the final values of low variables do not depend on the initial values of high variables, as long as assumptions and guarantees correctly capture how threads access the shared memory. Further justification is given in Section III-D.

SIFUM-security classifies intuitively secure programs as secure that are not classified as secure by existing compositional analysis techniques for concurrent programs such as [25], [12], [19], [26]. On the one hand, SIFUM-security permits that variables temporarily store values at a security level above their own security domain (exploiting no-read assumptions). This is the basis for flow-sensitive security analyses (compare [32]). On the other hand, no-write assumptions are exploited in two ways in order to arrive at a more precise security analysis. They allow, firstly, to track the current value of variables with no-write assumption, and, secondly, to track whether such variables store values at a security level below their security domain. We provide realistic examples illustrating these benefits in Section IV. Beyond this, SIFUM-security is compositional and hence supports a modular security analysis.

D. Compositionality

In the remainder of this section we show that multi-threaded programs executing SIFUM-secure commands have secure information flow if they use assumptions and guarantees in a sound way. How the latter can be checked is sketched in Section V. Before establishing the compositionality result in Theorem 1, we make the notions of secure information flow for multi-threaded programs and of a sound use of assumptions and guarantees precise.

We define security for multi-threaded programs by a simple noninterference-like security condition.

Definition 6. The multi-threaded program executing the commands $c_1, \ldots, c_n$ is ***SIFUM-secure*** if whenever $mem_1 =_{low} mem_2$ and

$$\begin{aligned}&\langle\langle(c_1, mds_0), \ldots, (c_n, mds_0)\rangle, mem_1\rangle\\ &\qquad\rightarrow^k \langle\langle(c_1', mds_1), \ldots, (c_n', mds_n)\rangle, mem_1'\rangle\end{aligned}$$

for some $k \in \mathbb{N}$, then there exist commands $c_1'', \ldots, c_n''$ and a memory state mem_2' such that

$$\begin{aligned}&\langle\langle(c_1, mds_0), \ldots, (c_n, mds_0)\rangle, mem_2\rangle\\ &\qquad\rightarrow^k \langle\langle(c_1'', mds_1), \ldots, (c_n'', mds_n)\rangle, mem_2'\rangle\end{aligned}$$

and $mem_1'(x) = mem_2'(x)$ for all $x \in Var$ with $\mathcal{L}(x) = low$ and $x \notin mds_i(asm\text{-}noread)$ for all $i \in \{1, \ldots, n\}$.

I.e., after any number of transitions the values of low variables for which no thread makes a no-read assumption are independent from the values of secrets. In particular, the following holds for SIFUM-secure multi-threaded programs:

Proposition 1. Assume that the multi-threaded program executing the commands $c_1, \ldots, c_n$ is SIFUM-secure. Then whenever $mem_1 =_{low} mem_2$ and

$$\begin{aligned}&\langle\langle(c_1, mds_0), \ldots, (c_n, mds_0)\rangle, mem_1\rangle\\ &\qquad\rightarrow^* \langle\langle(\mathsf{stop}, mds_0), \ldots, (\mathsf{stop}, mds_0)\rangle, mem_1'\rangle\end{aligned}$$

there exists mem_2' such that $mem_1' =_{low} mem_2'$ and

$$\begin{aligned}&\langle\langle(c_1, mds_0), \ldots, (c_n, mds_0)\rangle, mem_2\rangle\\ &\qquad\rightarrow^* \langle\langle(\mathsf{stop}, mds_0), \ldots, (\mathsf{stop}, mds_0)\rangle, mem_2'\rangle.\end{aligned}$$

Due to space restrictions, the proof of the above theorem as well as the proofs of all other theorems are omitted in this article, but they are provided on the authors' website.

By Proposition 1, if programs release assumptions and guarantees before termination, SIFUM-security implies that an attacker who can observe the initial and final values of low variables cannot conclude anything about secret values.

Intuitively, a multi-threaded program uses assumptions and guarantees in a sound way if (a) whenever one of its threads makes an assumption then this assumption is matched by the corresponding guarantee in all concurrent threads, and (b) no thread locally violates the guarantees that it provides. To make this connection between program executions and the assumptions and guarantees made by threads precise, we formally characterize requirement (a) in Definition 8 and requirement (b) in Definition 12. These definitions characterize the natural conditions under which we establish the compositionality of SIFUM-security.

Definition 7. A mode state tuple $(mds_1, \ldots, mds_n)$ *has compatible modes* if for all $i \in \{1, \ldots, n\}$ and for all $x \in Var$ the following conditions hold:

(1) $x \in mds_i(asm\text{-}noread) \implies$
$\qquad \forall j \neq i:\ x \in mds_j(guar\text{-}noread)$
(2) $x \in mds_i(asm\text{-}nowrite) \implies$
$\qquad \forall j \neq i:\ x \in mds_j(guar\text{-}nowrite)$

A set of mode state tuples has compatible modes if each of its elements has compatible modes.

Definition 8. The set of *mode state tuples reachable from a global configuration gc* is defined as

$$\begin{aligned}&\{(mds_1, \ldots, mds_n) \mid \exists c_1, \ldots, c_n \in Com, mem \in Mem:\\ &\qquad gc \rightarrow^* \langle\langle(c_1, mds_1), \ldots, (c_n, mds_n)\rangle, mem\rangle\}.\end{aligned}$$

The global configuration gc *ensures a globally sound use of modes* if the set of mode state tuples reachable from gc has compatible modes.

For capturing that a thread never violates the guarantees it provides during its execution, we formally define when a command does not read respectively modify a variable.

Definition 9. A command c *does not read the variable* x if, for all mds, mem, and lc', whenever $lc = \langle c, mds, mem\rangle \rightharpoonup lc'$, then one of the following conditions holds:

(1) $\forall v \in Val: lc[x \mapsto v] \rightharpoonup lc'[x \mapsto v]$
(2) $\forall v \in Val: lc[x \mapsto v] \rightharpoonup lc'$

Note that Condition (1) in Definition 9 covers transitions in which the value of x is not modified, while Condition (2) covers transitions in which the value of x is modified.

Definition 10. A command c *does not modify the variable* x if (for all mds, mem, c', mds', mem') whenever $\langle c, mds, mem\rangle \rightharpoonup \langle c', mds', mem'\rangle$, then $mem(x) = mem'(x)$ holds.

Moreover, we approximate reachability from the perspective of a single thread without knowing what its environment might be. In this approximation, we exploit assumptions about concurrent modifications of the shared memory to reduce the number of possibly reachable local configurations. More concretely, we take into account that variables with no-write assumption are not modified by other threads between execution steps. Note that this approximation is only sound if the thread is executed in an environment that modifies the shared memory only as described by these assumptions.

