Noninterference under Weak Memory Models

Heiko Mantel, Matthias Perner, Jens Sauer
TU Darmstadt, Germany, {mantel,perner,sauer}@cs.tu-darmstadt.de

Abstract—**Research on information flow security for concurrent programs usually assumes sequential consistency although modern multi-core processors often support weaker consistency guarantees. In this article, we clarify the impact that relaxations of sequential consistency have on information flow security. We consider four memory models and prove for each of them that information flow security under this model does not imply information flow security in any of the other models. This result suggests that research on security needs to pay more attention to the consistency guarantees provided by contemporary hardware. The other main technical contribution of this article is a program transformation that soundly enforces information flow security under different memory models. This program transformation is significantly less restrictive than a transformation that first establishes sequential consistency and then applies a traditional information flow analysis for concurrent programs.**

I. Introduction

Before granting a program access to private information or other secrets, one might like to know whether there is any danger that the program leaks the secrets. Research on information flow security aims at answering this question. The fact that researchers have kept making foundational contributions on information flow security [1], [2], [3], [4], [5] for more than 40 years by now, shows that information flow security is not only of practical relevance, but also a very rich domain of non-trivial research problems. Concurrency is a rich domain of foundational research problems itself. Combining it with information flow security results in intriguing new problems, such as how to achieve reliable information flow security without knowing how the scheduler works (see, e.g., [6], [7]), and further complicates problems that are already non-trivial in a sequential setting, such as how to control declassification (see, e.g., [8], [9]). In order to achieve reliable security for concurrent programs, all of these problems require solutions.

The focus of this article is on information flow security in the presence of concurrency. More concretely, we study the effects of relaxed consistency guarantees on noninterference. Weak memory models provide weaker guarantees than the sequential consistency property [10] that programmers of concurrent programs often take for granted. There are multiple benefits of relaxing sequential consistency. In particular, it enables more efficient uses of caches in multi-core processors and of program optimizations during compilation that are not compatible with sequential consistency. For an introduction to weak memory models, we refer to [11], [12].

This is not the first study of noninterference under relaxed consistency guarantees. Vaughan and Millstein studied the effects of one weak memory model, namely total store order, on noninterference [13]. They showed that noninterference under sequential consistency does not imply noninterference under total store order and that noninterference under total store order does not imply noninterference under sequential consistency. This is an insight of great significance, because it shows how closely security depends on the memory model provided by the hardware on which a program runs. Vaughan and Millstein also proposed a security type system, showed that it is sound under total store order, and demonstrated how it can be made more precise without loosing soundness.

The three main, novel contributions of this article are:

- We clarify the effects of four memory models on information flow security. For each of the models, we show that noninterference under this memory model does not imply noninterference under any of the other memory models. On the one hand, our results lift the observation by Vaughan and Millstein to further memory models. On the other hand, our results back that their observations are not just a peculiarity of one particular weak memory model. Hence, our results suggest that research on security should pay more attention to the consistency guarantees provided by modern hardware and optimizations.
- We propose a security type system and prove that it soundly verifies noninterference under four different memory models. This is the first security type system that is known to be sound for multiple weak memory models. Our type system not only checks whether programs are secure, but also transforms some potentially insecure programs into secure ones. This is the first transforming type system that is suitable for establishing noninterference under relaxed consistency guarantees. Our transformation inserts fence commands into a program such that it becomes noninterferent under weak memory models. Although inspired by fence-insertion techniques that establish sequentially consistent behavior of a program [14], our transformation does not force sequential consistency on a program executed under a weak memory model.
- We present a novel model of concurrent computation that is parametric in a set of consistency guarantees and that, hence, can be applied to different memory models. Our model of computation originated as a side product of our research project on security. Though originally a side product, we view this model itself also as a valuable contribution because it was helpful for our research on information flow security and might be helpful for others, not only in security. In contrast to the well known parametric model by Alglave [15], our model features an explicit representation of intermediate states.

