Hybrid Monitors for Concurrent Noninterference

Hybrid Monitors for Concurrent Noninterference

Aslan Askarov
Aarhus University
aslan@cs.au.dk

Stephen Chong
Harvard University
chong@seas.harvard.edu

Heiko Mantel
TU Darmstadt
mantel@cs.tu-darmstadt.de

***Abstract*—Controlling confidential information in concurrent systems is difficult, due to covert channels resulting from interaction between threads. This problem is exacerbated if threads share resources at fine granularity.**

In this work, we propose a novel monitoring framework to enforce strong information security in concurrent programs. Our monitors are hybrid, combining dynamic and static program analysis to enforce security in a sound and rather precise fashion. In our framework, each thread is guarded by its own local monitor, and there is a single global monitor. We instantiate our monitoring framework to support rely-guarantee style reasoning about the use of shared resources, at the granularity of individual memory locations, and then specialize local monitors further to enforce flow-sensitive progress-sensitive information-flow control. Our local monitors exploit rely-guarantee-style reasoning about shared memory to achieve high precision. Soundness of rely-guarantee-style reasoning is guaranteed by all monitors cooperatively. The global monitor is invoked only when threads synchronize, and so does not needlessly restrict concurrency. We prove that our hybrid monitoring approach enforces a knowledge-based progress-sensitive noninterference security condition.

***Keywords*-Language-based security; information-flow control for concurrent systems; hybrid information-flow monitor.**

I. Introduction

Computer systems increasingly exhibit concurrency, including systems that handle confidential information. Providing confidentiality in concurrent systems is challenging, as sharing of resources and synchronization may create *covert channels*, thus facilitating inadvertent leakage of sensitive information. An additional challenge is to enforce confidentiality without unnecessarily limiting or reducing concurrency.

Most previous mechanisms to control the flow of sensitive information (and to prevent information leaks) are inadequate in the presence of modern concurrency features. Operating system abstractions (e.g., [1, 2]) are too coarse grained to control information flow between concurrent threads that share fine-grained resources. Purely static mechanisms (e.g., [3, 4, 5, 6, 7]) are often too restrictive with respect to sharing of resources and synchronization between threads. Purely dynamic mechanisms (such as information-flow control monitors [8, 9, 10, 11]) have not yet been extended to handle fine-grained concurrency, due in part to the difficulties in dynamically and efficiently preventing covert channels resulting from thread interactions.

We investigate hybrid information-flow control for enforcing confidentiality in concurrent programs. *Hybrid information-flow control monitors* (or, simply, *hybrid monitors*) combine static and dynamic program analysis to secure the flow of information in a system. We present a general framework for monitoring concurrent programs, and instantiate this framework for information-flow security. In our framework, monitoring occurs at two levels: each thread is guarded by a local monitor, and there is a single global monitor to provide control across threads. We enable modular, rely-guarantee-style reasoning about the behavior of multi-threaded programs by extending the concepts of modes and mode states [12] to a dynamic setting. Our global monitor ensures that assumptions made by threads regarding the exclusive or shared use of memory are justified. Our local monitors ensure that individual threads provide the guarantees they promise. Our local monitors additionally control information flow within the guarded threads. The global monitor is not concerned with information-flow control; its sole purpose is to ensure that all assumptions are justified. Crucially, these assumptions can be exploited by the local monitors to establish information-flow control, both effectively and precisely.

Threads' assumptions about memory resources may change only at synchronization points. Thus, the global monitor is accessed only when threads synchronize, and the global monitor does not needlessly restrict concurrency in the program. Existing hybrid information-flow control monitors for concurrent programs (e.g., [13]) use a single monitor shared by all threads, thus causing unnecessary synchronization between threads.

This article's contributions can be summarized as follows.

- We provide efficient and precise information-flow control for concurrent programs using hybrid monitors. Our monitoring approach is novel: each thread has its own local monitor, and there is a single global monitor. Efficiency is achieved because the global monitor is accessed only at thread synchronization points, and so does not needlessly restrict concurrency. We prove that our monitors enforce a progress-sensitive noninterference-like security guarantee.
- The generic framework for monitoring concurrent programs is itself a novel technical contribution. It enables sound rely-guarantee-style reasoning in a dynamic setting, where the global monitor ensures the compatibility of assumptions made with guarantees provided by threads, and the local monitors enforce the guarantees promised by threads. Local monitors can be specialized to impose further constraints on the guarded thread. By exploiting assumptions, local monitors are able to effectively enforce also global system properties. This is the feature from which we benefit in our hybrid solution for information-flow security.

We consider a simple imperative concurrent calculus, with input and output operators and barrier synchronization. Our local monitors support flow-sensitive security types [14], which facilitates precise reasoning about information flow via program variables. A thread may soundly allow the security level of a variable to change only if the thread has exclusive read-access or exclusive write-access to the variable.

Our local monitors separately track upper bounds on information that may be revealed by the relative order of events (the *timing level* of a thread) and information that may be revealed by learning whether control has reached the current program point or diverged earlier (the *termination level* of a thread). These levels are analogous to the *pc level* traditionally used in security-type systems for non-concurrent programs [3]. The termination level enables us to enforce *progress sensitive* security [15], a generalization of *termination sensitive* security [16] to interactive programs. Most existing work on enforcing information-flow security ignores progress sensitivity due to the complexity and imprecision of the enforcement mechanism. Use of the timing level enables us to prevent *internal timing leaks* [17]. Although the termination level of a thread increases monotonically, we provide additional precision by lowering the timing level of threads at synchronization points, since synchronization between threads restricts the relative order of events before and after the synchronization. Our work leverages the insight that resetting the timing levels of threads at synchronization points is analogous to lowering the pc level at post-dominators of control-flow branches [17, 3, 18].

II. Information Flow in Concurrent Programs

In this section, we provide intuition for how concurrency and progress sensitivity complicate the enforcement of information-flow security, and for how our monitors enforce security in this challenging setting. We then present our general framework for monitoring concurrent programs (Sections III–IV), instantiate it for rely-guarantee reasoning (Section V), and further instantiate it to enforce information-flow security (Sections VI–VII). Due to space restrictions, we provide only proof sketches. A full version of this article [19] provides detailed proofs and additional material, such as calculus rules that we omit here.

Information flow through concurrent access: Confidential information may be inadvertently leaked through thread interaction. For example, consider the following (insecure) program, which consists of two threads that execute concurrently.

```
Thread 1:  input H to foo; output foo × 2 to H
Thread 2:  foo := 42; output foo to L
```

The first thread inputs confidential information from high-security channel H, stores it in variable foo, and then outputs 2 times foo to channel H. The other thread sets foo to the constant integer 42, then outputs foo to low-security channel L, which we assume can be observed by an adversary. Each of these threads is intuitively secure if it were executed in isolation. However, when executed concurrently, since they both access foo, it is possible that Thread 2 will output confidential information to channel L, violating security by revealing confidential information to the attacker. Indeed, if an observer of channel L sees an output of anything other than 42, she can infer the confidential input.

Fine-grained resource sharing: Following the work of Mantel, Sands, and Sudbrock [12], we support fine-grained reasoning about threads' access to shared resources (like the variable foo in the example above) in order to prevent security violations via thread interaction through such resources. Each thread uses rely-guarantee reasoning about what variables it and other concurrently executing threads might read or write. Consider, for example, the following (secure) program, where Thread 1 assumes that no other threads will read variable w (note that Thread 1 stores confidential information in w), and Thread 2 assumes that no other threads will write variable v (note that Thread 2 sends the content of v to channel L). Provided both assumptions hold, the program is secure, even in the presence of additional threads.

```
Thread 1:  input H to w; output w + v to H
Thread 2:  v := 14; output v × 42 to L
```

Threads' assumptions and guarantees originate from protocols the concurrently running threads comply with, where coordination among threads is often achieved through synchronization mechanisms. In the following (secure) example, Thread 1 has exclusive access to variable y before the barrier, and Thread 2 has exclusive access after the barrier.

```
Thread 1:  input L to y; barrier
Thread 2:                 barrier; output y to L
```

Assumptions and guarantees of threads must be compatible when threads are composed. We assume that a thread's assumptions are stated explicitly. We use the global monitor to impose the corresponding guarantees on all other threads, and use the local monitors of the threads ensure that these guarantees are indeed provided. Thus, our approach supports fine-grained resource sharing while preventing information leakage through concurrent access.

Information flow through nondeterminism: Information can also be leaked when the relative order of observable events (such as outputs to low-security channels) depend on confidential information. In the following (insecure) example, the output on channel L is either 0 followed by 1, or vice versa, where the order likely depends on confidential information.

```
Thread 1:  input h to H; while h > 0 do h := h − 1 od;
                 output 1 to L
Thread 2:  output 0 to L
```

Our monitors prevent information leakage by this program as follows: Regardless of the value of the confidential input, Thread 1's local monitor notices that the relative ordering between its own output and output of other threads might be influenced by the while loop, and, hence, the local monitor intervenes by blocking the thread's execution before the output occurs. The following program is, however, secure and permitted by our monitors.

```
Thread 1:  input h to H; while h > 0 do h := h − 1 od;
                 barrier; output 1 to L
Thread 2:  barrier; output 0 to L
```

Even though the relative order of the low outputs is undetermined, the threads cannot perform the output until after the barrier. Intuitively, the program is secure because the synchronization between threads ensures that the order of the low output does not depend on confidential information, even though the low output is nondeterministic [20, 21].

To correctly and precisely track what information might be revealed by the relative timing of events, each local monitor tracks the *timing level*, a security level that is an upper bound on information that has influenced the timing of the thread's execution with respect to other threads. Since after synchronization, the relative timing of threads is independent of information that affected the timing before the synchronization, the timing level of a thread can be lowered immediately after synchronization. This is similar to information-flow control in single-threaded programs, where the *pc level* (a bound on the information that influences whether the current statement is executed) can be lowered at post-dominators of control flow decisions. That is, synchronization points are the post-dominators of concurrent programs, and allow a similar improvement in precision.

Precision of hybrid monitors: Our use of monitors improves the precision of security enforcement, since we consider the security of a single execution, and do not need to determine whether all possible program executions are secure. For example, our monitor permits complete execution of the following single-threaded program if the low input is positive. In contrast, a static analysis would have to classify this program as insecure, because executing it will leak confidential information if the low input is negative or zero.

```
input $L$ to low;
if low $>$ 0 then input $L$ to x else input $H$ to x fi;
output x to $L$
```

Progress sensitivity: If confidential information influences whether a program terminates, observing the termination or non-termination of a program can leak confidential information. This is exacerbated in interactive settings [15] (i.e., where the program produces observable events before the end of the program) and concurrent settings (where many threads may each reveal a small amount of information). For example, in the following (insecure) program, the program silently diverges after outputting a low value that is equal to the secret (i.e., it diverges without producing any further output). Thus, an observer of channel L can learn the secret by noting the last value output.

```
input $H$ to high; low := 0;
while 1 do
   output low to $L$;
   if low<high then low := low+1 else (while 1 do skip od) fi
od
```

An additional concern is that an enforcement mechanism itself might reveal information. In the following (insecure) program, if the secret is positive, a monitor might intervene to prevent the insecure output of high to L by blocking the thread. As a consequence, the output of 42 to channel L would never occur and, hence, an observer of L could infer that the secret must be a positive value, once she realizes that 42 will not be output.

```
input $H$ to high;
if high $>$ 0 then output high to $L$ else skip fi;
output 42 to $L$
```

We prevent information leaks via termination and monitor interventions by tracking the *termination level* and *blocking level* of each thread within our local monitors. These are upper bounds on information that might have influenced, respectively, the termination behavior of loops and the blocking behavior of the monitor. By tracking these levels, we ensure that the termination and blocking behavior does not leak information.