Definition 11. The set $lReach(lc)$ of *local configurations that are potentially reachable from the local configuration lc* is inductively defined as follows:

(1) $lc \in lReach(lc)$
(2) $\forall lc' \in lReach(lc) : \forall lc'' :$
$lc' \twoheadrightarrow lc'' \implies lc'' \in lReach(lc)$
(3) $\forall \langle c', mds', mem' \rangle \in lReach(lc) : \forall mem'' \in Mem :$
$(\forall x \in mds'(asm\text{-}nowrite) : mem'(x) = mem''(x))$
$\implies \langle c', mds', mem'' \rangle \in lReach(lc)$

Definition 12. A local configuration lc *ensures a locally sound use of modes* if for all $lc' = \langle c', mds', mem' \rangle \in lReach(lc)$ and all $x \in Var$ the following conditions hold:
(1) $x \in lc'(guar\text{-}noread) \implies c'$ does not read x
(2) $x \in lc'(guar\text{-}nowrite) \implies c'$ does not modify x

Definition 13. We say that the global configuration $\langle\langle(c_1, mds_1), \ldots, (c_n, mds_n)\rangle, mem\rangle$ *ensures a sound use of modes* if it ensures a globally sound use of modes and each local configuration $\langle c_i, mds_i, mem\rangle$ ensures a locally sound use of modes (for $i \in \{1, \ldots, n\}$).

The following proposition ensures that our approximation in the definition of local reachability is sound if we consider global configurations that ensure a sound use of modes.

Proposition 2. Assume that the global configuration $\langle\langle(c_1, mds_1), \ldots, (c_n, mds_n)\rangle, mem\rangle$ ensures a sound use of modes and that

$$\langle\langle(c_1, mds_1), \ldots, (c_n, mds_n)\rangle, mem\rangle \rightarrow^* \langle\langle(c'_1, mds'_1), \ldots, (c'_n, mds'_n)\rangle, mem'\rangle.$$

Then $\langle c'_i, mds'_i, mem'\rangle \in lReach(\langle c_i, mds_i, mem\rangle)$ holds for all $i \in \{1, \ldots, n\}$.

Now we establish the compositionality result:

Theorem 1 (Compositionality). Let $c_1, \ldots, c_n$ be SIFUM-secure commands such that $\langle\langle(c_1, mds_0), \ldots, (c_n, mds_0)\rangle, mem\rangle$ ensures a sound use of modes for every memory state mem. Then the multi-threaded program executing the commands $c_1, \ldots, c_n$ is SIFUM-secure.

Proof Sketch: We briefly sketch the proof here, the complete proof is provided on the authors' website. We firstly establish the following central result that connects strong low bisimilarity modulo modes of local configurations to global configurations that ensure a sound use of modes:

Let $\langle\langle(c_{1,1}, mds_1), \ldots, (c_{1,n}, mds_n)\rangle, mem_1\rangle$ *and* $\langle\langle(c_{2,1}, mds_1), \ldots, (c_{2,n}, mds_n)\rangle, mem_2\rangle$ *be global configurations that ensure a sound use of modes. Whenever* $\langle\langle(c_{1,1}, mds_1), \ldots, (c_{1,n}, mds_n)\rangle, mem_1\rangle \rightarrow \langle\langle(c'_{1,1}, mds'_1), \ldots, (c'_{1,n}, mds'_n)\rangle, mem'_1\rangle$ *and there exist* $mem_{1,i}$ *and* $mem_{2,i}$ *for all* $i \in \{1, \ldots, n\}$ *with*
- $\langle c_{1,i}, mds_i, mem_{1,i}\rangle \approx \langle c_{2,i}, mds_i, mem_{2,i}\rangle$ *and*
- $mem_{1,i}(x) = mem_1(x)$ *and* $mem_{2,i}(x) = mem_2(x)$ *whenever* $\left[(\mathcal{L}(x) = high) \vee (mem_1(x) = mem_2(x)) \vee (\forall j \in \{1, \ldots, n\} : x \notin mds_j(asm\text{-}noread))\right]$ *holds,*

then there exist $c'_{2,1} \ldots, c'_{2,n}$ *and* mem'_2 *such that*
(1) $\langle\langle(c_{2,1}, mds_1), \ldots, (c_{2,n}, mds_n)\rangle, mem_2\rangle \rightarrow \langle\langle(c'_{2,1}, mds'_1), \ldots, (c'_{2,n}, mds'_n)\rangle, mem'_2\rangle$ *and*
(2) for all $i \in \{1, \ldots, n\}$ *there are* $mem'_{1,i}$ *and* $mem'_{2,i}$ *with*
- $\langle c'_{1,i}, mds'_i, mem'_{1,i}\rangle \approx \langle c'_{2,i}, mds'_i, mem'_{2,i}\rangle$ *and*
- $mem'_{1,i}(x) = mem'_1(x)$ *and* $mem'_{2,i}(x) = mem'_2(x)$ *whenever* $\left[(\mathcal{L}(x) = high) \vee (mem'_1(x) = mem'_2(x)) \vee (\forall j \in \{1, \ldots, n\} : x \notin mds'_j(asm\text{-}noread))\right]$ *holds.*

For establishing SIFUM-security of the multi-threaded program executing the commands $c_1, \ldots, c_n$, it suffices to inductively apply the above result. Note that for the induction to go through, the result needs to take into account that the corresponding local configurations resulting after k steps in two executions of a multi-threaded program are not necessarily strong low bisimilar modulo modes. This is due to the fact that a concurrent SIFUM-secure thread may store secrets in low variables for which it makes a no-read assumption. Hence, strong low bisimilarity modulo modes is only guaranteed after modifying the values of such variables for $mem'_{1,i}$ and $mem'_{2,i}$. ■

The compositionality result allows to reduce the security analysis to the individual analysis of the single threads for programs in which no thread violates the intended patterns of memory usage captured by the assumptions and guarantees.

IV. Benefits of the Novel Security Condition

To illustrate the benefits of SIFUM-security, we instantiate our approach for a simple imperative programming language and exploit the assumptions that concurrent threads do not read respectively do not write certain variables in the security analysis of three small but realistic example programs.

A. Instantiating the Approach

We introduce a simple imperative programming language for implementing multi-threaded programs. Its syntax supports annotations to specify when threads make assumptions about concurrent memory accesses and when threads provide guarantees about their own memory accesses. Annotations are enclosed in comments (//$\ldots$//) and do not contribute to the runtime behavior of the program. They affect, however, the mode state tracked in the security analysis. The syntax of the programming language is defined by the following grammar (using a set of expressions Exp over variables that we do not specify further):

$$\begin{aligned} ann ::= {} & \mathsf{acq}(m, x) \mid \mathsf{rel}(m, x) \\ c ::= {} & \mathsf{skip} \mid x{:=}e \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \\ & \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} \mid c; c \mid \mathsf{stop} \mid /\!/ann/\!/\ c \end{aligned}$$

where $m \in Mod$, $x \in Var$, and $e \in Exp$. The symbol stop is not intended for use in actual programs, we only use it for defining the language's formal semantics. The annotation $\mathsf{acq}(m, x)$ indicates that the thread acquires the mode m for the variable x, and the annotation $\mathsf{rel}(m, x)$ indicates that the thread releases the mode m for the variable x. Each command may be annotated with multiple such annotations.