To appear in Proceedings of the 27th IEEE Computer Security Foundations Symposium, Vienna, Austria, July 19-22, 2014. IEEE Computer Society.

We are confident that our results constitute a significant step towards better foundations for software security under relaxed consistency guarantees. However, the exploration of the correlation between noninterference and weak memory models has just begun. To our knowledge, this is only the second article on this correlation. Beyond the memory models that we investigate in this article, there are further memory models whose impact on information flow security needs to be clarified. Our novel model of computation under relaxed consistency guarantees could be helpful for such studies.

In Section II and III, we introduce our model of computation. We present a concurrent language that features fence commands and dynamic thread creation in Section IV, where we use our novel model of computation to define the operational semantics. In Section V, we show how to specialize our model of computation for four concrete memory models. We present our clarification of the correlation between noninterference and relaxed consistency guarantees in Section VI and our security type system in Section VII. After a discussion of related work in Section VIII, we conclude in Section IX.

Notational conventions: For a set A, we use A^n and A^* to denote the set of all n-tuples and the set of all finite lists, respectively, over A. Moreover, we use $[]$ to denote the empty list, $[a]$ to denote the list with one element a, $[a]{::}as$ to denote the list with first element a and rest as, and $as{::}as'$ to denote the result of concatenating two lists as and as'.

We denote the length of a list as by $|as|$. Given a list as and a number $i < |as|$, we write $as[i]$ to denote the i'th element in as. We also write $\mathit{last}(as)$ for the last element in as, i.e., $\mathit{last}(as) = as[|as|-1]$. Moreover, we use $as[m \dots n]$ to denote the list as' of length $n - m + 1$ with $as'[k] = as[k+m]$ for all $k \in \{0, \dots, n-m\}$. Finally, we use $as \setminus i$ to denote the list that results from as by deleting the ith element.

We use $A \to B$ and $A \rightharpoonup B$ to denote the set of all total functions and of all partial functions, respectively, with domain A and range B. For a total or partial function f with domain A and range B, we use, both $f^{-1}(B)$ and $\mathit{pre}(f)$ to denote the pre-image of f, i.e. $\mathit{pre}(f) = \{a \in A \mid f(a) \in B\}$. Moreover, we write $f[a \mapsto b]$ for the function f' with $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \in (\mathit{pre}(f) \setminus \{a\})$. Note that a function update might augment the pre-image of a partial function.

We refer to partial functions with domain $\mathbb{N}$, range A, and a finite pre-image also as *vectors over A*. That is, a partial function $\vec{a} : \mathbb{N} \rightharpoonup A$ is a *vector over A* if $|\mathit{pre}(\vec{a})| \in \mathbb{N}$ holds.

II. A Basic Model of Computation

We introduce an event-based model of computation for multi-threaded programs. Our model captures the concurrent execution of multiple threads, where each thread has access to a globally shared memory and to its own set of registers, which cannot be accessed by other threads.

We simplify our presentation by not considering dynamic thread creation, synchronization, and caching in this section. We extend our model of computation to a more sophisticated, generic model that supports caching under different memory models in Section III. In Section IV, we demonstrate how this model of computation can be used for a concurrent programming language with a spawn and a fence command.

States: We assume pair-wise disjoint sets $\mathcal{X}$, $\mathcal{R}$, and $\mathcal{V}$ of variable names, register names, and values, respectively.

We use functions in the set $\mathit{Mem} = \mathcal{X} \to \mathcal{V}$ to model states of the global memory and functions in $\mathit{Reg} = \mathcal{R} \to \mathcal{V}$ to model states of the register set, i.e., each thread's local memory. We identify threads by identifiers in $\mathcal{I} = \mathbb{N}$ and use vectors in the set $\vec{\mathit{Reg}} = \mathcal{I} \rightharpoonup \mathit{Reg}$ to model states of the registers of all threads. For a given thread identifier, a vector $\vec{\mathit{reg}} \in \vec{\mathit{Reg}}$ returns a function of type Reg that models the content of the register set of the thread with this identifier.