III. Monitored Multi-threaded Computations

We propose a formal model for multi-threaded programs that interact with their environment via channels and whose concurrent threads communicate with each other using shared memory. In our model, we reuse the concepts of modes and of mode states from [12] to facilitate rely-guarantee-style reasoning about the behavior of such programs.

The novelty of our model is that it supports reasoning about multi-threaded programs that are monitored. In our model, programs can be monitored at two levels: local monitors provide control over individual threads while global monitors provide control across threads. By lifting the concept of mode states to threads that are monitored, we enable rely-guarantee-style reasoning both about monitored programs and within monitors.

We recall the concepts of modes and of mode states in Section III-A, and also define a formal notation for mode state changes. In Section III-B, we introduce judgments that capture the behavior of local and global monitors. This provides the basis for lifting the concepts of mode states to multi-threaded programs that are monitored in Section III-C.

Notation: We denote the powerset of a set S by $\mathcal{P}(S)$. We use $A \to B$ and $A \rightharpoonup B$ to denote the set of all total functions and all partial functions, respectively, with domain A and range B. We denote the pre-image of a function $f : A \rightharpoonup B$ by $pre(f)$, i.e., $pre(f) = \{a \in A \mid f(a) \in B\}$. We denote the update of a function f by $f[d \mapsto v]$, where $f[d \mapsto v](d) = v$ and $f[d \mapsto v](d') = f(d')$ for $d' \in pre(f) \setminus \{d\}$.

We refer to partial functions with domain $\mathbb{N}_0$, range A, and a finite pre-image of consecutive numbers starting at 0 as *lists over* A, and denote the set of all such lists by A^*. We use ϵ to denote the empty list, $l{\cdot}a$ to denote the list that results from appending an element $a \in A$ to the end of a list $l \in A^*$, and $l_1{\cdot}l_2$ to denote the concatenation of two lists $l_1, l_2 \in A^*$. The function filter removes from a list over A those elements that do not satisfy a predicate $p \subseteq A$, i.e., $\mathsf{filter}(p, \epsilon) = \epsilon$, $\mathsf{filter}(p, l{\cdot}a) = \mathsf{filter}(p, l){\cdot}a$ if $a \in p$, and $\mathsf{filter}(p, l{\cdot}a) = \mathsf{filter}(p, l)$ if $a \notin p$.

A. Modes, Mode States, and Annotations

We use *modes* [12] to capture a program developer's expectations about the environment in which a program shall run as well as his intentions. For instance, a program might be designed to use a particular communication protocol with its environment and, hence, a thread running such a program will provide the desired functionality only in an environment that complies with this protocol. Such a convention incorporates both an expectation (namely that the environment will follow the protocol) and an intention (namely that the thread itself is programmed correctly to follow the protocol).

In general, modes can be used to express assumptions and guarantees about arbitrary entities that are relevant for a program's behavior. In this article, we restrict ourselves to modes that express assumptions and guarantees about particular entities, namely program variables. More concretely, we focus on assumptions about which variables might be read and written by a thread's environment and the dual guarantees. We use Var to denote the set of all variables.

We use the symbols A-NR and A-NW to express the assumption that a particular variable will not be read and will not be written, respectively, by the environment of a given thread. Moreover, we use the symbols G-NR and G-NW to express the guarantee that a thread will not read and will not write, respectively, a particular variable. That is, the modes A-NR and G-NR are dual to each other, and so are the modes A-NW and G-NW. We refer to the modes A-NR and A-NW as the *no-read* and the *no-write assumption*. Moreover, we refer to the modes G-NR and G-NW as the *no-read* and the *no-write guarantee*. Formally, we define the *set of assumptions* by $\mathit{Asm} = \{\mathsf{A\text{-}NR}, \mathsf{A\text{-}NW}\}$, the set of guarantees by $\mathit{Gua} = \{\mathsf{G\text{-}NR}, \mathsf{G\text{-}NW}\}$, and the *set of all modes* by $\mathit{Mod} = \mathit{Asm} \cup \mathit{Gua}$.

We use *mode states* [12] to track which assumptions are made and which guarantees are provided. Formally, a mode state is a function $\mathit{mdst} : \mathit{Mod} \to \mathcal{P}(\mathit{Var})$ that returns the set of all variables that are in a given mode. Accordingly, we define the set of all mode states by $\mathit{MdSt} = \mathit{Mod} \to \mathcal{P}(\mathit{Var})$.

We use terms of the form $\mathsf{acq}(\mathit{mod}, X)$ and $\mathsf{rel}(\mathit{mod}, X)$ to specify that the mode mod is acquired and released, respectively, for all variables in the set $X \subseteq \mathit{Var}$. We call such terms *annotations* and define the set of all annotations by $\mathit{Ann} = \{\mathsf{acq}(\mathit{mod}, X), \mathsf{rel}(\mathit{mod}, X) \mid \mathit{mod} \in \mathit{Mod}, X \subseteq \mathit{Var}\}$. For singleton sets of variables, we use a shorthand notation and write $\mathsf{acq}(\mathit{mod}, x)$ instead of $\mathsf{acq}(\mathit{mod}, \{x\})$. Similarly, we write $\mathsf{rel}(\mathit{mod}, x)$ instead of $\mathsf{rel}(\mathit{mod}, \{x\})$.

We introduce the predicate *Has-Mode-In* $\subseteq \mathit{Ann} \times \mathcal{P}(\mathit{Mod})$ to identify annotations with particular modes. We define this predicate by ann *Has-Mode-In* M iff $\mathit{ann} \in \{\mathsf{acq}(\mathit{mod}, X), \mathsf{rel}(\mathit{mod}, X) \mid X \subseteq \mathit{Var} \wedge \mathit{mod} \in M\}$ and the projection of a list of annotations γ to a set of modes M by $\gamma \restriction M = \mathsf{filter}(\lambda \mathit{ann} \in \mathit{Ann} : \mathit{ann}\ \textit{Has-Mode-In}\ M, \gamma)$.

To model the effects of annotations on mode states, we define the function $\mathit{update} : (\mathit{MdSt} \times \mathit{Ann}) \to \mathit{MdSt}$ by

$$\begin{aligned} &\mathit{update}(\mathit{mdst}, \mathsf{acq}(\mathit{mod}, X)) = \\ &\qquad \mathit{mdst}[\mathit{mod} \mapsto (\mathit{mdst}(\mathit{mod}) \cup X)] \\ &\mathit{update}(\mathit{mdst}, \mathsf{rel}(\mathit{mod}, X)) = \\ &\qquad \mathit{mdst}[\mathit{mod} \mapsto (\mathit{mdst}(\mathit{mod}) \setminus X)] \end{aligned}$$

Overloading notation, we lift the function $\mathit{update} : (\mathit{MdSt} \times \mathit{Ann}) \to \mathit{MdSt}$ to a function $\mathit{update} : (\mathit{MdSt} \times \mathit{Ann}^\star) \to \mathit{MdSt}$ on lists of annotations by $\mathit{update}(\mathit{mdst}, \epsilon) = \mathit{mdst}$ and $\mathit{update}(\mathit{mdst}, \gamma \cdot \mathit{ann}) = \mathit{update}(\mathit{update}(\mathit{mdst}, \gamma), \mathit{ann})$, where $\mathit{mdst} \in \mathit{MdSt}$, $\mathit{ann} \in \mathit{Ann}$, and $\gamma \in \mathit{Ann}^\star$.

For instance, the annotation $\mathsf{acq}(\mathsf{A\text{-}NW}, x)$ can be used to specify that a thread from now on runs under the assumption that variable x is not written by other threads, $\mathit{mdst}(\mathsf{A\text{-}NW})$ equals the set of all variables for which the no-write assumption is made in mode state mdst, and, in particular, $x \in (\mathit{update}(\mathit{mdst}, \mathsf{acq}(\mathsf{A\text{-}NW}, x)))(\mathsf{A\text{-}NW})$ holds.

B. Monitors and Events

To control the behavior of individual threads, each thread is guarded by a *local monitor* which is invoked each time the thread performs a computation step. We use *local events* to capture information about computation steps needed by a local monitor. We denote the set of all local events by Ev, use $\alpha \in \mathit{Ev}$ as a meta-variable, use LMon to denote the set of all possible internal states of local monitors, and use $\mathit{lmon} \in \mathit{LMon}$ as a meta-variable for local monitor states.

We employ the judgment $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{\mathit{perm}} \mathit{lmon}'$ to capture that a local monitor in state $\mathit{lmon} \in \mathit{LMon}$ permits the combination of $\alpha \in \mathit{Ev}$ and $\delta \in \mathit{Ann}^\star$. The local monitor's internal state is updated to $\mathit{lmon}' \in \mathit{LMon}$.

To control the behavior across threads, a multi-threaded program is guarded by one *global monitor*. We use *global events* to capture the information about such steps that is needed by the global monitor. We denote the set of all global events by GEv, use $\beta \in \mathit{GEv}$ as a meta-variable, use GMon_n to denote the set of all possible internal states of global monitors for pool states with n threads, and identify the initial global monitor states in these sets by $\mathit{gmon}_{\mathit{init},n} \in \mathit{GMon}_n$. We define the set of all global monitor states by $\mathit{GMon} = \bigcup_{n \in \mathbb{N}_0} \mathit{GMon}_n$ and use $\mathit{gmon} \in \mathit{GMon}$ as a meta-variable.

In this article, we use the global monitor only to control mode-state updates. To simplify our exposition, we assume that threads update their mode state only when synchronizing with other threads, and we consider only one synchronization primitive, namely *global barrier synchronization*. Consequently, the global monitor needs to be invoked only when the program passes a barrier and only needs to distinguish between two global events: sync for barrier synchronization and ϵ for all other steps. Throughout this article, the set of all global events is $\mathit{GEv} = \{\mathsf{sync}, \epsilon\}$. We use a function $\chi : \mathit{Ev} \to \mathit{GEv}$ to extract the corresponding global event from a local event.

When passing a barrier, each alive thread requests changes to its mode state, and the global monitor decides if a given combination of requests by the individual threads is permissible. Moreover, the global monitor may decide to impose mode-state changes on threads that differ from the requested mode-state changes. We employ the judgment $\mathit{gmon} \longrightarrow^{\Gamma,\Delta} \mathit{gmon}'$ to capture that a global monitor in state $\mathit{gmon} \in \mathit{GMon}$ accepts the combination of mode-state-change requests modeled by Γ, imposes the mode-state changes modeled by Δ in response, and updates its internal state to gmon'. Formally, Γ and Δ are functions of type $N \to \mathit{Ann}^\star$, where $N \subseteq \mathbb{N}_0$ is a finite set. That is, Γ and Δ return a list of annotations for each number in their pre-image. As explained later, whenever we use this judgment, N is the set of identifiers of all threads that are alive when the global monitor is invoked.

We provide a concrete instantiation of GMon_n and $\mathit{gmon}_{\mathit{init},n}$ together with a calculus for global monitor transitions (i.e., for deriving $gmon \longrightarrow^{\Gamma,\Delta} gmon'$) in Section V. In Section VI, we provide instantiations of Ev and of χ. An instantiation of $LMon$ and a calculus for local monitor transitions are presented in Section VII.

C. Monitored, Multi-threaded Computations

We capture control states of individual threads by terms that we call *commands*, use Com to denote the set of all commands, and express that a thread has terminated by the special command $\mathsf{term} \in Com$.