The operational semantics is formalized by a calculus for the judgment $\langle c, mds, mem\rangle \twoheadrightarrow \langle c', mds', mem'\rangle$, which is defined in Figure 2. The derivation rules are based on the auxiliary judgment $\langle c, mem\rangle \rightharpoonup \langle c', mem'\rangle$ for commands that are not annotated and that, hence, do not modify the mode state in their first execution step. The calculus for this judgment defines a standard semantics for assignments, conditionals, and while loops. For the semantics of sequential composition, we use evaluation contexts of the form

$$\mathcal{E} \ ::= \ \bullet \mid \mathcal{E}; c.$$

The command $\mathcal{E}[c]$ is defined as the evaluation context $\mathcal{E}$ in which the hole $\bullet$ is replaced with c, i.e., c is the command that shall next be executed when executing $\mathcal{E}[c]$. We assume that expression evaluation is total, atomic, and unambiguous, and express that expression e evaluates to value v in the memory state mem by the judgment $\langle e, mem\rangle \downarrow v$. We denote with $vars(e)$ the set of variables on which the value of e depends, i.e.,

$$\begin{aligned}&[\forall x \in vars(e) : mem_1(x) = mem_2(x)]\\ &\quad\implies [\forall v \in Val : \langle e, mem_1\rangle \downarrow v \iff \langle e, mem_2\rangle \downarrow v]\end{aligned}$$

holds for all $e \in Exp$ and all $mem_1, mem_2 \in Mem$.

The annotations directly preceding a command are evaluated atomically with the first execution step of the command. Hence, evaluating annotations does not introduce additional computation steps. Moreover, the evaluation of annotations only affects the mode state and neither the memory state nor the control flow. The effect of an annotation on the mode state is specified by the function $update$:

Definition 14. We define the mode state $update(mds, ann)$ as $mds[m \mapsto mds(m) \cup \{x\}]$ if $ann = \mathsf{acq}(m, x)$, and as $mds[m \mapsto mds(m) \setminus \{x\}]$ if $ann = \mathsf{rel}(m, x)$.

From the definition of the operational semantics one directly obtains that how a program modifies the memory is not influenced by its annotations:

Proposition 3. Let c_1, c_1', c_2, c_2' be commands, where c_2 and c_2' are obtained from c_1 and c_1' be removing all annotations, and let mds be a mode state. Then $\langle c_2, mem\rangle \rightharpoonup \langle c_2', mem'\rangle$ if and only if there is a mode state mds' such that $\langle c_1, mds, mem\rangle \rightharpoonup \langle c_1', mds', mem'\rangle$.

Moreover, if a command without annotations is strongly secure [12, Definition 6], then the command is also SIFUM-secure. However, SIFUM-security classifies more intuitively secure programs as secure than strong security. This is illustrated by the examples in the following section that exploit how threads coordinate their memory accesses.

B. Exploiting Assumptions for Increased Precision

We illustrate with three small but realistic example programs how the precision of the security analysis increases when using assumptions and guarantees to express how threads coordinate their memory accesses. For showing that the example programs are SIFUM-secure, we exploit the assumptions that they make. For each program, we argue informally why concurrent threads provide the appropriate guarantees if they use the memory in the way intended for the program.

```
c_debug =
  //acq(asm-nowrite, debug)//
  debug:=False;
  if (debug) then  log:=log + secret  else  skip  fi
  //rel(asm-nowrite, debug)//
```

Figure 3. Thread using a debug flag

We firstly provide two examples that each illustrate the benefits of only one of the modes $asm\text{-}nowrite$ and $asm\text{-}noread$. An example of a multi-threaded server subsequently serves to illustrate the benefits of both modes in a more complex scenario. In the example programs, we abbreviate the command "$/\!\!/ann/\!\!/$ skip" with "$/\!\!/ann/\!\!/$".

We take up Example 1 from Section II, to which we add annotations that acquire respectively release the mode $asm\text{-}nowrite$ for the variable debug. The resulting command c_{debug} is displayed in Figure 3. Its annotations capture the intention that debug is written only once at the beginning of the program. Concurrent threads provide appropriate guarantees if they comply with this intention.

Theorem 2. Assume that $\mathcal{L}(\mathsf{log}) = \mathcal{L}(\mathsf{debug}) = low$ and $\mathcal{L}(\mathsf{secret}) = high$. Then c_{debug} is SIFUM-secure.

Due to Theorem 1, the SIFUM-security of c_{debug} assures that c_{debug} has secure information flow in all environments respecting the agreement that debug is not modified by other threads during the execution of c_{debug}. Note that the program obtained from c_{debug} by removing all annotations is not SIFUM-secure. Moreover, this program is classified as insecure by existing compositional analysis techniques such as in [25], [12], [19], [26]. While the compositional analysis techniques in [15], [14] classify programs like c_{debug} as secure, they impose the restriction that the variable log must not be concurrently accessed by the environment, which is restrictive as logging is often performed in multiple threads of a multi-threaded program. Note that in c_{debug} SIFUM-security does not restrict concurrent accesses to the log.

For proving Theorem 2, one can construct an appropriate strong low bisimulation modulo modes iteratively starting from $\langle c_{debug}, mds_0, mem_1\rangle \ \mathcal{R} \ \langle c_{debug}, mds_0, mem_2\rangle$ where $mem_1 =_{low} mem_2$, exploiting that globally consistent changes do not reset the value of debug due to the mode $asm\text{-}nowrite$ and, in consequence, the branch containing the assignment of secret to log need not be explored.

Next, we consider a program with annotations using mode $asm\text{-}noread$. The command c_{temp} in Figure 4 subsequently swaps the values of the two secret variables key1 and key2, and the values of the two public variables pub1 and pub2. The swaps are accomplished using a temporary variable which has mode $asm\text{-}noread$ during the execution of c_{temp}.

$$\frac{}{\langle \mathsf{skip}, mem\rangle \twoheadrightarrow \langle \mathsf{stop}, mem\rangle} \qquad \frac{\langle e, mem\rangle \downarrow v}{\langle x{:=}e, mem\rangle \twoheadrightarrow \langle \mathsf{stop}, mem[x \mapsto v]\rangle} \qquad \frac{}{\langle \mathsf{stop}; c, mem\rangle \twoheadrightarrow \langle c, mem\rangle}$$

$$\frac{\langle e, mem\rangle \downarrow \mathsf{True}}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}, mem\rangle \twoheadrightarrow \langle c_1, mem\rangle} \qquad \frac{\langle e, mem\rangle \downarrow \mathsf{False}}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}, mem\rangle \twoheadrightarrow \langle c_2, mem\rangle}$$

$$\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}, mem\rangle \twoheadrightarrow \langle \mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\ \mathsf{else}\ \mathsf{stop}\ \mathsf{fi}, mem\rangle}$$

$$\frac{\langle c, mem\rangle \twoheadrightarrow \langle c', mem'\rangle}{\langle \mathcal{E}[c], mds, mem\rangle \twoheadrightarrow \langle \mathcal{E}[c'], mds, mem'\rangle} \qquad \frac{\langle c, update(mds, ann), mem\rangle \twoheadrightarrow \langle c', mds', mem'\rangle}{\langle \mathcal{E}[/\!/ann/\!/\ c], mds, mem\rangle \twoheadrightarrow \langle \mathcal{E}[c'], mds', mem'\rangle}$$

Figure 2. Small-step operational semantics

```
$c_{temp}$ =
  //acq($asm\text{-}noread$, temp)//;
  temp:=key1; key1:=key2; key2:=temp;
  temp:=pub1; pub1:=pub2; pub2:=temp
  //rel($asm\text{-}noread$, temp)//
```

Figure 4. Thread reusing temporary variable

This assumption is, for instance, natural in an environment where temp is used as a local variable in the thread c_{temp}.