We model snapshots during a program run by pairs from the set $\mathit{Gst} = \vec{\mathit{Reg}} \times \mathit{Mem}$ and refer to such pairs as *global states*. We use pairs from $\mathit{Lst} = \mathit{Reg} \times \mathit{Mem}$ to capture the part of a global state that is relevant for a single thread and refer to such pairs as *local states*. We write $\mathit{gst}[i]$ for the local state of thread $i \in \mathit{pre}(\mathit{gst})$ in a global state $\mathit{gst} \in \mathit{Gst}$, i.e., if $\mathit{gst} = (\vec{\mathit{reg}}, \mathit{mem})$ then $\mathit{gst}[i] = (\vec{\mathit{reg}}(i), \mathit{mem}) \in \mathit{Lst}$. We call a thread $i \in \mathcal{I}$ *active* in gst if $i \in \mathit{pre}(\mathit{gst})$, and *inactive* otherwise. Note that $\mathit{gst}[i]$ is only defined if i is active.

As a notational convention, we use meta-variables as follows: k, m, n for natural numbers in $\mathbb{N}$, x for variables in $\mathcal{X}$, r for registers in $\mathcal{R}$, v for values in $\mathcal{V}$, i, j for thread identifiers in $\mathcal{I}$, mem for global memories in Mem, reg for local memories of a single thread in Reg, $\vec{\mathit{reg}}$ for local memories in $\vec{\mathit{Reg}}$, gst for global states in Gst, and lst for local states in Lst. We use each of these meta-variables also with indices and primes.

Events and Traces: We use *operators* to model operations that a thread can perform on its registers and *events* to model the transfer of data between memory and register sets.

We leave the set of operators Op parametric, assuming that the arity of each operator in Op is defined by a function $\mathit{arity} : \mathit{Op} \to \mathbb{N}$. We use terms of the form $\mathit{op}(\mathit{rs})$ to model the execution of the operation specified by the operator op on the register tuple $\mathit{rs} \in \mathcal{R}^{\mathit{arity}(\mathit{op})}$. We refer to such terms as *expressions* and define the set of all expressions by

$$\mathcal{E} = \{\mathit{op}(\mathit{rs}) \mid \mathit{op} \in \mathit{Op} \land \mathit{rs} \in \mathcal{R}^{\mathit{arity}(\mathit{op})}\} \ .$$

For an expression $e \in \mathcal{E}$, we use $\mathit{args}(e)$ to denote the set of all registers that appear as arguments of the operator in e.

We define the set of events Ev by the following grammar:

$$\mathit{ev} = x \leftarrow v@r \mid v@x \rightarrow r \mid v@e \circlearrowright r$$

where $e \in \mathcal{E}$. Intuitively, an event $x \leftarrow v@r$ models the copying of the value v from the register r to the variable x. Moreover, an event $v@x \rightarrow r$ models the copying of the value v from the variable x to the register r. Finally, an event $v@e \circlearrowright r$ models the updating of the register r with the value v, where the expression e captures how v was computed.

We formalize this intuition about the effects of events by a function $\mathit{effect} : \mathit{Ev} \to (\mathit{Lst} \to \mathit{Lst})$ that we define by

$$\begin{array}{rcl} \mathit{effect}(x \leftarrow v@r)(\mathit{reg}, \mathit{mem}) & = & (\mathit{reg}, \mathit{mem}[x \mapsto v]) \\ \mathit{effect}(v@x \rightarrow r)(\mathit{reg}, \mathit{mem}) & = & (\mathit{reg}[r \mapsto v], \mathit{mem}) \\ \mathit{effect}(v@e \circlearrowright r)(\mathit{reg}, \mathit{mem}) & = & (\mathit{reg}[r \mapsto v], \mathit{mem}) \ . \end{array}$$

Note that each event models the update of either a single variable or a single register. We refer to events that model the transfer of a value from the global memory into a register ($v@x \rightarrowtriangle r$) as *read events*, to events that model a register update after an operation on registers ($v@e \circlearrowleft r$) as *computation events*, and to events that model the transfer of a value from a register to the global memory ($x \leftarrowtriangle v@r$) as *write events*.