We capture local states of individual threads by *thread states* and collections of the local states of multiple threads by *pool states*. Formally, a *thread state* is a triple $[com, lmon, mdst]$, where $com \in Com$, $lmon \in LMon$, and $mdst \in MdSt$. Accordingly, we define the set of all thread states by $ThSt = Com \times LMon \times MdSt$. Formally, a *pool state* is a list of thread states. We define the set of all pool states with n threads by $PSt_n = \{0, \ldots, n-1\} \to ThSt$ and define the set of all pool states with arbitrarily many threads by $PSt = \bigcup_{n \in \mathbb{N}_0} PSt_n$. We use $thread \in ThSt$ as meta-variable for thread states and $pool \in PSt$ as a meta-variable for pool states.

From a thread state $thread = [com, lmon, mdst]$, we retrieve its components with selector functions, defined by $thread.\mathsf{com} = com$, $thread.\mathsf{lmon} = lmon$, and $thread.\mathsf{mdst} = mdst$. For instance, $(pool(i)).\mathsf{mdst}$ equals the mode state of the ith thread in the pool state $pool$.

We introduce the predicate $Alive \subseteq ThSt$ to capture that a thread has not yet terminated by $Alive(thread) \equiv thread.\mathsf{com} \neq \mathsf{term}$. To retrieve the identifiers of those threads in a given pool state $pool \in PSt$ that have not yet terminated and that have terminated, respectively, we define the functions $alive : PSt \to \mathcal{P}(\mathbb{N}_0)$ and $terminated : PSt \to \mathcal{P}(\mathbb{N}_0)$ by $alive(pool) = \{i \in pre(pool) \mid Alive(pool(i))\}$ and $terminated(pool) = pre(pool) \setminus alive(pool)$.

Threads communicate with each other via a globally shared memory and with their environment via channels. We model the *state of the shared memory* by a function from variables to values and the history of prior communications on a given channel by a trace. Formally, we use Val to denote the set of all *values*, define the set of states of the shared memory by $Mem = Var \to Val$, and use Ch to denote the set of all *channels*. We model that a value $v \in Val$ is received from channel $ch \in Ch$ and that v is output on ch by the terms $\mathsf{inp}(ch, v)$ and $\mathsf{out}(ch, v)$, respectively. We refer to such terms as *interactions* and denote the set of all interactions by IO. We call lists of interactions *traces*, define the set of all traces by $Tr = IO^\star$, and use $\tau \in Tr$ as a meta-variable for traces.

We capture the behavior of a program's environment by a *communication strategy*. A strategy determines which input the environment supplies to a program on a given channel after a sequence of prior interactions. Formally, a strategy is a function $\sigma : (Tr \times Ch) \to Val$ such that $\sigma(\tau, ch)$ is the value supplied next on channel ch if all prior interactions are captured by the trace τ. We use Σ to denote the set of all such strategies.

Configurations: For capturing snapshots during program execution, we employ three layers of configurations: *global configurations*, *local configurations*, and *command configurations*, where global and local configurations incorporate the state of a global monitor and of a local monitor, respectively.

We use *global configurations* to model global snapshots during a program run. Formally, a global configuration $gcnf$ is a quadruple $\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$, where $pool \in PSt$, $mem \in Mem$, $\tau \in Tr$, and $gmon \in GMon$. In the global configuration $gcnf$, the pool state $pool$ captures the local state of each thread of the multi-threaded program, the memory state mem captures the content of all memory locations, the trace τ captures the inputs and outputs that have occurred so far, and $gmon$ captures the internal state of the global monitor.

We use *local configurations* to capture the local view of individual threads during a run. Formally, a local configuration $lcnf$ is a triple $\langle thread, mem, \tau \rangle$ where $thread \in ThSt$, $mem \in Mem$, and $\tau \in Tr$. While the thread state $thread$ models a thread's local state, mem and τ model the content of the memory and the prior communications, respectively. In a global configuration $\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$, the local configuration of thread i is $\langle pool(i), mem, \tau \rangle$.

We use *command configurations* to define the local effects of computation steps. Formally, a command configuration $ccnf$ is a triple (com, mem, τ), where $com \in Com$, $mem \in Mem$, and $\tau \in Tr$. While com models a thread's internal state, mem and τ model the memory content and the prior communications, respectively. The command configuration of a local configuration $\langle [com, lmon, mdst], mem, \tau \rangle$ is (com, mem, τ).

We use $GCnf$, $LCnf$, and $CCnf$ to denote the set of all global, local, and command configurations, respectively.

Judgments: For capturing the effects of computation steps at the level of global, local, and command configurations, respectively, we employ the following three judgments:

$$gcnf \twoheadrightarrow_\sigma gcnf' \qquad lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf' \qquad ccnf \overset{\alpha,\gamma}{\rightarrowtriangle}_\sigma ccnf'$$

A strategy $\sigma \in \Sigma$ appears as a subscript of the arrow in all three judgments. It captures the communication strategy of the program's environment. For instance, the first judgment captures that a transition from a global configuration $gcnf$ to a global configuration $gcnf'$ is possible under the strategy σ. The arrow in the second judgment carries three additional annotations: a global event $\beta \in GEv$ and two lists of annotations $\gamma, \delta \in Ann^\star$ that serve different purposes. While γ captures which changes to its mode state a thread desires, δ captures which mode state changes are imposed on the thread in response. The arrow in the third judgment carries as annotations a local event $\alpha \in Ev$ and a list of annotations $\gamma \in Ann^\star$ that captures which changes to its mode state a thread desires.

In the remainder of this section, we provide calculi for the judgments $lcnf \xrightarrow{\beta,\gamma,\delta}_\sigma lcnf'$ and $gcnf \twoheadrightarrow_\sigma gcnf'$. An exemplary calculus for the judgment $ccnf \overset{\alpha,\gamma}{\rightarrowtriangle}_\sigma ccnf'$ is provided in Section VI together with an instantiation of Com.

As usual, these calculi induce transition relations. For instance, the calculus for local configurations induces a family of transition relations $(\xrightarrow{\beta,\gamma,\delta}_{\sigma})_{\beta \in GEv, \gamma, \delta \in Ann^\star, \sigma \in \Sigma}$, where $\xrightarrow{\beta,\gamma,\delta}_{\sigma}$ relates two local configurations $lcnf$ and $lcnf'$ iff $lcnf \xrightarrow{\beta,\gamma,\delta}_{\sigma} lcnf'$ is derivable. We use the usual notation for the reflexive, transitive closure of such relations, i.e., for instance, $(\xrightarrow{\beta,\gamma,\delta}_{\sigma})^*$ denotes the reflexive, transitive closure of $\xrightarrow{\beta,\gamma,\delta}_{\sigma}$.

Local Transitions: The judgment $lcnf \xrightarrow{\beta,\gamma,\delta}_{\sigma} lcnf'$ defines which transitions between local configurations are possible. The only rule for deriving instances of this judgment is depicted in Fig. 1.

The judgment for transitions between command configurations in the first premise of the rule in Fig. 1 reflects that a thread's behavior is determined by the command that this thread is executing. In this article, we employ local monitors to guard the behavior of individual threads. Accordingly, the judgment for transitions between local monitor states is used in the second premise to capture that the local monitor must deem the step acceptable. Note that the transition between local monitor states is based on δ and not on γ, i.e., it is based on the mode state changes imposed on the thread, not on the mode state changes requested by the thread. Also note that δ is used to update the mode state in the third premise. The global event β is extracted from the local event α using the function $\chi : Ev \to GEv$ in the last premise of the rule.

Global Transitions: Transitions between global configurations are captured by the judgment $gcnf \twoheadrightarrow_{\sigma} gcnf'$. To simplify our presentation, we make three restrictions. First, we assume that the process structure is static, i.e., if $\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle \twoheadrightarrow_{\sigma} \langle\!\langle pool', mem', \tau', gmon' \rangle\!\rangle$ then $pre(pool') = pre(pool)$. Second, we assume that threads are scheduled nondeterministically. Third, as stated before, we assume that barrier synchronization is the only synchronization primitive and that threads only request changes to their mode state when they pass a barrier.

The two rules for deriving instances of the judgment $gcnf \twoheadrightarrow_{\sigma} gcnf'$ are depicted in Fig. 2. The first rule captures steps by individual threads, and the second rule captures a barrier synchronization across all threads.

According to the first rule in Fig. 2, an alive thread i (first premise) may be chosen nondeterministically to perform a computation step (second premise). This step must not involve synchronization (third premise), must neither request nor impose mode state changes (fourth premise), and must not affect the global monitor state (fifth premise). Such a step by an individual thread might affect the thread's control state, the state of the local monitor supervising this thread, the shared memory, and the trace. It cannot affect the mode state of this thread (since $\delta = \epsilon$), the local states of other threads (conclusion of the rule), and the global monitor state (since $gmon' = gmon$).

The second rule in Fig. 2 captures synchronization steps. This rule requires that all alive threads jointly pass a barrier (last premise of the rule), which faithfully reflects the intuition of a barrier synchronization. In order to perform a synchronization step, at least one thread must be alive (first premise). The function Γ captures which mode state changes are requested by the individual alive threads (last premise). That is, $\Gamma(i)$ is the list of annotations capturing the mode state changes requested by the ith thread, where $i \in alive(pool)$. The function Δ captures the mode state changes that are imposed on the individual threads (last premise). Which mode state changes are imposed on the individual threads in response to their requests is determined by the global monitor (third premise). Note that the set of threads cannot change during a synchronization step (fourth premise) and that the thread states of terminated threads remain unmodified (fifth premise). Also note that a synchronization step cannot affect the shared memory or the trace (conclusion of the rule). Synchronization steps can only affect the thread state of all alive threads and the state of the global monitor (conclusion of the rule).

Reachability: We say that *a global configuration $gcnf'$ is reachable from a global configuration $gcnf$ under a strategy σ* iff $gcnf\ (\twoheadrightarrow_{\sigma})^*\ gcnf'$ holds. We assume that runs of multi-threaded programs start in an initial memory mem_{init} that assigns a dedicated value $v_{init} \in Val$ to all variables (i.e., $\forall x \in Var : mem_{init}(x) = v_{init}$) and with an empty initial trace τ_{init} (i.e., $\tau_{init} = \epsilon$). We say that *a global configuration $gcnf'$ is reachable from a pool state $pool$ under a strategy σ* iff $gcnf'$ is reachable from the global configuration $\langle\!\langle pool, mem_{init}, \tau_{init}, gmon_{init,n} \rangle\!\rangle$ under σ, where $n = |pre(pool)|$. We use $greach_{\sigma}(gcnf)$ and $reach_{\sigma}(pool)$ to denote the set of all global configurations reachable from $gcnf \in GCnf$ and $pool \in PSt$, respectively, under $\sigma \in \Sigma$.

IV. Semantics of Modes

We formally define what it means for a thread to provide a guarantee and what it means for an assumption to be justified. While we define the semantics of G-NR and G-NW in terms of transitions between local configurations, we define the semantics of A-NR and A-NW for a given thread in terms of guarantees given by other threads in a global configuration. Based on the formal semantics of modes, we define conditions that allow one to soundly exploit assumptions when reasoning about the behavior of multi-threaded programs.