Theorem 3. Assume that $\mathcal{L}(\mathsf{key1}) = \mathcal{L}(\mathsf{key2}) = \mathit{high}$ and $\mathcal{L}(\mathsf{pub1}) = \mathcal{L}(\mathsf{pub2}) = \mathcal{L}(\mathsf{temp}) = \mathit{low}$. Then c_{temp} is SIFUM-secure.

The example illustrates that SIFUM-security, by allowing low variables to temporarily store secrets, makes flow-sensitive security analyses possible. This is not supported by existing compositional analysis techniques such as in [25], [12], [19], [26] which classify the program as insecure.

One can prove Theorem 3 by iteratively constructing an appropriate strong low bisimulation modulo modes, exploiting that low variables with mode $asm\text{-}noread$ may store secrets, and, hence, the assignment of key1 to temp does not violate the requirements of the bisimulation definition. We provide a flow-sensitive security type system to automate the analysis of such programs in Section V.

Finally, to illustrate how both no-write and no-read assumptions can be exploited in the security analysis in more complex application scenarios we consider a multi-threaded server application. Command c_{srv} in Figure 5 implements a thread that is part of a multi-threaded server. The thread serves a client that requests information from the server. The contents of the variables request and src are set up by the main server thread (which we do not display here) specifying the data that is requested and the network address of the requester, respectively. This technique is used in multi-threaded server implementations like NULL HTTPD [33] that spawn worker threads working on data structures set up by the main server thread. To simplify the setting, we assume identifiers for two categories of data ("secret-data" and "public-data") and two network addresses ("local" and "nonlocal"). The variables secretData and publicData, respectively, contain the data identified by "secret-data" and "public-data". The variables localout and nonlocalout represent output channels to the network addresses "local" and "nonlocal", respectively. The command c_{srv} operates in three steps: It firstly checks whether the request's source is authorized to access the requested data, where the policy is that "secret-data" may only be sent to "local". Afterwards, it computes the answer to the request (which is either the requested data or an error message if the authorization failed). Finally, it writes the answer to the channel identified by src, deletes the answer, and logs the

```
$c_{srv}$ =
  acquire(mutex);
  //acq($asm\text{-}nowrite$, src)////acq($asm\text{-}nowrite$, request)//
  //acq($asm\text{-}nowrite$, auth)////acq($asm\text{-}noread$, answer)//
  if (src $\neq$ "local" $\wedge$ request = "secret-data")
    then auth:=False
     else auth:=True
   fi;
  if (auth = False)
    then answer:="not authorized"
    else if (request = "public-data")
      then answer:=publicData
      else answer:=secretData
    fi
  fi;
  if (src = "local")
    then localout:=answer
    else nonlocalout:=answer
  fi;
  answer:="";
  log:=log + src + request;
  //rel($asm\text{-}nowrite$, src)////rel($asm\text{-}nowrite$, request)//
  //rel($asm\text{-}nowrite$, auth)////rel($asm\text{-}noread$, answer)//
  release(mutex)
```

Figure 5. Worker thread of multi-threaded server

request. We use a mutex variable mutex[2] to ensure exclusive access to variables that are shared with other threads (as for instance request and src which are shared with the main server thread). If threads only access these variables when holding the mutex variable mutex, they provide guarantees matching the assumptions of c_{srv}.

Theorem 4. Assume that $\mathcal{L}(\mathsf{secretData}) = \mathcal{L}(\mathsf{localout}) = \mathit{high}$ and $\mathcal{L}(x) = \mathit{low}$ for all remaining variables. Then c_{srv} is SIFUM-secure.

For establishing the SIFUM-security of c_{srv}, we exploit that the variable answer has the mode $\mathit{asm\text{-}noread}$ and may hence contain information at both the high and the low security level until the mode is released. Moreover, we exploit the assumption that the variables src, request, and auth are not modified by concurrent threads as long as c_{srv} holds the mutex variable mutex. This is essential as otherwise, for instance, the value of request could be modified from "public-data" to "secret-data" after checking the authorization. This modification would result in secretData being sent to nonlocalout although this is not authorized, constituting a type of attack also referred to as a time-of-check-to-time-of-use (TOCTTOU) attack [35].

The above example illustrates how SIFUM-security can be beneficially exploited in the security analysis in realistic multi-threaded example scenarios. We are not aware of a compositional analysis technique for multi-threaded programs that allows a successful information flow security analysis for the multi-threaded server in the above example.

V. A Flow-sensitive Security Type System

Our novel security condition enables us not only to achieve higher precision in compositional reasoning by exploiting assumptions and guarantees, but also allows us to define a security type system for concurrent programs that is flow-sensitive. To our knowledge, this is the first flow-sensitive security type system that establishes information flow security guarantees for concurrent programs.

The typing judgments for commands have the form

$$\vdash \Gamma \; \{c\} \; \Gamma',$$

where $\Gamma, \Gamma' : \mathit{Var} \rightharpoonup \{\mathit{low}, \mathit{high}\}$ are partial functions representing type environments and c is a command. Type environments contain flow-sensitive security types (in the style of [32]) for low variables with mode $\mathit{asm\text{-}noread}$ and for high variables with mode $\mathit{asm\text{-}nowrite}$. The idea is that $\mathit{dom}(\Gamma)$ (respectively $\mathit{dom}(\Gamma')$) contains the low and high variables about which no-read and no-write assumptions are made, respectively, before (respectively after) the execution of c. Hence, $\mathit{dom}(\Gamma)$ may differ from $\mathit{dom}(\Gamma')$. Moreover, if $x \in \mathit{dom}(\Gamma)$, then $\Gamma(x)$ is the security type of x before the execution of c, and if $x \in \mathit{dom}(\Gamma')$, then $\Gamma'(x)$ is the security type of x after the execution of c. If $x \notin \mathit{dom}(\Gamma)$, then the security type of x is determined by its security level $\mathcal{L}(x)$. This is captured by extending a type environment Γ to the corresponding *total lookup function* $\Gamma\langle\cdot\rangle : \mathit{Var} \to \{\mathit{low}, \mathit{high}\}$, which is defined as

$$\Gamma\langle x\rangle = \begin{cases} \Gamma(x) & \text{if } x \in \mathit{dom}(\Gamma), \\ \mathcal{L}(x) & \text{otherwise.} \end{cases}$$

The intuition is that whenever the security type $\Gamma\langle x\rangle$ of a variable x is low, then x cannot contain a secret value.