Steps by a single thread result in changes of the thread's local state. For each local state $lst = (reg, mem) \in Lst$, each global state $gst' = (\vec{reg}', mem')$, and each thread $i \in \mathcal{I}$, we define the update of gst' with lst by

$$gst'[i \mapsto lst] = (\vec{reg}'[i \mapsto reg], mem) \ .$$

We use finite lists of events to model sequences of computation steps by one thread. We refer to such lists as *traces* and define the set of all traces by $Tr = Ev^*$.

We use meta-variables as follows: op for operators in Op, rs for register tuples in $\mathcal{R}^n$, e for expressions in $\mathcal{E}$, ev for events in Ev, and tr for traces in Tr.

III. Supporting Relaxed Consistency Guarantees

Weak memory models relax sequential consistency, for instance, in order to support a more efficient use of caches in multi-core architectures. Weak memory models also provide consistency guarantees but these are weaker than sequential consistency. The various weak memory models differ in the consistency guarantees that they provide (see, e.g., [11]).

This variety of memory models is of conceptual interest and also relevant in practice. For instance, the processors Alpha, x86 and POWER support weak memory models that differ from each other [11], [16], [17].

In this section, we extend our basic model of computation from Section II to a model that supports weaker consistency guarantees than sequential consistency. This results in a novel model of computation that is generic in the sense that it can be instantiated for different memory models. Conceptually, we build on the common distinction between program-order relaxations and write-atomicity relaxations [11]. This distinction leads to a modular definition of weak memory models by sets of permitted primitive relaxations. We exploit this modularity in the construction of our model of computation.

Two key technical concepts in our model are obligations and paths. They complement events and traces as follows.

While we use events to capture computation steps and data transfers that a thread has performed, we use obligations to capture steps and transfers that have not yet happened, but for which a thread already made a commitment. For the purposes of this article, we define events and obligations to have the same granularity as commands in the considered programming language. That is, each event and obligation reflects the execution of a single command.

While we use traces to capture the order in which computation steps and data transfers have happened, we use paths to capture the order in which commitments have been made. We require each thread to commit to obligations in the order in which the corresponding commands appear in the program that the thread runs. That is, obligations must be assumed in *program order*. Under a weak memory model, the order in which a thread performs steps might differ from the order in which the thread has made commitments or, more figuratively, different traces might appear on one path.

ob	$x \leftarrowtriangle v@r$	$v@x \rightarrowtriangle r$	$?@x \rightarrowtriangle r$	$v@e \circlearrowleft r$	fe
$sources(ob)$	$\{r\}$	$\{x\}$	$\{x\}$	$args(e)$	$\emptyset$
$sinks(ob)$	$\{x\}$	$\{r\}$	$\{r\}$	$\{r\}$	$\emptyset$

Table I
Sources and sinks of an obligation

The preconditions for assuming obligations and the effects of fulfilling them are not explained here, but in Section IV.

A. Obligations, Paths and Advancing Paths

We define the set of *obligations* Ob by the grammar:

$$ob = ?@x \rightarrowtriangle r \mid ev \mid fe$$

where $ev \in Ev$ and $fe \in Fe$. The set Fe of *fences* may only contain obligations that do not involve updates of variables and registers. We leave Fe parametric in this section. In Section IV, we define a concrete set Fe that contains obligations that capture the effects of fence commands and spawn commands.

Intuitively, an obligation $?@x \rightarrowtriangle r$ models the copying of the value of variable x into register r. The question mark indicates that the value of x is not yet known. Once the value v of x has been determined, the question mark can be replaced by v, resulting in the obligation $v@x \rightarrowtriangle r$. We re-use our syntax for events to denote the corresponding obligations. That is, $v@x \rightarrowtriangle r$ is the obligation to copy v from x to r, $v@e \circlearrowleft r$ is the obligation to update r to v after an operation on registers, and $x \leftarrowtriangle v@r$ is the obligation to copy v from r to x.