Semantics of G-NW and G-NR: We say that *a local configuration $lcnf = \langle thread, mem, \tau \rangle$ provides the no-write guarantee for a variable x* iff the value of x will remain unmodified by each next possible step of the thread. This requirement is captured by the following formula:

$$\begin{array}{l}\forall \sigma \in \Sigma : \forall \beta \in GEv : \forall \gamma, \delta \in Ann^\star : \\ \forall \langle thread', mem', \tau' \rangle \in LCnf : \\ \langle thread, mem, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle thread', mem', \tau' \rangle \\ \quad \Longrightarrow\ mem'(x) = mem(x)\end{array}$$

$$\frac{(com, mem, \tau) \overset{\alpha,\gamma}{\rightarrowtriangle}_{\sigma} (com', mem', \tau') \qquad lmon \longrightarrow^{\delta,\alpha}_{perm} lmon' \qquad mdst' = update(mdst, \delta) \qquad \beta = \chi(\alpha)}{\langle [com, lmon, mdst], mem, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle [com', lmon', mdst'], mem', \tau' \rangle}$$

Fig. 1. Transitions between local configurations

$$\frac{i \in alive(pool) \qquad \langle pool(i), mem, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle thread', mem', \tau' \rangle \qquad \beta = \epsilon \qquad \gamma = \delta = \epsilon \qquad gmon' = gmon}{\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle \twoheadrightarrow_{\sigma} \langle\!\langle pool[i \mapsto thread'], mem', \tau', gmon' \rangle\!\rangle}$$

$$\frac{\begin{array}{c} alive(pool) \neq \emptyset \qquad \Gamma, \Delta : alive(pool) \longrightarrow Ann^{\star} \qquad gmon \longrightarrow^{\Gamma,\Delta} gmon' \qquad pool' \in PSt_{|pre(pool)|} \\ \forall j \in terminated(pool) : pool'(j) = pool(j) \qquad \forall i \in alive(pool) : (\langle pool(i), mem, \tau \rangle \xrightarrow{\mathsf{sync}, \Gamma(i), \Delta(i)}_{\sigma} \langle pool'(i), mem, \tau \rangle) \end{array}}{\langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle \twoheadrightarrow_{\sigma} \langle\!\langle pool', mem, \tau, gmon' \rangle\!\rangle}$$

Fig. 2. Transitions between global configurations

We say that *a local configuration* $lcnf = \langle thread, mem, \tau \rangle$ *provides the no-read guarantee for a variable* y iff

$$\begin{array}{l} \forall \sigma \in \Sigma : \forall \beta \in GEv : \forall \gamma, \delta \in Ann^{\star} : \\ \forall \langle thread', mem', \tau' \rangle \in LCnf : \forall v \in Val : \\ \langle thread, mem, \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle thread', mem', \tau' \rangle \\ \quad \Longrightarrow \langle thread, mem[y \mapsto v], \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle thread', mem', \tau' \rangle \vee \\ \qquad\quad \langle thread, mem[y \mapsto v], \tau \rangle \xrightarrow{\beta,\gamma,\delta}_{\sigma} \langle thread', mem'[y \mapsto v], \tau' \rangle \end{array}$$

That is, a no-read guarantee for a variable y ensures that changing the value of y before a computation step does not alter the effects of this computation step. The two disjuncts on the right-hand side of the implication correspond respectively to the cases where y is overwritten and where y is not overwritten in a computation step.

A local configuration $lcnf = \langle [com, lmon, mdst], mem, \tau \rangle$ *provides its guarantees* iff it provides the no-write guarantee for each variable $x \in mdst(\mathsf{G\text{-}NW})$ and the no-read guarantee for each $y \in mdst(\mathsf{G\text{-}NR})$. Consequently, if $lcnf$ provides its guarantees then the values of all variables in $mdst(\mathsf{G\text{-}NW})$ will remain unchanged by the thread's next step and the values of all variables in $mdst(\mathsf{G\text{-}NR})$ will not affect the thread's next step. We say that *a thread state thread provides its guarantees* iff, for all $mem \in Mem$ and all $\tau \in Tr$, the local configuration $\langle thread, mem, \tau \rangle$ provides its guarantees. Moreover, we say that $gcnf = \langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$ *provides its guarantees* iff $pool(i)$ provides its guarantees for all $i \in pre(pool)$.

Semantics of A-NW and A-NR: Given a global configuration $gcnf = \langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$, we say that *gcnf justifies the assumption* A-NW *of a thread* $i \in pre(pool)$ *about a variable* x iff every other alive thread has acquired the mode G-NW for x. Similarly, we say that *gcnf justifies the assumption* A-NR *of a thread* $i \in pre(pool)$ *about a variable* y iff every other alive thread has acquired the mode G-NR for y.

A global configuration $gcnf = \langle\!\langle pool, mem, \tau, gmon \rangle\!\rangle$ *justifies its assumptions* iff $gcnf$ justifies both the assumption A-NW about each variable in $(pool(i)).\mathsf{mdst}(\mathsf{A\text{-}NW})$ and the assumption A-NR about each variable in $(pool(i)).\mathsf{mdst}(\mathsf{A\text{-}NR})$ for all $i \in pre(pool)$. Note that assumptions of all threads, including terminated threads, must be justified. In contrast, only alive threads need to explicitly provide the dual guarantees for assumptions of other threads. Terminated threads need not acquire the modes G-NW and G-NR because, it is clear that they will not be able to write or read variables in the future.

Sound use of modes: We say that *a global configuration gcnf ensures a sound use of modes* iff, for each strategy $\sigma \in \Sigma$, each reachable global configuration $gcnf' \in greach_{\sigma}(gcnf)$ provides its guarantees and justifies its assumptions.

Moreover, we say that *a pool state pool ensures a sound use of modes* iff $\langle\!\langle pool, mem_{init}, \tau_{init}, gmon_{init,n} \rangle\!\rangle$ ensures a sound use of modes. If a pool state $pool$ ensures a sound use of modes then, at each intermediate state during each possible run, each assumption made is justified by guarantees that are, indeed, provided. Hence, all assumptions made can be exploited soundly when reasoning about possible behaviors.

V. A Monitoring Framework

We propose a framework for monitoring multi-threaded programs based on our model of computation from Section III. Our monitoring framework consists of the definition of a global monitor and of a local monitor. The role of these monitors is complementary. Our global monitor ensures that assumptions made by threads are, indeed, justified. Our local monitor ensures that an individual thread, indeed, provides the guarantees that it promises to provide. The combination of one global monitor and a local monitor at each thread jointly ensure a sound use of modes and, hence, the soundness of modular, rely-guarantee-style reasoning about the behavior of multi-threaded programs.

Our local monitor can be specialized to enforce additional properties. By exploiting assumptions, local monitors can not only establish properties of individual threads, but also global properties of entire multi-threaded programs. We present a specialization of local monitors for information-flow control in Section VII and demonstrate that this specialization soundly enforces end-to-end information-flow security.

Global Monitoring of Multi-threaded Programs: We define a global monitor that grants all acquisitions and releases of assumptions exactly as desired by each thread. In addition, our global monitor ensures that all assumptions of all threads are justified. To justify all assumptions, guarantees might be needed that differ from the guarantees that the individual threads desire to provide. Consequently, our global monitor cannot always grant modifications of guarantees according to these desires.

Our global monitor keeps track of both the assumptions that each alive thread currently makes and the assumptions that each terminated thread had made when it terminated. Formally, *a global monitor state for n threads* is a function that returns a mode state for each thread identifier in $\{0,\ldots,n-1\}$. Accordingly, we instantiate the *set of all global monitor states for n threads* by $GMon_n = \{0,\ldots,n-1\} \longrightarrow MdSt$ and $gmon_{init,n} \in GMon_n$, the *initial global monitor state for n threads*, by $gmon_{init,n}(i) = \{\}$ for all $i \in \{0,\ldots,n-1\}$.

We say that *a global monitor state $gmon \in GMon$* (recall $GMon = \bigcup_{n\in\mathbb{N}_0} GMon_n$) *is compatible with a pool state $pool \in PSt$* iff $pre(gmon) = pre(pool)$ and if $(gmon(i))(mod) = (pool(i)).\mathsf{mdst}(mod)$ for each $i \in pre(pool)$ and $mod \in Asm$.

When threads modify their mode state, the global monitor state is updated accordingly. To capture such updates of the global monitor state, we define the function $\mathit{gmon\text{-}update} : (GMon \times (\mathbb{N}_0 \rightharpoonup Ann^\star)) \longrightarrow GMon$ by

$$\begin{array}{l}
\mathit{gmon\text{-}update}(gmon,\Gamma) = \\
\lambda i \in pre(gmon) : \\
\mathit{if}\ \ i \in pre(\Gamma)\ \ \mathit{then}\ \ \mathit{update}(gmon(i),(\Gamma(i) \restriction Asm)) \\
\qquad\qquad\qquad\quad \mathit{else}\ \ gmon(i)
\end{array}$$

Note that acquisitions and releases of guarantees in $\Gamma(i)$ are ignored when updating the global monitor state. Our global monitor does not keep track of which guarantees threads provide.

Our global monitor uses its internal state to determine which guarantees must be imposed on each individual thread. To determine the list of annotations that our global monitor imposes on the individual threads, we define the function $\mathit{gmon\text{-}impose} : (GMon \times (\mathbb{N}_0 \rightharpoonup Ann^\star)) \longrightarrow (\mathbb{N}_0 \rightharpoonup Ann^\star)$ by

$$\begin{array}{l}
\mathit{gmon\text{-}impose}(gmon',\Gamma) = \\
\quad \mathit{let}\ \mathit{NW} = \lambda i \in pre(gmon') : \\
\qquad\qquad \bigcup\{(gmon'(j))(\mathsf{A\text{-}NW}) \mid j \in pre(gmon') \setminus \{i\}\} \\
\qquad\ \ \mathit{NR} = \lambda i \in pre(gmon') : \\
\qquad\qquad \bigcup\{(gmon'(j))(\mathsf{A\text{-}NR}) \mid j \in pre(gmon') \setminus \{i\}\} \\
\quad \mathit{in}\ \lambda i \in pre(\Gamma) : (\Gamma(i) \restriction Asm) \\
\qquad\qquad\quad \cdot \mathit{acq}(\mathsf{G\text{-}NW}, \mathit{NW}(i)) \cdot \mathit{acq}(\mathsf{G\text{-}NR}, \mathit{NR}(i)) \\
\qquad\qquad\quad \cdot \mathit{rel}(\mathsf{G\text{-}NW}, pre(gmon') \setminus \mathit{NW}(i)) \\
\qquad\qquad\quad \cdot \mathit{rel}(\mathsf{G\text{-}NR}, pre(gmon') \setminus \mathit{NR}(i))
\end{array}$$

For each pair $(gmon',\Gamma)$ with $pre(\Gamma) \subseteq pre(gmon')$, the function $\mathit{gmon\text{-}impose}$ is well defined and returns a function with the same pre-image as Γ.

Fig. 3 presents our global monitor. It is the only rule for deriving instances of the judgment for transitions between global monitor states. In the first premise of this rule, the function $\mathit{gmon\text{-}update}$ is used to update the global monitor state based on the acquisitions and releases of assumptions in Γ. The second premise ensures that the global monitor is aware of all threads that request mode state changes. In the third premise, the function $\mathit{gmon\text{-}impose}$ is used to determine Δ, i.e., the mode state changes to be imposed on all threads that requested mode state changes. Due to the second premise of the rule, $\mathit{gmon\text{-}impose}$ is well defined for the arguments used.

$$\frac{\begin{array}{c} gmon' = \mathit{gmon\text{-}update}(gmon,\Gamma) \\ pre(\Gamma) \subseteq pre(gmon) \qquad \Delta = \mathit{gmon\text{-}impose}(gmon',\Gamma) \end{array}}{gmon \longrightarrow^{\Gamma,\Delta} gmon'}$$

Fig. 3. Transitions between global monitor states

Note that, for each $i \in pre(gmon')$ and each assumption in $gmon'(i)$, the corresponding guarantee is acquired in $(\mathit{gmon\text{-}impose}(gmon',\Gamma))(j)$ for all $j \in pre(\Gamma) \setminus \{i\}$. That is, programs do not need to explicitly contain annotations to acquire guarantees, since guarantees will be imposed on threads if needed. This means that no program analysis or human effort is required to determine the guarantees that threads provide. Annotations for assumptions, however, do need to be provided explicitly. There are practical analyses that can infer, e.g., whether a memory location is thread-local (i.e., exclusively accessed by a thread). Such analyses might be suitable building blocks to infer assumption annotations for programs.