The derivation rules for the judgment $\vdash \Gamma \; \{c\} \; \Gamma'$ use the auxiliary judgment $\Gamma \vdash e : t$ for typing expressions. The derivation rules for both judgments are displayed in Figure 6.

As usual for a two-level security policy, we assume $\mathit{low} \sqsubseteq \mathit{high}$ and denote the least upper bound operator on security domains with $\sqcup$. For subtyping, we write $\Gamma \sqsubseteq \Gamma'$ if $\mathit{dom}(\Gamma) = \mathit{dom}(\Gamma')$ and $\Gamma(x) \sqsubseteq \Gamma'(x)$ for all $x \in \mathit{dom}(\Gamma)$. Rule [anno] adjusts a type environment based on an annotation. To this end, we write $\Gamma \oplus \mathit{ann}$ for the type environment with $(\Gamma \oplus \mathit{ann})(x) = \Gamma\langle x\rangle$ for all $x \in \mathit{dom}(\Gamma \oplus \mathit{ann})$, where $\mathit{dom}(\Gamma \oplus \mathsf{acq}(m, x)) = \mathit{dom}(\Gamma) \cup \{x\}$ and $\mathit{dom}(\Gamma \oplus \mathsf{rel}(m, x)) = \mathit{dom}(\Gamma) \setminus \{x\}$ if $m = \mathit{asm\text{-}noread}$ and $\mathcal{L}(x) = \mathit{low}$ or if $m = \mathit{asm\text{-}nowrite}$ and $\mathcal{L}(x) = \mathit{high}$, and $\mathit{dom}(\Gamma \oplus \mathit{ann}) = \mathit{dom}(\Gamma)$ otherwise. The third premise in rule [anno] requires that security types of variables do not decrease due to annotations. A decrease of a security type is possible by rule [assign$_2$]. If the security type of a variable x has been reset to its original security level (i.e., it is safe to release the assumption on x), then rule [anno] can remove x from $\mathit{dom}(\Gamma)$. The rules for assignments distinguish between variables without and with floating security type. Rule [assign$_1$] forbids assignments from expressions with type high to a low variable x if $x \notin \mathit{dom}(\Gamma)$. In contrast, if $x \in \mathit{dom}(\Gamma)$, rule [assign$_2$] does not restrict assignments to x. Note that in this case the security type of x is changed to the type of the expression that is assigned to x. Rule [if] covers conditionals with low guards and with high guards. Its third premise prevents implicit information leaks by requiring that the branches under high guards are strong low bisimilar modulo modes. The precondition *mds is consistent with* Γ holds if and only if $\mathit{dom}(\Gamma) = \{x \in \mathit{Var} \mid (\mathcal{L}(x) = \mathit{low} \wedge x \in \mathit{mds}(\mathit{asm\text{-}noread})) \vee (\mathcal{L}(x) = \mathit{high} \wedge x \in \mathit{mds}(\mathit{asm\text{-}nowrite}))\}$. To approximate the third premise syntactically, the approach proposed in [36] can be applied. Note that rule [while] requires low loop guards, but it would be straightforward to support high loop guards by adding a protect-command to the language, ensuring that protected loops are executed atomically (cf. [8], [37]).

Theorem 5. Assume that $\vdash \Gamma \; \{c\} \; \Gamma'$ is derivable, and let mds be a mode state that is consistent with Γ. Then $\langle c, \mathit{mds}, \mathit{mem}_1\rangle \approx \langle c, \mathit{mds}, \mathit{mem}_2\rangle$ holds for all $\mathit{mem}_1, \mathit{mem}_2 \in \mathit{Mem}$ that satisfy $\mathit{mem}_1(x) = \mathit{mem}_2(x)$

[2]Given that our simple language does not support mutexes as primitives, the operations for acquiring and releasing mutexes can be implemented using, e.g., Peterson's algorithm [34] without leaving the language fragment.

$$[\textsf{exp}]\,\frac{}{\Gamma \vdash e : \bigsqcup_{x \in vars(e)} \Gamma\langle x\rangle} \qquad [\textsf{skip}]\,\frac{}{\vdash \Gamma\ \{\textsf{skip}\}\ \Gamma}$$

$$[\textsf{assign}_1]\,\frac{x \notin dom(\Gamma) \quad \Gamma \vdash e : t \quad t \sqsubseteq \mathcal{L}(x)}{\vdash \Gamma\ \{x{:=}e\}\ \Gamma} \qquad [\textsf{assign}_2]\,\frac{x \in dom(\Gamma) \quad \Gamma \vdash e : t}{\vdash \Gamma\ \{x{:=}e\}\ \Gamma[x \mapsto t]}$$

$$[\textsf{if}]\,\frac{\begin{array}{c}\vdash \Gamma\ \{c_1\}\ \Gamma' \qquad\qquad \vdash \Gamma\ \{c_2\}\ \Gamma' \\ \Gamma \vdash e : high \Rightarrow \big[(\forall mds : mds \textit{ is consistent with } \Gamma \Rightarrow c_1 \sim^{mds}_{low} c_2) \wedge (\forall x \in dom(\Gamma') : \Gamma'(x) = high)\big]\end{array}}{\vdash \Gamma\ \{\textsf{if } e \textsf{ then } c_1 \textsf{ else } c_2 \textsf{ fi}\}\ \Gamma'}$$

$$[\textsf{while}]\,\frac{\Gamma \vdash e : low \quad \vdash \Gamma\ \{c\}\ \Gamma}{\vdash \Gamma\ \{\textsf{while } e \textsf{ do } c \textsf{ od}\}\ \Gamma} \qquad [\textsf{anno}]\,\frac{\Gamma' = \Gamma \oplus ann \quad \vdash \Gamma'\ \{c\}\ \Gamma'' \quad \forall x \in Var : \Gamma\langle x\rangle \sqsubseteq \Gamma'\langle x\rangle}{\vdash \Gamma\ \{/\!/ann/\!/\ c\}\ \Gamma''}$$

$$[\textsf{seq}]\,\frac{\vdash \Gamma\ \{c_1\}\ \Gamma' \quad \vdash \Gamma'\ \{c_2\}\ \Gamma''}{\vdash \Gamma\ \{c_1; c_2\}\ \Gamma''} \qquad [\textsf{sub}]\,\frac{\vdash \Gamma_1\ \{c\}\ \Gamma_1' \quad \Gamma_2 \sqsubseteq \Gamma_1 \quad \Gamma_1' \sqsubseteq \Gamma_2'}{\vdash \Gamma_2\ \{c\}\ \Gamma_2'}$$

Figure 6. Flow-sensitive security type system

for all $x \in Var$ with $\Gamma\langle x\rangle = low$.

Typing judgments for concurrent programs have the form $\vdash\ c_1, \ldots, c_n$. The single derivation rule is as follows, where Γ_0 is the type environment with $dom(\Gamma_0) = \{\}$:

$$[\textsf{par}]\,\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\} :\ \vdash \Gamma_0\ \{c_i\}\ \Gamma_0 \\ \forall mem{:}\langle (c_1, mds_0), \ldots, (c_n, mds_0), mem\rangle \text{ ensures} \\ \text{a sound use of modes}\end{array}}{\vdash c_1, \ldots, c_n}$$

Theorem 6. Let $c_1, \ldots, c_n$ be commands such that the judgment $\vdash c_1, \ldots, c_n$ is derivable. Then the program consisting of the commands $c_1, \ldots, c_n$ is SIFUM-secure.