Like for events, we distinguish between *read obligations*, *computation obligations*, and *write obligations*. We capture this distinction by three predicates, where $isRead(ob)$ holds if ob has the form $?@x \rightarrowtriangle r$ or $v@x \rightarrowtriangle r$, $isComp(ob)$ holds if ob has the form $v@e \circlearrowleft r$, and $isWrite(ob)$ holds if ob has the form $x \leftarrowtriangle v@r$. Moreover, we define two functions $sinks, sources : ob \to 2^{\mathcal{X} \cup \mathcal{R}}$ in Table I that, as their names indicate, retrieve the set of all registers and variables appearing as sources and sinks, respectively, in an obligation.

We record the order in which a program assumes obligations by finite lists from the set $Pa = (\mathcal{I} \times Ob)^*$ and refer to such lists as *paths*. We recursively define the *projection of a path* pa to a thread identifier $i \in \mathcal{I}$ by

$$pa\restriction_i = \begin{cases} [] & \text{, if } pa = [] \\ [ob]::(pa'\restriction_i) & \text{, if } pa = [(i, ob)]::pa' \\ pa'\restriction_i & \text{, if } pa = [(j, ob)]::pa' \text{ and } j \neq i \end{cases}$$

Note that the projection $pa\restriction_i$ reflects the order in which the thread i has assumed obligations. We lift the functions $sources$ and $sinks$ to lists of obligations by $sources([]) = \emptyset$, $sources([ob]::obs) = sources(ob) \cup sources(obs)$, $sinks([]) = \emptyset$, and $sinks([ob]::obs) = sinks(ob) \cup sinks(obs)$.

We use the events $v@x \rightarrowtriangle r$, $v@e \circlearrowleft r$, and $x \leftarrowtriangle v@r$ to record the fulfillment of the corresponding obligations. We use events of the form $v@x \rightarrowtriangle r$ also to record that an obligation of the form $?@x \rightarrowtriangle r$ has been fulfilled. An obligation $?@x \rightarrowtriangle r$ can only be fulfilled if the value of x is known. We do not record the fulfillment of obligations in Fe as they do not correspond to operations that modify registers or variables.

We use traces to record in which order obligations have been fulfilled by a single thread. To record the order in which obligations have been fulfilled by a multi-threaded program, we use *trace vectors* in $\vec{Tr} = \mathcal{I} \rightharpoonup Tr$. Note that trace vectors only capture the order between obligations that have been fulfilled by the same thread and not the order between obligations that have been fulfilled by different threads.

We use pairs from the set $APa = Pa \times \vec{Tr}$ to model snapshots during a run of a multi-threaded program and refer to elements in this set as *advancing paths*. In an advancing path $(pa, \vec{tr}) \in APa$, the path pa captures the obligations that a program has not yet fulfilled, and the trace vector $\vec{tr}$ captures the obligations that have been fulfilled so far.

As a notational convention, we use meta-variables as follows: ob for obligations in Ob, obs for lists of obligations in Ob^*, fe for events in Fe, pa for paths in Pa, $\vec{tr}$ for trace vectors in $\vec{Tr}$, and apa for advancing paths in APa.

B. Weak Memory Models

Lamport defined sequential consistency as the requirement

> "[...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [10]

As elaborated in [11], there are two aspects to sequential consistency. Firstly, the operations of each individual processor must take effect in the *program order*, i.e., the order in which operations appear in a program and, secondly, that operations of all processors must take effect in a single sequential order.