Note also that guarantees are acquired in $(\mathit{gmon\text{-}impose}(gmon',\Gamma))(i)$ even if the ith thread is providing these guarantees already. Analogously, guarantees are released even if the ith thread is not providing them. Such unnecessary acquisitions and releases of guarantees could be avoided by letting the global monitor keep track of the guarantees that the individual threads provide. We refrain from elaborating this optimization here in more detail.

Local Monitoring of Individual Threads: Our local monitor keeps track of assumptions that a monitored thread makes and of guarantees that a thread provides. In this section, we assume that the state of a local monitor incorporates a mode state, but otherwise leave local monitor states under-specified. We use $lmon.\mathsf{mdst}$ to denote the mode state within $lmon \in LMon$, and we say that *a thread state $[com, lmon, mdst]$ is well formed* iff $lmon.\mathsf{mdst} = mdst$ holds.

We say that *a calculus for local monitor transitions properly tracks modes* iff the derivability of $lmon \longrightarrow^{\delta,\alpha}_{perm} lmon'$ implies $lmon'.\mathsf{mdst} = \mathit{update}(lmon.\mathsf{mdst},\delta)$. Moreover, we say that *a calculus for local monitor transitions enforces guarantees* iff it ensures that every well-formed thread state provides its guarantees. We leave the calculus for local monitor transitions unspecified. Such a calculus and a concrete definition of $LMon$ are provided in Section VII.

Sound Use of Modes: We say that *a global configuration $gcnf = \langle\!\langle pool, mem, \tau, gmon\rangle\!\rangle$ is well formed* iff $gmon$ is compatible with $pool$ and $pool(i)$ is a well-formed thread state for each $i \in pre(pool)$.

The following theorem states that our framework soundly enables rely-guarantee-style reasoning. This result is conditional on two assumptions about the calculus for local monitor transitions, which we discharge in Section VII (see Theorem 2).

Theorem 1. *Let $gcnf$ be a well-formed global configuration that justifies its assumptions. If the calculus for local monitor transitions properly tracks modes and enforces guarantees then $gcnf$ ensures a sound use of modes.*

Proof sketch: Well-formedness is an invariant for global configurations and justifying all assumptions is an invariant for well-formed global configurations if the calculus for local monitor transitions properly tracks modes. From these invariants, we conclude that every global configuration $gcnf'$ that is reachable from $gcnf$ is also well formed and justifies its assumptions by induction on the number of steps from $gcnf$ to $gcnf'$. From the well-formedness of $gcnf'$ and the assumption that the calculus for local monitor transitions enforces guarantees, we conclude that $gcnf'$ provides its guarantees. Hence, $gcnf$ ensures a sound use of modes. ■

VI. Example Programming Language

As an example language, we use a simple concurrent imperative language that supports multi-threading, communication between threads using shared memory, coordination between threads using barrier synchronization, and interaction between a program and its environment using channels.

Expressions: We use Exp to denote the set of expressions in our language and leave this set under-specified. We assume that the expressions are free of side effects, and use judgment $e, \mathit{mem} \Downarrow v$ to model that $e \in \mathit{Exp}$ evaluates to $v \in \mathit{Val}$ in memory state $\mathit{mem} \in \mathit{Mem}$. Function $\mathit{vars} : \mathit{Exp} \to \mathcal{P}(\mathit{Var})$ retrieves from an expression $e \in \mathit{Exp}$ a set of variables that contains all variables that the value of e might depend on. That is, for all $e \in \mathit{Exp}$ and $\mathit{mem}, \mathit{mem}' \in \mathit{Mem}$, we have

$$\begin{gathered}(\forall x \in \mathit{vars}(e) : \mathit{mem}(x) = \mathit{mem}'(x)) \\ \implies (e, \mathit{mem} \Downarrow v) \implies (e, \mathit{mem}' \Downarrow v)\end{gathered}$$

Commands: The *set of commands Com* is defined by:

$$\begin{array}{lll} \mathit{com} & ::= & x := e \mid \mathsf{skip} \mid \mathit{com}; \mathit{com} \mid \\ & & \mathsf{if}\ e\ \mathsf{then}\ \mathit{com}\ \mathsf{else}\ \mathit{com}\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od} \mid \\ & & \mathsf{input}\ \mathit{ch}\ \mathsf{to}\ x \mid \mathsf{output}\ e\ \mathsf{to}\ \mathit{ch} \mid //\gamma//\ \mathsf{barrier} \mid \\ & & \mathsf{stop} \mid \mathsf{join} \mid \mathsf{more}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od} \mid \mathsf{term} \end{array}$$

where $x \in \mathit{Var}$, $e \in \mathit{Exp}$, $\mathit{ch} \in \mathit{Ch}$, and $\gamma \in \mathit{Ann}^{\star}$. Terms of the form stop, join, $\mathsf{more}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od}$, and term capture snapshots of the control state at intermediate computation points, and are not meant to be part of the surface syntax. The sub-language without these terms is the programming language to be used by a programmer.

The behavior of assignments, skip, semicolon, conditionals, and loops is as usual. A command $\mathsf{input}\ \mathit{ch}\ \mathsf{to}\ x$ reads the next input from the channel ch into the variable x, and a command $\mathsf{output}\ e\ \mathsf{to}\ \mathit{ch}$ sends the value of the expression e on the channel ch. The command $\mathsf{barrier}$ causes a thread to block until all non-terminated threads jointly pass the barrier. Annotations that request a mode state change are placed as a comment in front of barrier commands as, e.g., in $//\epsilon{\cdot}\mathsf{acq}(\mathsf{A\text{-}NR}, x)//\ \mathsf{barrier}; \mathsf{skip}$. The control state stop models that the execution of a subprogram has completed. The control state join models that the join point of a conditional has been reached. The control state $\mathsf{more}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od}$ models that a loop with the guard e and the body com has been entered and that it will be decided next whether to execute the body or to leave the loop. Finally, the control state term models that the execution of an entire program has terminated.

Local Events: We define the *set of local events Ev* for our example language by the grammar:

$$\begin{array}{lll} \alpha & ::= & \mathsf{a}(x, e) \mid \mathsf{s} \mid \mathsf{b}(e, \mathit{com}_1, \mathit{com}_2) \mid \mathsf{join} \mid \\ & & \mathsf{enter}(e, \mathit{com}) \mid \mathsf{more}(e, \mathit{com}) \mid \mathsf{leave}(e, \mathit{com}) \mid \\ & & \mathsf{input}(x, \mathit{ch}, v) \mid \mathsf{output}(\mathit{ch}, e, v) \mid \mathsf{sync} \mid \mathsf{term} \end{array}$$

and $\chi : \mathit{Ev} \to \mathit{GEv}$, the abstraction function from local events to global events, as follows:

$$\chi(\alpha) = \begin{cases} \mathsf{sync} & \text{if } \alpha = \mathsf{sync} \\ \epsilon & \text{otherwise} \end{cases}$$

A local event $\mathsf{a}(x, e)$ models that the value of $e \in \mathit{Exp}$ is being assigned to $x \in \mathit{Var}$. The local event s models that a skip command is being executed. A local event $\mathsf{b}(e, \mathit{com}_1, \mathit{com}_2)$ models that a conditional with guard $e \in \mathit{Exp}$ and the branches com_1 and com_2 is being executed, where com_1 and com_2 are the "then" and "else" branches respectively. The local event join models that the join point of a conditional is being passed. The local event $\mathsf{enter}(e, \mathit{com})$ models that a loop $\mathsf{while}\ e\ \mathsf{do}\ \mathit{com}\ \mathsf{od}$ is being entered. A local event $\mathsf{more}(e, \mathit{com})$ models that the guard $e \in \mathit{Exp}$ of a loop with body com has evaluated to a non-zero value, and the loop body com is being entered. The local event $\mathsf{leave}(e, \mathit{com})$ models that a loop with guard $e \in \mathit{Exp}$ and with body com is being left. A local event $\mathsf{output}(\mathit{ch}, e, v)$ models that a value $v \in \mathit{Val}$ resulting from the evaluation of expression $e \in \mathit{Exp}$ is being output to channel $\mathit{ch} \in \mathit{Ch}$. A local event $\mathsf{input}(x, \mathit{ch}, v)$ models that $v \in \mathit{Val}$ is being received from $\mathit{ch} \in \mathit{Ch}$ and stored in $x \in \mathit{Var}$. The local events sync and term model that a barrier is being passed and that the thread is about to terminate, respectively.

Note that our language for local events closely resembles the syntax of our programming language, though there is no one-to-one correspondence. Note also that some of our local events capture information that goes beyond the actual next computation step. For instance, $\mathsf{b}(e, \mathit{com}_1, \mathit{com}_2)$ provides complete information about both branches of a conditional. That is, this local event captures information about the next computation steps, about computation steps that will occur sometime in the future, and about computation steps that would have occurred if the control flow were resolved differently.

Formal Semantics: Fig. 4 shows selected inference rules for the calculus that defines which transitions on command configurations are possible. The first two rules capture the execution of assignments and skip. The third rule captures the passing of a barrier. The fourth rule captures the choice of a branch in a conditional. It inserts join into the resulting control state to mark the join point. The fifth rule captures the passing of a join point. All rules are shown in the full version, including rules for sequential composition, loops, input and output, and termination of programs.

$$\frac{e, mem \Downarrow v}{(x := e, mem, \tau) \xrightarrow{\mathsf{a}(x,e),\epsilon}_{\sigma} (\mathsf{stop}, mem[x \mapsto v], \tau)} \qquad \frac{}{(\mathsf{skip}, mem, \tau) \xrightarrow{\mathsf{s},\epsilon}_{\sigma} (\mathsf{stop}, mem, \tau)} \qquad \frac{}{(//\gamma//\ \mathsf{barrier}, mem, \tau) \xrightarrow{\mathsf{sync},\gamma}_{\sigma} (\mathsf{stop}, mem, \tau)}$$

$$\frac{e, mem \Downarrow v \qquad (v \neq 0 \implies i = 1) \qquad (v = 0 \implies i = 2)}{(\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}, mem, \tau) \xrightarrow{\mathsf{b}(e,com_1,com_2),\epsilon}_{\sigma} (com_i; \mathsf{join}, mem, \tau)} \qquad \frac{}{(\mathsf{join}, mem, \tau) \xrightarrow{\mathsf{join},\epsilon}_{\sigma} (\mathsf{stop}, mem, \tau)}$$

Fig. 4. Transitions between command configurations: selected rules

The requested mode state change, i.e., the first label on the arrow, is empty (i.e., $\gamma = \epsilon$) in the conclusions of all rules except for in the rule for barriers and one of the rules for sequential composition. In the rule for barriers, the list of annotations is retrieved from the comment that precedes the barrier command. In the sequential composition rule, the list of annotations is simply propagated from the premise to the conclusion. This reflects our simplifying assumption from Section III-B, that threads request mode state changes only when synchronizing with other threads.