The command c_{temp} from Section IV is typable and, hence, SIFUM-secure: As long as the low variable temp has mode *asm-noread*, its type is flow-sensitive and may vary during the execution. The typing rules set its type to *high* when assigning key1 to temp (compare rule [assign$_2$]), and resets it to *low* when assigning pub1 to temp.

Our type system focuses on the novel aspect of flow-sensitive security types for concurrent programs. Note that the definition of SIFUM-security facilitates further improvements for increased precision without losing soundness, which we omit due to space restrictions. It is possible to soundly integrate abstractions of the current values of variables with mode *asm-nowrite*. For instance, considering command c_{debug} from Section IV, an analysis could derive that "debug = True" from the initial assignment to debug, and later exploit this to determine the value of the guard expression "debug". Note that in the above type system, floating types of high variables can in fact be seen as a two-valued abstraction of their current value, where the abstract value is *low* only if the variable's value is public.

Local and Global Soundness. So far we have not considered local and global soundness of the modes. We provide a straightforward type system for establishing locally sound use of modes in an addendum to this article that is available on the authors' website. It is essentially just a Hoare logic specialised to reason about guarantees in the mode state: triples of the form $mds\{c\}mds'$ determine that if c is executed in mode state mds then mds' describes an upper bound on the guarantees after c has executed.

Establishing global soundness is not the focus of this work, but here we sketch one simple approach. By analogy with the type system for establishing locally sound use of modes, we can effectively provide a safe approximation of the modes at each program point, where "safe" means an upper bound on assumptions and a lower bound on guarantees. We can use this to establish global soundness by using a *may-happen-in-parallel* analysis (see, e.g., [38], [39], [40]) to determine all pairs of program points which may execute concurrently. To verify global soundness we simply check that for all such pairs of program points, the modes given by the type system for those program points are compatible. The aforementioned may-happen-in-parallel analyses have been automated, but they are not compositional. Compositional approaches to checking data-sharing annotations exist, but are yet somewhat restricted (see Section VII).

VI. Treatment of Intermediate Outputs

While we assume an attacker who observes only the final values of low variables when arguing for the adequacy of SIFUM-security (compare Proposition 1), we illustrate here that SIFUM-security also adequately captures security when extending the execution model with intermediate outputs that are visible to the attacker. We model an output channel by a distinguished low variable out $\in Var$ storing a String value to which programs only append information. I.e., whenever $\langle c, mds, mem\rangle \rightarrowtriangle \langle c', mds', mem'\rangle$ then there is a (possibly empty) String s with $mem'(\textsf{out}) = mem(\textsf{out}){:}s$, where the colon denotes string concatenation. The intuition is that outputs are appended to the variable out, i.e., out stores the sequence of all outputs. We assume that commands do not acquire the mode *asm-noread* for out, capturing that outputs might always be read (e.g., by the attacker). Then we obtain

Theorem 7. Let $p = \langle (c_1, mds_0), \dots, (c_n, mds_0) \rangle$ be a list of pairs of commands and mode states such that $c_1, \dots, c_n$ are SIFUM-secure and $\langle p, mem \rangle$ ensures a sound use of modes for all mem. Then for all k, all p', all $mem_1 =_{low} mem_2$, and all mem'_1 with $\langle p, mem_1 \rangle \rightarrow^k \langle p', mem'_1 \rangle$ there exist p'' and mem'_2 with $\langle p, mem_2 \rangle \rightarrow^k \langle p'', mem'_2 \rangle$ and $mem'_1(\mathsf{out}) = mem'_2(\mathsf{out})$.

I.e., an attacker cannot deduce secret information from observing outputs, as the sequence of outputs after any number of execution steps does not depend on secrets. (The theorem follows directly from Theorem 1 and Definition 6.)

VII. Related Work

Here, we focus on approaches that consider rely-guarantee techniques in a security context, on information flow security for concurrent programs, and on approaches to reasoning about concurrent programs using data-sharing annotations.

Rely-Guarantee Reasoning for Security: There are a number of works which use the general idea of rely-guarantee reasoning in order to perform modular security analyses. Jürjens [41] studied a probabilistic noninterference for trace-based message-passing processes, and used rely-guarantee reasoning in the proof of compositionality. Guttman *et al* [42] annotate Strand-space representations of protocol participants with rely-guarantee statements related to trust. In common with the approach here a global soundness is required for annotated protocols. Garg *et al* [43] place heavy emphasis on rely-guarantee proof rules in reasoning about safety-property-based security properties in a concurrent first-order functional language.

Information Flow Security for Concurrent Programs: To our knowledge, the first approach to information flow analysis for concurrent programs is proposed by Andrews and Reitman [7], [44]. It considers a flow logic that supports the derivation of noninterference proofs for programs. They sketch an extension to concurrent programs based on the Owicki-Gries method [45]. The idea is that one must show that any derivation for a given thread is unaffected by the assignment statements in any other. As well as being noncompositional, a consequence of this approach is an assumption that any assignment in one thread may run concurrently with any other statement. This would make the opportunities to exploit flow sensitivity limited to thread-local variables. Moreover, the approach neither provides a semantically justified soundness result nor a formal semantics for the underlying programming language.

At the end of the 90s, Volpano and Smith developed security type systems for concurrent programs together with a semantical soundness proof [8], [25]. Several more semantically sound analysis techniques for concurrent programs were subsequently proposed ([12], [13], [18], [36], [19], [29], [30], etc.), providing, e.g., scheduler-independence results and support for controlled declassification. However, we are not aware of any prior approaches with a sound semantic foundation that provide a flow-sensitive security analysis for concurrent programs, which is possible with our assumption-guarantee based approach.

A desirable property of security analyses for concurrent programs that has been widely investigated is scheduler independence [12], [18], [15], [19], [26], i.e., the adequacy of the security analysis for multiple runtime environments differing in the scheduler. The first such scheduler-independent security condition was the strong security condition proposed by Sabelfeld and Sands [12]. The scheduler independence of strong security is due to its bisimulation-based security definition that requires lock step execution of threads. As SIFUM-security is also based on a bisimulation requiring lock step execution, we are very confident that the argument for scheduler independence is also applicable for SIFUM-security. Quite recently, Mantel and Sudbrock developed a more flexible scheduler-independent security property, supporting loops whose guards depend on secrets [26]. We believe that it is also possible to integrate their improved approach to scheduler independence with SIFUM-security.

There are some approaches to information flow security that consider synchronization [44], [28], [16], [46], [47]. These approaches focus on the prevention of information leaks that arise if the occurrence of synchronization depends on secrets. In contrast, our approach allows to exploit the effects of thread synchronization, as different types of synchronization primitives (for instance, mutexes, barriers, and critical regions) can be used to ensure that the assumptions made by threads about concurrent variable accesses are justified. The only other approach we are aware of that exploits synchronization primitives is followed in [14], where barrier synchronization is added to programs for making a successful security analysis possible. Russo and Sabelfeld [19], [47] follow an approach that assumes a nonstandard interface to the scheduler through which threads convey security-relevant information. In contrast, we support standard synchronization primitives to let threads communicate with other threads directly.