Using our concepts from Section III-A, we make these two aspects precise. Since a thread $i \in \mathcal{I}$ assumes obligations in the order in which the corresponding commands appear in the program that this thread executes, the order of obligations in the projection $pa \restriction_i$ of a path pa reflects the program order. Hence, if each thread $i \in \mathcal{I}$ fulfills its obligations in the order in which they appear in $pa \restriction_i$ for a given path pa and if obligations only cause effects when they are fulfilled, then this ensures the first aspect of sequential consistency. The second aspect of sequential consistency requires the existence of a single sequential order in which commands take effect. If there is a total order in which obligations are fulfilled by all threads and if each obligation only causes effects when it is fulfilled, then this ensures the second aspect of sequential consistency.

We capture the relaxations of these two aspects of sequential consistency with predicates. As usual, we distinguish between *program-order relaxations*, which relax the first aspect of sequential consistency, and *write-atomicity relaxations*, which relax both aspects of sequential consistency. More concretely, in terms of Section III-A, a program-order relaxation permits that, in certain cases, obligations of a thread i are fulfilled in a different order than specified by $pa \restriction_i$, while a write-atomicity relaxation permits that, in certain cases, obligations may have an effect already before they are fulfilled.

We capture each program-order relaxation by a predicate ϕ that takes a list of obligations obs and a position $k < |obs| - 1$ as arguments. Each predicate ϕ defines conditions under which the last obligation in obs may be fulfilled before an obligation that occurs at position k in obs. Note that one can use ϕ also to check whether an obligation at an arbitrary position m in obs may be fulfilled before the obligation at position $k < m$, by applying ϕ to the arguments $obs[0 \dots m]$ and k.

In order to fulfill an obligation by thread i out of order, it must be possible to re-order this obligation with all obligations that the thread has assumed before and not yet fulfilled. For a given set Φ of program-order relaxations, we formalize the condition that the last obligation in a non-empty list of obligations obs may be fulfilled next by

$$\bar{\Phi}(obs) \equiv \forall k < (|obs| - 1). \exists \phi \in \Phi. \phi(obs, k) \ .$$

Now we are ready to define under which conditions an obligation at position m in a given path pa may be fulfilled next for a given set Φ of program-order relaxations:

$$\begin{aligned} next_\Phi(pa, m) \equiv {} & \exists i \in \mathcal{I}. ob \in Ob. \\ & pa[m] = (i, ob) \wedge \bar{\Phi}(pa[0 \dots m] \restriction_i) \end{aligned}$$

Note that the thread identifier i of the thread that assumed the obligation at position m is used to project the path pa to a list of obligations and that only obligations up to position m in pa are used in this projection. Also note that $next_\Phi(pa'{::}(i, ob), m)$ holds trivially if pa' contains no obligations of thread i. That is, a thread is always permitted to fulfill its first obligation in a path.

We capture each write-atomicity relaxation by a predicate ψ that defines conditions under which a write obligation at position k in a path pa may impact an obligation at a later position $m > k$. This constitutes a relaxation of sequential consistency if the obligation at position k is not the obligation that is fulfilled next. Again, we assume that the last obligation in pa is the one that shall be influenced. Nevertheless, one can use ψ to check whether an obligation at an arbitrary position m in a path pa may be influenced by the write obligation at position k, by applying ψ to the arguments $pa[0 \dots m]$ and k.

Now we are ready to define how the unknown value in an obligation $?@x \rightarrowtriangle r$ in a path may be specialized for a given set Ψ of write-atomicity relaxations. We define a function $specialize_\Psi$ that returns the set of all events to which one may specialize an obligation ob of thread i that occurs at the end of a path $pa{::}(i, ob)$ in a global state $(\vec{reg}, mem)$. We first define the case where the obligation ob has the form $?@x \rightarrowtriangle r$:

$$\begin{aligned} & specialize_\Psi(pa, i, ?@x \rightarrowtriangle r, (\vec{reg}, mem)) = \\ & \left\{ v@x' \rightarrowtriangle r' \in Ob \;\middle|\; \begin{array}{l} x' = x \wedge r' = r \wedge x' \notin sinks(pa \restriction_i) \\ \wedge v = mem(x) \end{array} \right\} \\ & \cup \left\{ v@x' \rightarrowtriangle r' \in Ob \;\middle|\; \begin{array}{l} x' = x \wedge r' = r \wedge \\ \exists j \in \mathcal{I}. \exists r'' \in \mathcal{R}. \exists k \in \mathbb{N}. \exists \psi \in \Psi. \\ \left(\begin{array}{l} pa[k] = (j, x \leftarrowtriangle v@r'') \\ \wedge \psi(pa{::}[(i, ?@x \rightarrowtriangle r)], k) \end{array} \right) \end{array} \right\} \end{aligned}$$

$$\begin{aligned}\phi_{\mathrm{WR}}(obs,k) \equiv{}& isWrite(obs[k]) \wedge isRead(last(obs))\\ &\wedge sinks(obs[k]) \cap sources(last(obs)) = \emptyset\\ \phi_{\mathrm{WW}}(obs,k) \equiv{}& isWrite(obs[k]) \wedge isWrite(last(obs))\\ &\wedge sinks(obs[k]) \cap sinks(last(obs)) = \emptyset\\ \phi_{\mathrm{RW}}(obs,k) \equiv{}& isRead(obs[k]) \wedge isWrite(last(obs))\\ &\wedge sources(obs[k]) \cap sinks(last(obs)) = \emptyset\\ \phi_{\mathrm{RR}}(obs,k) \equiv{}& isRead(obs[k]) \wedge isRead(last(obs))\\ &\wedge sinks(obs[k]) \cap sinks(last(obs)) = \emptyset\end{aligned}$$

Figure 1. Program-order relaxations for read and write

$$\begin{aligned}\psi_{\mathrm{ROwn}}(pa,k) \equiv{}& \exists i \in \mathcal{I}.\exists x \in \mathcal{X}.\exists r,r' \in \mathcal{R}.\exists v \in \mathcal{V}.\\ & last(pa) = (i, ?@x \twoheadrightarrow r)\\ &\wedge pa[k] = (i, x \twoheadleftarrow v@r') \wedge r' \neq r\\ &\wedge x \notin sinks(pa[k+1\ldots|pa|-2]\lceil_i)\\ \phi_{\mathrm{ROwn}}(obs,k) \equiv{}& isWrite(obs[k]) \wedge isRead(last(obs))\\ &\wedge sinks(obs[k]) = sources(last(obs))\end{aligned}$$

Figure 2. Program-order relaxations for read-own write early

The first set in the above definition captures that a thread i may retrieve the value of x from the global memory mem if there are no obligations of this thread in the path pa that involve writing the variable x (i.e., if $x \notin sinks(pa\lceil_i)$ holds). The second set in the above definition captures the condition under which the thread i may retrieve the value of x from some write obligation that is pending in the path pa.

If $ob \in Ev$ or $ob \in Fe$ holds, then $specialize_\Psi$ returns the singleton set containing ob or the empty set, respectively:

$$specialize_\Psi(pa, i, ob, (\vec{reg}, mem)) = \{ob' \in Ev \mid ob' = ob\}$$

Definition 1. *A* memory model *is a pair* (Φ, Ψ) *where* Φ *and* Ψ *are sets of predicates on* $Ob^* \times \mathcal{I}$ *and* $Pa \times \mathcal{I}$*, respectively.*

Note that predicates in Φ and Ψ constitute relaxations of sequential consistency and, hence, the bigger the two sets are, the weaker is the consistency guarantee that is provided.

In Figures 1, 2, and 3, we present formal definitions of prominent program-order and write-atomicity relaxations.