VII. Enforcing Information Flow Security through Local Monitoring

We present our novel hybrid approach to establish information-flow security for multi-threaded programs, building on our monitoring framework from Section V. We specialize our generic local monitor definition to a monitor that tracks and controls information flow. This specialization satisfies the requirements of Section V, modes are properly tracked and guarantees are enforced. Our solution does not require any modification of the global monitor definition from Section V.

We capture information-flow requirements by multi-level security policies and prove the soundness of our approach with respect to a knowledge-based definition of information-flow security *à la* [22]. We are able to establish such an end-to-end security property through thread-local checks by exploiting the assumptions that a thread makes about its environment. The ability to perform rely-guarantee-style reasoning about information-flow security within local monitors of individual threads is a distinctive technical feature of our approach. The practical value of this feature is that it substantially improves precision of local monitoring. Without being able to exploit assumptions, a local monitor would have to conservatively secure the guarded thread for all possible environments, resulting in severe restrictions on the behavior of threads.

The knowledge-based security definition requires that an attacker cannot distinguish a given program run from certain other hypothetical runs. To perform such counter-factual reasoning, information about other possible runs is needed within local monitors. This information is provided by the local events that are emitted during steps of a thread. For instance, the evaluation of the guard of a conditional emits a local event $\mathsf{b}(e, com_1, com_2)$, which provides information about the guard and both branches of the conditional. That is, the approach to information-flow security proposed in this section is a hybrid approach.

A. Information-Flow Security

A *security policy* is a tuple $SP = (Lev, \sqsubseteq, \sqcup, \bot)$ consisting of a set of security levels Lev, a partial order $\sqsubseteq \subseteq Lev \times Lev$, a least-upper-bound operator $\sqcup : (Lev \times Lev) \longrightarrow Lev$, and a least security level $\bot \in Lev$. A *domain assignment* is a function $chlev : Ch \to Lev$ that associates a security level with each channel. Intuitively, the security level $chlev(ch)$ of a channel ch is the upper bound on the confidentiality of information that the channel's endpoint (e.g., a user or another system) is permitted to learn. Thus, $chlev(ch)$ constitutes an upper bound on the confidentiality of information that might be received from ch and of information that may be sent over ch. When it is clear from context, we conflate channels with their security levels, and write, e.g., $ch \sqsubseteq \ell$ instead of $chlev(ch) \sqsubseteq \ell$.

Attacker Model: We assume that each attacker is associated with a security level, where an attacker at level ℓ can observe all interactions on channels ch with $ch \sqsubseteq \ell$, but cannot observe interactions on other channels. To express what an attacker at level ℓ observes during a program run, we project the trace emitted during the run to the level ℓ. We define *the projection of trace τ to security level ℓ* by

$$\begin{aligned} \tau{\downarrow}\,\ell = &\mathsf{filter}(\tau, \{\mathsf{inp}(ch, v), \mathsf{out}(ch, v) \\ &\qquad\qquad \mid ch \in Ch, chlev(ch) \sqsubseteq \ell, v \in Val\} \end{aligned}$$

That is, if τ is the trace produced by some run of a multi-threaded program then an attacker at level ℓ observes $\tau{\downarrow}\,\ell$. Based on his observations, an attacker can try to infer information about the communication strategy used. To capture an upper bound on the attacker's knowledge about which communication strategy might be in use, we define the function $\kappa : (Lev \times PSt \times Tr) \longrightarrow \mathcal{P}(\Sigma)$ by

$$\kappa(\ell, pool, \tau) = \left\{ \sigma \in \Sigma \,\middle|\, \begin{array}{l} \exists \langle\!\langle pool', mem', \tau', gmon' \rangle\!\rangle \in reach_{\sigma}(pool) : \\ \tau'{\downarrow}\,\ell = \tau{\downarrow}\,\ell \end{array} \right\}$$

The set $\kappa(\ell, pool, \tau)$ contains all strategies that are compatible with the observation $\tau{\downarrow}\,\ell$ and that thus, from the perspective of an attacker at level ℓ, might be in use. The smaller the set $\kappa(\ell, pool, \tau)$, the more accurate the attacker's knowledge. The longer the trace that the attacker observes, the more accurate is the attacker's knowledge, i.e., attacker knowledge is monotonic in the length of the trace the attacker observes.

Note that our definition of κ conservatively allows an attacker to know the program. That is, $\kappa(\ell, pool, \tau)$ is an upper bound on the knowledge of an attacker at level ℓ after this attacker observes trace τ, even if the attacker knows the program that is contained in state *pool*. However, we assume the attacker has no a-priori knowledge about which strategy is used.

Security Property: We regard strategies as confidential information. An attacker at level ℓ should not be able to distinguish two strategies that provide identical inputs at level ℓ and below when all prior interactions at level ℓ and below are identical. We capture classes of strategies that should be indistinguishable by the notion of ℓ-equivalence, defined by

$$\begin{aligned}&\sigma_1 =_\ell \sigma_2 \triangleq\\ &\forall ch \in Ch : \forall \tau_1, \tau_2 \in Tr :\\ &(ch \sqsubseteq \ell \wedge \tau_1 {\downarrow} \ell_1 = \tau_2 {\downarrow} \ell_2) \Longrightarrow \sigma_1(\tau_1, ch) = \sigma_2(\tau_2, ch)\end{aligned}$$

We use a knowledge-based definition of information-flow security, inspired by [22]. The property that we define is progress-sensitive and suitable for our model from Section III.

Definition 1. *We say that* a pool state $pool \in PSt$ is secure for a level $\ell \in Lev$ *iff*

$$\begin{aligned}&\forall \sigma \in \Sigma : \forall \langle\!\langle pool', mem', \tau', gmon' \rangle\!\rangle \in reach_\sigma(pool) :\\ &\kappa(\ell, pool, \tau') \supseteq \{\sigma' \in \Sigma \mid \sigma =_\ell \sigma'\}\end{aligned}$$

Our security property requires that if an attacker at level ℓ observes an execution starting in *pool* under strategy σ, then his knowledge must be bounded by the set of strategies that are ℓ-equivalent to σ. That is, the attacker cannot learn anything about the behavior of the actual strategy on any channel $ch \not\sqsubseteq \ell$.

B. A Specialized Local Monitor

Our local monitor is parametric in the security policy $SP = (Lev, \sqsubseteq, \sqcup, \bot)$ and in the domain assignment $chlev : Ch \to Lev$. A third parameter is a function $\mathcal{L} : Var \to Lev$ that assigns a *default security level* to each variable. These three parameters must be chosen identically for the local monitors of all threads of a multi-threaded program.

Our local monitor maintains a local copy of the mode state of the guarded thread. As a convention, we use *lmdst* as a meta-variable for such copies of a thread's mode state.

Typing Environment: Information flow into and out of a variable x is constrained by the local monitor based on the default security level $\mathcal{L}(x)$. However, if the guarded thread has exclusive write-access to x and the thread previously wrote information into x that is less confidential than $\mathcal{L}(x)$ then the local monitor can use that. Moreover, if the guarded thread has exclusive read-access to x then the local monitor may allow the thread to temporarily store information in x that is more confidential than $\mathcal{L}(x)$. The local monitor uses a typing environment Γ to track the actual security level of variables for which the guarded thread has exclusive access in some sense. Formally, a typing environment is a partial function $\Gamma : FloatVar \rightharpoonup Lev$, where $FloatVar \subseteq Var$. The variables whose security level may float might be limited, for instance, because the run-time environment accesses some variables while relying that they store information of a particular security level (e.g., variables that define thread priorities, accessed by a priority-based scheduler). The set of variables whose security level must not float is $NonFloatVar = Var \setminus FloatVar$.

We lift a typing environment $\Gamma : FloatVar \rightharpoonup Lev$ to a total function in $Var \to Lev$ by

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

Mode-State-Says Notation: To improve readability, we introduce a notation for properties of mode states. We write $mdst \rhd \mathsf{fact}$ (read "$mdst$ says fact") iff mode state $mdst$ has the property expressed by a fact from the following language

$$\mathsf{mayread}(x) \mid \mathsf{maywrite}(x) \mid \mathsf{exclusiveread}(x) \mid \\ \mathsf{exclusivewrite}(x) \mid \mathsf{othersmightread}(x) \mid \mathsf{othersmightwrite}(x)$$

with $x \in Var$. The semantics of $mdst \rhd \mathsf{fact}$ are defined by:

$$\begin{aligned}
mdst \rhd \mathsf{mayread}(x) &\triangleq x \notin mdst(\mathsf{G\text{-}NR})\\
mdst \rhd \mathsf{maywrite}(x) &\triangleq x \notin mdst(\mathsf{G\text{-}NW})\\
mdst \rhd \mathsf{exclusiveread}(x) &\triangleq x \in mdst(\mathsf{A\text{-}NR})\\
mdst \rhd \mathsf{exclusivewrite}(x) &\triangleq x \in mdst(\mathsf{A\text{-}NW})\\
mdst \rhd \mathsf{othersmightread}(x) &\triangleq x \notin mdst(\mathsf{A\text{-}NR})\\
mdst \rhd \mathsf{othersmightwrite}(x) &\triangleq x \notin mdst(\mathsf{A\text{-}NW})
\end{aligned}$$

For brevity, we write $mdst \rhd [\mathsf{fact}_1, \ldots, \mathsf{fact}_n]$ instead of $mdst \rhd \mathsf{fact}_1$, ..., $mdst \rhd \mathsf{fact}_n$, a list of mode-state-says statements concerning the same mode state. We write $mdst \rhd \mathsf{mayread}(e)$ instead of $mdst \rhd \mathsf{mayread}(x_1)$, ..., $mdst \rhd \mathsf{mayread}(x_{n'})$, where $vars(e) = \{x_1, \ldots, x_{n'}\}$.

Local Monitor States: A local monitor state is a tuple $\langle \Gamma, lmdst, \overline{pc}, \overline{br}, time, term, block \rangle$. Typing environment $\Gamma : FloatVar \rightharpoonup Lev$ tracks the actual security level of variables to which the guarded thread has exclusive access, as already described. Mode state $lmdst \in MdSt$ is the local monitor's copy of the mode state of the guarded thread. The pc stack $\overline{pc}$ and branch environment stack $\overline{br}$ summarize, respectively, the control flow decisions to reach the current program point, and the behavior of the local monitor on execution paths not taken. The pc stack is a stack of security levels, and the branch environment stack is a stack of tuples, described below. Each time the thread enters a conditional or loop, a security level that bounds the control flow decision is pushed on the pc stack, and a tuple that approximates the monitor's behavior on the branch not taken is pushed on the branch environment stack. When a conditional or loop is exited, the top element of each stack is popped.

To track information flow via internal timing, progress channels, and monitor interventions, local monitor states include timing level $time : Lev$, termination level $term : Lev$, and blocking level $block : Lev$. Termination level $term$ is an upper bound on information that influenced termination of loops prior to this point in the guarded thread's execution. The termination level only increases during thread execution. Blocking level $block$ is an upper bound on information that influenced whether the monitor blocked or allowed thread execution prior to this point in the execution. The blocking level captures information flow via monitor interventions (or lack of interventions), and only increases during execution.

Timing level time is an upper bound on information since the last synchronization that influenced *when* the guarded thread reaches its current state. The timing level is lowered after synchronization barriers, but otherwise only increases during execution. The timing level describes the information that may affect the relative timing of this guarded thread with respect to other threads and is used to prevent internal timing leaks.