Data-Sharing Annotations: The data-access assumptions upon which our approach is based are not security specific, but are nevertheless somewhat different from the classic properties. As our focus is compositional information flow security, we do not, in this paper, consider in depth the problem of verifying the global soundness of modes. Hence, it is useful to consider related work on correctness of concurrent programs which make use of similar assumptions, and in addition provide some form of verification method.

One influential line of work stems from Boyland's *frac-*

tional permissions [48] that permit to divide permissions among multiple threads: Full permissions permit both writing and reading, partial permissions permit only reading. I.e., full permissions correspond to the combination of our no-read and no-write assumption, and partial permissions to the combination of our no-write assumption and no-write guarantee. Other kinds of fractional permissions have also been investigated: In [49], fractional *deny*-permissions, roughly, correspond to the combination of our no-write assumption and guarantee, and fractional *guar*-permissions to neither making no-write assumptions nor providing no-write guarantees. Based on ideas from concurrent separation logic [50], compositional logics for reasoning with fractional permissions have been developed, where a key to compositionality is to exploit specific synchronization primitives that permit to safely transfer permissions between threads. For instance, [49] uses synchronization points provided by fork/join synchronization to transfer permissions between dynamically forked threads, and [51] allows to transfer permissions at synchronization points established by barrier synchronization. Other approaches exploit atomic statements [52] and conditional critical regions [50]. Our approach is, in contrast, not specific to some particular synchronization primitive – the price we pay for this generality is that we cannot describe the soundness of modes in a compositional way. An approach to transferring permissions that is similar to our approach for transferring modes using the annotations acq and rel are the *inhale* and *exhale* commands in Chalice [53].

The closest form of assumption to those used in the present paper are the thread-level *exclusive ownership* (nobody else will read or write) and *read ownership* policies used in recent work of Martin *et al* [54]. Ownership is acquired and released via annotations. The soundness of the annotations are not verified however, but are checked using a runtime monitor. This is also related to the *sharing cast* annotations checked by a combination of static and dynamic analysis in the SharC tool [55].

VIII. Conclusion

Assuring secure information flow for concurrent programs has been a long standing problem in security research. Existing compositional solutions reject many intuitively secure programs. This limits the applicability of information flow security analyses for concurrent programs. Aiming at overcoming the limitations of existing analysis techniques, we successfully exploited assumptions and guarantees in the information flow analysis of concurrent programs. We developed a novel compositional information flow security property that is compatible with assumptions and guarantees. Based on the novel security property, we were able to successfully analyze realistic concurrent programs that are intuitively secure but could not be successfully analyzed with prior approaches. Moreover, based on our approach we provide the first compositional security analysis for concurrent programs that is flow-sensitive.

The approach presented in this article is complementary to the approach pursued in [26]. In [26], the stepwise bisimulation in the security definition could be relaxed while remaining both compositional and scheduler-independent. This enabled a compositional security analysis which is more flexible than prior scheduler-independent analyses. In contrast, here our goal is to improve compositional reasoning by introducing assumptions and guarantees to exploit how concurrently executing threads coordinate their accesses to the shared memory, leading for instance to the possibility of a flow-sensitive security type system for concurrent programs. We expect the two approaches to be compatible. A next step in our research project will be to combine these complementary approaches in a uniform framework.

While our objective was to increase the precision of information flow analysis for concurrent programs, we believe that assumption-guarantee style reasoning can also improve the treatment of other aspects, like, e.g., method calls (in a concurrent as well as in a sequential setting).

We hope that our contributions will initiate the adoption of assumption-guarantee based reasoning in information flow security and enable more widely and better applicable information flow security analyses for concurrent programs.

Acknowledgment

The authors thank Greg Morrisett and the anonymous reviewers for helpful comments. The first author thanks the ProSec Group at Chalmers University for the inspiring environment during his sabbatical. This work was partially funded by the DFG (German research foundation) under the projects FM-SecEng in the Computer Science Action Program (MA 3326/1-2) and RSCP in the priority programme RS^3 (MA 3326/4-1), by the European Commission under the WebSand project, and by the Swedish research agencies SSF and VR. This article reflects only the authors' views, and the DFG, the European Commission, the SSF, the VR, and the authors are not liable for any use that may be made of the information contained therein.

References

[1] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *Proceedings of the 3rd IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982, pp. 11–20.

[2] D. Sutherland, "A Model of Information," in *Proceedings of the 9th National Computer Security Conference*, 1986.

[3] D. McCullough, "Specifications for Multi-Level Security and a Hook-Up Property," in *Proceedings of the 8th IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1987, pp. 161–166.

[4] J. D. McLean, “A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions,” in *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1994, pp. 79–93.

[5] H. Mantel, “On the Composition of Secure Systems,” in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002, pp. 88–104.

[6] H. Mantel, “The Framework of Selective Interleaving Functions and the Modular Assembly Kit,” in *Proceedings of the 3rd ACM Workshop on Formal Methods for Security Engineering: From Specifications to Code (FMSE)*. ACM, 2005, pp. 53–62.

[7] R. P. Reitman and G. R. Andrews, “Certifying Information Flow Properties of Programs: An Axiomatic Approach,” in *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1979, pp. 283–290.

[8] D. Volpano and G. Smith, “Probabilistic Noninterference in a Concurrent Language,” in *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 1998, pp. 34–43.

[9] C. Hammer and G. Snelting, “Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control based on Program Dependence Graphs,” *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, 2009.

[10] J. P. Banâtre and C. Bryce, “Information Flow Control in a Parallel Language Framework,” in *Proceedings of the 6th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 1993, pp. 39–52.

[11] T. C. Tsai, A. Russo, and J. Hughes, “A Library for Secure Multi-threaded Information Flow in Haskell,” in *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2007, pp. 187–202.

[12] A. Sabelfeld and D. Sands, “Probabilistic Noninterference for Multi-threaded Programs,” in *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2000, pp. 200–215.

[13] G. Smith, “A New Type System for Secure Information Flow,” in *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2001, pp. 115–125.

[14] H. Mantel, H. Sudbrock, and T. Kraußer, “Combining Different Proof Techniques for Verifying Information Flow Security,” in *Proceedings the 16th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR)*, Springer LNCS 4407. Springer, 2007, pp. 94–110.

[15] S. Zdancewic and A. C. Myers, “Observational Determinism for Concurrent Program Security,” in *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2003, pp. 29–43.

[16] M. Huisman, P. Worah, and K. Sunesen, “A Temporal Logic Characterisation of Observational Determinism,” in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2006, pp. 3–15.

[17] T. Terauchi, “A Type System for Observational Determinism,” in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2008, pp. 287–300.