When a conditional or loop is exited, the timing, termination, and blocking levels are updated to account for information flows due to execution paths that could have been taken, but weren't. For example, when a loop is exited, the termination level is increased to ensure that it is an upper bound of the information that influenced the loop guard expression, which determines how many times the loop is executed.

Calculus for Local Monitor Transitions: Selected inference rules for local monitor transitions are shown in Fig. 5. All inference rules are presented and explained in the full version.

Rule (M-Assign1) is used for an assignment $x := e$ when x is readable by other threads, according to the current mode state lmdst. Level ℓ bounds the information that might be revealed by evaluating e at this point in the execution: it is influenced by the level of the variables in e, by the decision to execute this command ($\sqcup\overline{pc}$, the join of the pc stack), the fact that the monitor did not previously block the thread (block), the fact that the thread did not previously diverge (term), and the relative timing of this thread with respect to others (time).[1] Since other threads might read x, we require that ℓ is bounded above by $\mathcal{L}(x)$, the default security level of x. Rule (M-Assign2) is similar, but applies when x cannot be read by other threads, which allows us to treat its level flow-sensitively. In both cases Γ' is computed using operator $\Gamma\langle x \mapsto_{lmdst} \ell\rangle$ (defined in Fig. 5) that returns an updated environment with the type of variable x updated to ℓ depending on mode state lmdst. Both rules require x to be writable and all variables in e to be readable according to lmdst.

Rules (M-Branch) and (M-Join) handle conditionals. Recall that monitor event $\mathsf{b}(e, com_1, com_2)$ and join are emitted, respectively, when a thread enters and exits a conditional $\mathsf{if}\ e\ \mathsf{then}\ com_1\ \mathsf{else}\ com_2\ \mathsf{fi}$. Rule (M-Branch) requires that the local monitor's mode state allows the thread to read the conditional expression: $\mathit{lmdst} \triangleright \mathsf{mayread}(e)$. It uses the static bounds oracle function $\mathsf{SB}(com, \Gamma, \mathit{lmdst}, \overline{pc}, \mathit{time}, \mathit{term}, \mathit{block})$ to approximate the behavior of the monitor on both branches. This is an on-the-fly static analysis (thus making the local monitor hybrid) needed for the soundness of information-flow tracking. Security level ℓ_{sb} returned by the oracle is an upper bound on the decision about which branch to take, and the monitor pushes it on pc stack $\overline{pc}$. The other elements returned describe, respectively, upper bounds on the timing level, termination level, blocking level, and typing environment that the local monitor would have after completing the conditional. The analysis in essence considers all possible executions of the conditional, and approximates the behavior of the guarded thread in these hypothetical executions. Level $\mathit{time}_{\mathsf{sb}}$ is an upper bound on information that affects when the conditional finishes, $\mathit{term}_{\mathsf{sb}}$ is an upper bound on information that affects whether execution of the conditional will terminate or diverge, $\mathit{block}_{\mathsf{sb}}$ is an upper bound on the information that affects whether the monitor will block the thread while executing the conditional, and Γ_{sb} describes upper bounds on the typing environment when the conditional terminates. Note that the oracle is a partial function: if the result is undefined the monitor blocks. For example, the result is undefined if a branch contains a barrier command and the branch condition is not $\bot$, since synchronizations are publicly observable and should not depend on confidential information. A full description of the static bounds oracle (including the oracle's semantic interface and an implementation) is available in the full version of the paper.

Rule (M-Join) pops the top elements of the pc stack and the static branching environment stack, and updates the variable context, timing level, termination level, and blocking level to account for potential information flows on the branch not taken.

Rule (M-Barrier-Local) regulates when a thread may synchronize. The first premise ($(\sqcup\overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} = \bot$) ensures that the decision to reach a barrier is influenced only by public information. The second premise computes the updated mode state lmdst'. The remaining premises ensure that the typing environments before and after the barrier (Γ and Γ' respectively) are appropriate based on the access this and other threads may have to variables before and after the barrier.

The other monitor rules, not presented here, are: (M-Skip) (for skip commands); (M-Input1), (M-Input2), and (M-Output) for input and output commands; (M-Enter), (M-More), and (M-Leave) for loops; and (M-Term) for terminated threads.

Theorem 2. *Our calculus for local monitor transitions properly tracks modes and enforces guarantees.*

Proof sketch: Given a derivation of $\mathit{lmon} \longrightarrow^{\delta,\alpha}_{perm} \mathit{lmon}'$, we show $\mathit{lmon}'.\mathsf{mdst} = \mathit{update}(\mathit{lmon}.\mathsf{mdst}, \delta)$. This implies that our calculus properly tracks modes. To prove that our calculus enforces guarantees, we show that every local configuration $\langle [com, \mathit{lmon}, \mathit{mdst}], \mathit{mem}, \tau\rangle$ with a well-formed thread state provides the no-write guarantee for each variable $x \in \mathit{mdst}(\mathsf{G\text{-}NW})$ and the no-read guarantee for each variable $y \in \mathit{mdst}(\mathsf{G\text{-}NR})$ by a case distinction on the local event emitted when this local configuration performs a step. ■

From Theorems 1 and 2, we obtain the following corollary.

Corollary 1. *If a global configuration is well formed and justifies its assumptions then it ensures a sound use of modes.*

Soundness of Information-Flow Control: Given a multi-threaded program $com_1 \cdot \ldots \cdot com_n \in Com^*$, we define the

[1] Other threads may modify variables in e concurrently with this thread's execution, and thus the relative timing may influence the result of evaluating e. If the guarded thread has exclusive write access to variables in e, then the second premise could be replaced by $\ell = \Gamma\langle e\rangle \sqcup (\sqcup\overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block}$, i.e., timing level time does not need to be included in the join. For simplicity, we do not provide this additional precision.

$$\textsc{M-Assign1}\ \dfrac{\begin{array}{c} \mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{othersmightread}(x)] \\ \ell = \Gamma\langle e\rangle \sqcup \mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \ell \sqsubseteq \mathcal{L}(x) \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle \end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow_{\mathit{perm}}^{\epsilon,\mathsf{a}(x,e)} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Assign2}\ \dfrac{\mathit{lmdst} \triangleright [\mathsf{maywrite}(x), \mathsf{mayread}(e), \mathsf{exclusiveread}(x)] \qquad \ell = \Gamma\langle e\rangle \sqcup \mathit{time} \sqcup (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} \qquad \Gamma' = \Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow_{\mathit{perm}}^{\epsilon,\mathsf{a}(x,e)} \langle \Gamma', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Branch}\ \dfrac{\mathit{lmdst} \triangleright \mathsf{mayread}(e) \qquad (\ell_{\mathsf{sb}}, \mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}) = \mathsf{SB}(\mathsf{if}\ e\ \mathsf{then}\ \mathit{com}_1\ \mathsf{else}\ \mathit{com}_2\ \mathsf{fi}, \Gamma, \mathit{lmdst}, \overline{pc}, \mathit{time}, \mathit{term}, \mathit{block})}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow_{\mathit{perm}}^{\epsilon,\mathsf{b}(e,\mathit{com}_1,\mathit{com}_2)} \langle \Gamma, \mathit{lmdst}, \overline{pc}\cdot\ell_{\mathsf{sb}}, \overline{br}\cdot(\mathit{time}_{\mathsf{sb}}, \mathit{term}_{\mathsf{sb}}, \mathit{block}_{\mathsf{sb}}, \Gamma_{\mathsf{sb}}), \mathit{time}, \mathit{term}, \mathit{block}\rangle}$$

$$\textsc{M-Join}\ \dfrac{\begin{array}{c} \mathit{time}'' = \mathit{time} \sqcup \mathit{time}' \sqcup \ell \qquad \mathit{term}'' = \mathit{term} \sqcup \mathit{term}' \\ \mathit{block}'' = \mathit{block} \sqcup \mathit{block}' \end{array} \qquad \Gamma'' = \lambda x. \begin{cases} \Gamma(x) \sqcup \Gamma'(x) & \text{if } x \in \mathit{pre}(\Gamma) \cap \mathit{pre}(\Gamma') \\ \Gamma(x) & \text{if } x \in \mathit{pre}(\Gamma) \setminus \mathit{pre}(\Gamma') \\ \mathsf{undef} & \text{otherwise} \end{cases}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}\cdot\ell, \overline{br}\cdot(\mathit{time}', \mathit{term}', \mathit{block}', \Gamma'), \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow_{\mathit{perm}}^{\epsilon,\mathsf{join}} \langle \Gamma'', \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}'', \mathit{term}'', \mathit{block}''\rangle}$$

$$\textsc{M-Barrier-Local}\ \dfrac{\begin{array}{c} (\sqcup \overline{pc}) \sqcup \mathit{term} \sqcup \mathit{block} = \bot \qquad \mathit{lmdst}' = \mathit{update}(\mathit{lmdst}, \delta) \\ \mathit{pre}(\Gamma') = \{x \mid x \in \mathit{FloatVar} \wedge (\mathit{lmdst}' \triangleright \mathsf{exclusiveread}(x) \vee \mathit{lmdst}' \triangleright \mathsf{exclusivewrite}(x))\} \\ (\mathit{lmdst} \triangleright \mathsf{exclusiveread}(x) \wedge \mathit{lmdst}' \triangleright \mathsf{othersmightread}(x)) \implies \Gamma(x) \sqsubseteq \mathcal{L}(x) \\ \mathit{lmdst}' \triangleright \mathsf{exclusivewrite}(x) \implies \Gamma'(x) = \Gamma\langle x\rangle \qquad (\mathit{lmdst} \triangleright \mathsf{othersmightwrite}(x) \wedge \mathit{lmdst}' \triangleright \mathsf{exclusiveread}(x)) \implies \Gamma'(x) = \Gamma\langle x\rangle \\ (\mathit{lmdst} \triangleright \mathsf{exclusivewrite}(x) \wedge \mathit{lmdst}' \triangleright [\mathsf{exclusiveread}(x), \mathsf{othersmightwrite}(x)]) \implies \Gamma'(x) = \Gamma(x) \sqcup \mathcal{L}(x) \end{array}}{\langle \Gamma, \mathit{lmdst}, \overline{pc}, \overline{br}, \mathit{time}, \mathit{term}, \mathit{block}\rangle \longrightarrow_{\mathit{perm}}^{\delta,\mathsf{sync}} \langle \Gamma', \mathit{lmdst}', \overline{pc}, \overline{br}, \bot, \bot, \bot\rangle}$$

$$\Gamma\langle x \mapsto_{\mathit{lmdst}} \ell\rangle(y) = \begin{cases} \mathsf{undef} & \text{if } y \notin \mathit{pre}(\Gamma) \\ \Gamma(y) & \text{if } y \in \mathit{pre}(\Gamma) \wedge y \neq x \\ \ell & \text{if } y \in \mathit{pre}(\Gamma) \wedge y = x \wedge \mathit{lmdst} \triangleright \mathsf{exclusivewrite}(x) \wedge x \in \mathit{FloatVar} \\ \ell \sqcup \mathcal{L}(x) & \text{if } y \in \mathit{pre}(\Gamma) \wedge y = x \wedge \mathit{lmdst} \triangleright \mathsf{othersmightwrite}(x) \wedge x \in \mathit{FloatVar} \end{cases}$$

Fig. 5. Local monitoring: selected rules

initial pool state for this program by

$$\begin{aligned} \mathit{pool}_{\mathit{com}_1\cdot\ldots\cdot\mathit{com}_n} \triangleq\ & [\mathit{com}_1, \mathit{lmon}_{\mathit{init}}, \mathit{mdst}_{\mathit{init}}] \\ & \cdot\ldots \\ & \cdot[\mathit{com}_n, \mathit{lmon}_{\mathit{init}}, \mathit{mdst}_{\mathit{init}}] \end{aligned}$$

where $\mathit{mdst}_{\mathit{init}}$ is the *inital mode state*, defined by

$$\begin{array}{ll} \mathit{mdst}_{\mathit{init}}(\mathsf{G\text{-}NR}) = \{\} & \mathit{mdst}_{\mathit{init}}(\mathsf{G\text{-}NW}) = \{\} \\ \mathit{mdst}_{\mathit{init}}(\mathsf{A\text{-}NR}) = \{\} & \mathit{mdst}_{\mathit{init}}(\mathsf{A\text{-}NW}) = \{\} \end{array}$$

and $\mathit{lmon}_{\mathit{init}}$ is the *initial local monitor state*, defined by

$$\mathit{lmon}_{\mathit{init}} \triangleq \langle \Gamma_{\mathit{init}}, \mathit{mdst}_{\mathit{init}}, \epsilon, \epsilon, \bot, \bot, \bot\rangle$$

where $\Gamma_{\mathit{init}} : \mathit{FloatVar} \rightharpoonup \mathit{Lev}$ is defined by $\mathit{pre}(\Gamma_{\mathit{init}}) = \{\}$.