[18] G. Boudol and I. Castellani, “Noninterference for Concurrent Programs and Thread Systems,” *Theoretical Computer Science*, vol. 281, no. 1-2, pp. 109–130, 2002.

[19] A. Russo and A. Sabelfeld, “Securing Interaction between Threads and the Scheduler,” in *19th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2006, pp. 177–189.

[20] E. W. Stark, “A Proof Technique for Rely/Guarantee Properties,” in *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Springer LNCS 206. Springer, 1985, pp. 369–391.

[21] M. Y. Vardi, “On the Complexity of Modular Model Checking,” in *Proceedings of the 10th IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1995, pp. 101–111.

[22] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, “You Assume, We Guarantee: Methodology and Case Studies,” in *Proceedings of the 10th International Conference on Computer Aided Verification (CAV)*, Springer LNCS 1427. Springer, 1998, pp. 440–451.

[23] C. S. Pasareanu, M. B. Dwyer, and M. Huth, “Assume-Guarantee Model Checking of Software: A Comparative Case Study,” in *Proceedings of the 5th and 6th Workshop on Theoretical and Practical Aspects of SPIN Model Checking*, Springer LNCS 1680. Springer, 1999, pp. 168–183.

[24] J. Dingel, “Computer-Assisted Assume/Guarantee Reasoning with VeriSoft,” in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2003, pp. 138–148.

[25] D. Volpano and G. Smith, “Probabilistic Noninterference in a Concurrent Language,” *Journal of Computer Security*, vol. 7, no. 2,3, pp. 231–253, 1999.

[26] H. Mantel and H. Sudbrock, “Flexible Scheduler-Independent Security,” in *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, Springer LNCS 6345. Springer, 2010, pp. 116–133.

[27] A. Sabelfeld and D. Sands, “A Per Model of Secure Information Flow in Sequential Programs,” in *Proceedings of the 8th European Symposium on Programming (ESOP)*, Springer LNCS 1576. Springer, 1999, pp. 50–59.

[28] A. Sabelfeld, “The Impact of Synchronisation on Secure Information Flow in Concurrent Programs,” in *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics (PSI)*, Springer LNCS 2244. Springer, 2001, pp. 225–239.

[29] H. Mantel and A. Reinhard, “Controlling the What and Where of Declassification in Language-Based Security,” in *Proceedings of the 16th European Symposium on Programming (ESOP)*, Springer LNCS 4421. Springer, 2007, pp. 141–156.

[30] A. Lux and H. Mantel, “Declassification with Explicit Reference Points,” in *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, Springer LNCS 5789. Springer, 2009, pp. 69–85.

[31] A. Almeida Matos and G. Boudol, “On Declassification and the Non-Disclosure Policy,” in *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2005, pp. 226–240.

[32] S. Hunt and D. Sands, “On Flow-Sensitive Security Types,” in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2006, pp. 79–90.

[33] NullLogic, “Null httpd Web Server,” http://nullhttpd.sourceforge.net/httpd/, accessed 2011 February 8.

[34] G. L. Peterson, “Myths About the Mutual Exclusion Problem,” *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, 1981.

[35] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, “A Taxonomy of Computer Program Security Flaws,” *ACM Computing Surveys*, vol. 26, no. 3, pp. 211–254, 1994.

[36] H. Mantel and D. Sands, “Controlled Declassification based on Intransitive Noninterference,” in *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS)*, Springer LNCS 3302. Springer, 2004, pp. 129–145.

[37] A. Russo and A. Sabelfeld, “Security for Multithreaded Programs Under Cooperative Scheduling,” in *Proceedings of the Andrei Ershov 6th Conference on Perspectives of System Informatics (PSI)*, Springer LNCS 4378. Springer, 2006, pp. 474–480.

[38] S. P. Masticola and B. G. Ryder, “Non-concurrency Analysis,” in *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 1993, pp. 129–138.

[39] G. Naumovich and G. S. Avrunin, “A Conservative Data Flow Algorithm for Detecting All Pairs of Statements That May Happen in Parallel,” in *Proceedings of the the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 1998, pp. 24–34.

[40] G. Naumovich, G. S. Avrunin, and L. A. Clarke, “An Efficient Algorithm for Computing *MHP* Information for Concurrent Java Programs,” in *Proceedings of the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 1999, pp. 338–354.

[41] J. Jürjens, “Secure Information Flow for Concurrent Processes,” in *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, Springer LNCS 1877. Springer, 2000, pp. 395–409.

[42] J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen, “Trust Management in Strand Spaces: A Rely-Guarantee Method,” in *Proceedings of the 13th European Symposium on Programming (ESOP)*, Springer LNCS 2986. Springer, 2004, pp. 325–339.

[43] D. Garg, J. Franklin, D. Kaynar, and A. Datta, “Compositional System Security with Interface-Confined Adversaries,” in *Proceedings of the 26th Conference on Mathematical Foundations of Programming Semantics (MFPS)*, ENTCS. Elsevier, 2010, pp. 49–71.

[44] G. R. Andrews and R. P. Reitman, “An Axiomatic Approach to Information Flow in Programs,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 56–76, 1980.

[45] S. S. Owicki and D. Gries, “An Axiomatic Proof Technique for Parallel Programs I,” *Acta Informatica*, vol. 6, pp. 319–340, 1976.

[46] G. Le Guernic, “Automaton-based Confidentiality Monitoring of Concurrent Programs,” in *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2007, pp. 218–232.

[47] A. Russo and A. Sabelfeld, “Securing Interaction between Threads and the Scheduler in the Presence of Synchronization,” *Journal of Logic and Algebraic Programming*, vol. 78, no. 7, pp. 593–618, 2009.

[48] J. Boyland, “Checking Interference with Fractional Permissions,” in *Proceedings of the 10th International Symposium on Static Analysis (SAS)*, Springer LNCS 2694. Springer, 2003, pp. 55–72.

[49] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis, “Deny-Guarantee Reasoning,” in *Proceedings of the 18th European Symposium on Programming (ESOP)*, Springer LNCS 5502. Springer, 2009, pp. 363–377.

[50] P. W. O’Hearn, “Resources, Concurrency and Local Reasoning,” in *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR)*, Springer LNCS 3170. Springer, 2004, pp. 49–67.

[51] A. Hobor and C. Gherghina, “Barriers in Concurrent Separation Logic,” in *Proceedings of the 20th European Symposium on Programming (ESOP)*, Springer LNCS 6602. Springer, 2011, pp. 276–296.

[52] M. J. Parkinson, R. Bornat, and P. W. O’Hearn, “Modular Verification of a Non-blocking Stack,” in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2007, pp. 297–302.

[53] K. R. M. Leino and P. Müller, “A Basis for Verifying Multi-threaded Programs,” in *Proceedings of the 18th European Symposium on Programming (ESOP)*, Springer LNCS 5502. Springer, 2009, pp. 378–393.

[54] J. P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro, “Dynamically Checking Ownership Policies in Concurrent C/C++ Programs,” in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2010, pp. 457–470.

[55] Z. R. Anderson, D. Gay, and M. Naik, “Lightweight Annotations for Controlling Sharing in Concurrent Data Structures,” in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 98–109.