Theorem 3. *If $\mathit{com}_1\cdot\ldots\cdot\mathit{com}_n \in \mathit{Com}^*$ is a multi-threaded program such that each command com_i is in the sub-language meant to be used by the programmer (as defined in Section VI) then $\mathit{pool}_{\mathit{com}_1\cdot\ldots\cdot\mathit{com}_n}$ is secure for every level $\ell \in \mathit{Lev}$.*

Proof sketch: Let global configuration $\mathit{gcnf} = \langle\!\langle \mathit{pool}_{\mathit{com}_1\cdot\ldots\cdot\mathit{com}_n}, \mathit{mem}_{\mathit{init}}, \tau_{\mathit{init}}, \mathit{gmon}_{\mathit{init},n}\rangle\!\rangle$. Since gcnf is well-formed and justifies its assumptions, we can soundly use rely-guarantee reasoning based on Definition 1 and Corollary 1. The rest of the proof is lengthy, but uses established proof techniques. It shows that for any security level ℓ, given an execution of gcnf with strategy σ that produces trace τ, and given an ℓ-equivalent strategy σ', there exists an execution of gcnf with σ' that produces trace τ' such that $\tau \downarrow \ell = \tau' \downarrow \ell$. Thus, the knowledge of an attacker at level ℓ that observes trace τ will include both σ and σ'. ■

When our monitoring framework is instantiated with the information-flow control local monitors, it accepts all the secure executions from Section II (with appropriate annotations to indicate assumptions), and correctly rejects the insecure executions (by a local monitor blocking at an appropriate point in the execution). This work presents the first hybrid progress-sensitive information-flow control monitoring framework for concurrent programs that uses fine-grained rely-guarantee reasoning about shared memory. As such, it is the only sound monitoring framework that accepts all secure execution examples from Section II. The examples, though simple, exhibit typical patterns of concurrent programs, and of programs that manipulate information of differing sensitivity.

VIII. Related Work

Static enforcement: Most existing work on information-flow security in concurrent programs uses static techniques. Volpano and Smith [23] provide a type system that enforces probabilistic noninterference in concurrent programs by providing an atomic construct, and preventing high-security loop guards. Russo and Sabelfeld [24] remove the need for the atomic construct under cooperative scheduling. Sabelfeld and Sands [25] provide a type system that ensures probabilistic noninterference for a wide class of schedulers, that also prevents high-security loop guards, and uses Agat's padding technique to prevent timing leaks [26]. Smith [27] presents a type system that reasons precisely about what information influences the timing of executions, prevents timing leaks, and thus enforces probabilistic noninterference for concurrent programs.

Andrews and Reitman [28] present an axiomatic program logic to reason about information flow in sequential and concurrent programs. They use two special "certification variables" in their logic, $\underline{local}$ and $\underline{global}$, which correspond to the pc level and the termination level respectively.

Boudol and Castellani [29] analyze the program and scheduler, and give a type system such that well-typed schedulers and threads satisfy noninterference, which Barthe and Nieto [30] verify. Barthe et al. [31] add mechanisms during compilation to enable a security-aware scheduler to enforce security.

Sabelfeld [7] considers a concurrent language with semaphores, and provides a type system that enforces security. High loops are not allowed, and padding is used to prevent timing leaks, for both branches and fork commands.

Zdancewic and Myers [17] propose that non-determinism should not be observable by low-security users (including non-determinism arising from scheduling, data races, etc.) and present a type system that enforces low-observational determinism for single memory locations. Their enforcement mechanism allows the pc level to be reset at thread synchronization points, similar to our lowering of the timing level at barrier synchronizations. Huisman et al. [32] note that the security condition may permit more information flow than intended and strengthen the condition. Terauchi [33] further improves this condition and enforces it via a fractional-capability type system, which permits updates of a thread's capabilities upon synchronization. This bears similarities to our updates of modes at synchronization points, but fractional capabilities and modes are different. For instance, in our framework a thread may have exclusive write access to a variable without exclusive read access. In contrast to all three articles, our security condition does not demand low-observable determinism.

Mantel and Sudbrock [34] present a security condition that allows nondeterminism in concurrent programs provided secret information does not influence this nondeterminism. They present a type system that enforces security for a broad class of schedulers: once a thread's timing or termination behavior is influenced by secret information, it may not interact with low-security threads. Muller and Chong [21] also permit low-observable nondeterminism via a type system for an extension of the X10 programming language.

Mantel, Sands, and Sudbrock [12] use rely-guarantee reasoning to support flow-sensitive security types in concurrent programs, thus allowing more precise enforcement of security. Our approach is inspired by theirs, but we exploit and justify rely-guarantee reasoning dynamically rather than statically.

Hybrid and dynamic enforcement: By contrast with static enforcement techniques, our approach is hybrid, combining static and dynamic techniques. This enables more precise enforcement of security, since static techniques must accept or reject a program in its entirety, whereas dynamic and hybrid techniques can accept or reject single executions.

Le Guernic [13] presents the first hybrid monitor for concurrent programs. It is flow sensitive, but uses a single monitor (and single type environment) for the entire thread pool, which restricts concurrency. To handle locks, the monitor uses static analysis to determine when a thread might require a lock, and acquires it before any high branch in that thread, thus ensuring lock acquisition does not depend on secret information. Le Guernic's monitor suppresses insecure output instead of blocking the thread. We block threads rather than modify the semantics of programs by altering or suppressing outputs.

Stefan et al. [35] present a dynamic termination-sensitive information-flow control mechanism. Their mechanism does not rely on a single global monitor but rather uses coarse-grained containers with "floating labels," where the label of the container is increased based on information read by the container, and the label restricts writes and other observable effects. In addition to preventing internal timing leaks, they mitigate external timing leaks using predictive mitigation [36]. We do not address external timing leaks, but enforce security at finer granularity (i.e., per program variable) and with greater precision (through flow-sensitivity).

IX. Conclusion

We have developed a novel framework to monitor concurrent programs, and instantiated this framework to enforce a knowledge-based progress-sensitive noninterference security condition in concurrent programs where threads share memory resources at the granularity of individual memory locations.

The framework uses a single global monitor to ensure that threads can soundly use rely-guarantee reasoning about shared memory. Each thread has its own local monitor that both enforces thread guarantees regarding shared memory, and also tracks and controls information flow within the thread. The global monitor is accessed only when threads synchronize, ensuring that the monitoring framework does not needlessly restrict concurrency. The local monitors are hybrid: they combine dynamic techniques for information-flow control with on-the-fly static program analysis to approximate information flow on untaken execution paths. Local monitor precision is improved by using rely-guarantee reasoning.

Acknowledgments

We thank the anonymous reviewers for their comments and Sylvia Grewe for discussions during early stages of this research project. The second author thanks the programming languages group at Harvard University for the inspiring environment during his sabbatical. This work is supported by the German Research Foundation (DFG) under project RSCP (MA 33326/4-2/3) in the priority program RS^3 (SPP 1496) and by the National Science Foundation under Grant No. 1054172.

References

[1] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 263–278.

[2] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proceedings of the 21st ACM Symposium on Operating System Principles*, Oct. 2007.

[3] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[4] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1999, pp. 228–241.

[5] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," 2001–2008, software release. Located at http://www.cs.cornell.edu/jif.

[6] V. Simonet, "The Flow Caml System: documentation and user's manual," Institut National de Recherche en Informatique et en Automatique (INRIA), Technical Report 0282, Jul. 2003.

[7] 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*, vol. 2244, 2002, pp. 225–239.

[8] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, 2009, pp. 352–365.

[9] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security*, 2009.

[10] ——, "Permissive dynamic information flow analysis," in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2010, pp. 3:1–3:12.

[11] ——, "Multiple facets for dynamic information flow," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2012, pp. 165–178.

[12] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium*, 2011, pp. 218–232.

[13] G. Le Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, 2007, pp. 218–232.

[14] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, Jan. 2006, pp. 79–90.

[15] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Proceedings of the 13th European Symposium on Research in Computer Security*, Oct. 2008.

[16] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, Jun. 1997, pp. 156–168.

[17] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, Jun. 2003, pp. 29–43.

[18] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[19] A. Askarov, S. Chong, and H. Mantel, "Global and local monitors to enforce noninterference in concurrent programs," School of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-02-15, 2015.

[20] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, Jun. 2006, pp. 190–201.

[21] S. Muller and S. Chong, "Towards a practical secure concurrent language," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, Oct. 2012, pp. 57–74.

[22] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *Proceedings of the 28th IEEE Symposium on Security and Privacy*, May 2007, pp. 207–221.

[23] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," in *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, 1998, pp. 34–45.

[24] A. Russo and A. Sabelfeld, "Security for multithreaded programs under cooperative scheduling," in *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, vol. 4378, 2006, pp. 474–480.

[25] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Jul. 2000, pp. 200–214.

[26] J. Agat, "Transforming out timing leaks," in *Conference Record of the Twenty-Seventh Annual ACM Symposium on Principles of Programming Languages*, Jan. 2000, pp. 40–53.

[27] G. Smith, "A new type system for secure information flow," in *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Jun. 2001, pp. 115–125.

[28] G. R. Andrews and R. P. Reitman, "An axiomatic approach to information flow in programs," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 56–76, 1980.

[29] G. Boudol and I. Castellani, "Non-interference for concurrent programs and thread systems," *Theoretical Computer Science*, vol. 281, no. 1, pp. 109–130, Jun. 2002.

[30] G. Barthe and L. P. Nieto, "Formally verifying information flow type systems for concurrent and thread systems," in *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, 2004, pp. 13–22.

[31] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld, "Security of multithreaded programs by compilation," *ACM Transactions on Information and System Security*, vol. 13, no. 3, pp. 21:1–21:32, Jul. 2010.

[32] M. Huisman, P. Worah, and K. Sunesen, "A temporal logic characterisation of observational determinism," in *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, 2006.

[33] T. Terauchi, "A type system for observational determinism," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, Jun. 2008, pp. 287–300.

[34] H. Mantel and H. Sudbrock, "Flexible scheduler-independent security," in *Proceedings of the European Symposium on Research in Computer Security*, 2010, pp. 116–133.

[35] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières, "Addressing covert termination and timing channels in concurrent information flow systems," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, June 2012.

[36] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.