In Figure 1, we define four predicates ϕ_{WR}, ϕ_{WW}, ϕ_{RW}, and ϕ_{RR} to capture conditions for re-ordering read and write operations. These predicates correspond to the program-order relaxations *Write-to-Read*, *Write-to-Write*, *Read-to-Write*, and *Read-to-Read* (see, e.g., [11]), respectively. Note that each of the four predicates requires that the obligation at the end of the list obs and the obligation at position k are of a particular type (read or write). Moreover, each of the predicates requires that modifications and observations caused by fulfilling

$$\begin{aligned}\psi^{\text{early}}_{\mathrm{ROth}}(pa,k) \equiv{}& \exists i,j \in \mathcal{I}.\exists x \in \mathcal{X}.\exists r,r' \in \mathcal{R}.\exists v \in \mathcal{V}.\\ & last(pa) = (i, ?@x \twoheadrightarrow r)\\ &\wedge pa[k] = (j, x \twoheadleftarrow v@r') \wedge j \in \text{early}(i)\\ &\wedge x \notin sinks(pa[k+1\ldots|pa|-2]\lceil_j)\end{aligned}$$

Figure 3. Program-order relaxations for read-others write early

$$\begin{aligned}\phi_{\mathrm{CR}}(obs,k) \equiv{}& isComp(obs[k]) \wedge isRead(last(obs))\\ &\wedge sinks(obs[k]) \cap sinks(last(obs)) = \emptyset\\ \phi_{\mathrm{CW}}(obs,k) \equiv{}& isComp(obs[k]) \wedge isWrite(last(obs))\end{aligned}$$

Figure 4. Program-order relaxations for computation obligations

these obligations do not interfere with each other. The latter condition ensures that these program-order relaxations do not affect the result of purely sequential computations. Program-order relaxations may only enable additional outcomes of a computation in case of a concurrent computation.

For instance, ϕ_{WR} requires that the obligation at position k in the list obs is a write obligation, and that the last obligation in obs is a read obligation. The condition $sinks(obs[k]) \cap sources(last(obs)) = \emptyset$ prevents a re-ordering if the read obligation depends on a variable that is influenced by the write obligation. This condition ensures that a write-to-read re-ordering cannot affect purely sequential computations.

In Figure 2, we define two predicates ψ_{ROwn} and ϕ_{ROwn} that together express the precondition for a *read-own-write-early relaxation* (see, e.g., [11]). The predicate ψ_{ROwn} captures for a path pa ending with a pair $(i, ?@x \twoheadrightarrow r)$ under which conditions the value of x in the obligation $?@x \twoheadrightarrow r$ of thread i may be influenced by a write obligation at position k. Namely, the write obligation at position k must be an obligation of the same thread i, the sink of this write obligation must be the same variable x, and the thread i must not have assumed further write obligations for x after position k. By permitting the earlier write obligation to influence the value of x in the later read obligation without making the update of x visible to other threads, the write becomes a non-atomic operation. The predicate ϕ_{ROwn} defines conditions under which a re-ordering of a read obligation and an earlier write obligation is permissible, if these two obligations involve the same variable. Note that ϕ_{WR} does not permit the re-ordering of two obligations that access the same variable.

In Figure 3, we define the predicate $\psi^{\text{early}}_{\mathrm{ROth}}$ that is parametric in the function early : $\mathcal{I} \rightharpoonup 2^{\mathcal{I}}$. This predicate expresses a *read-others-write-early relaxation*, where early(i) specifies the set of threads whose writes a thread i may read early. The condition $\psi^{\text{early}}_{\mathrm{ROth}}(pa,k)$ captures for a path that ends with a pair $(i, ?@x \twoheadrightarrow r)$ that the value of the variable x may be influenced by a write obligation of some thread $j \in \text{early}(i)$ at position k in pa if this is the most recently assumed write obligation of thread j for x. By permitting the write obligation to influence the read obligation without making the update of x visible to all threads, the write becomes non-atomic.

Remark 1. *Our model of computation allows one to capture computation obligations explicitly. None of the program-order and write-atomicity relaxations presented so far permit a re-ordering of computation obligations with read or write obligations. The role of computation operations in the context of weak memory models has received little attention so far. As examples, we present two speculative program-order relaxations for computation obligations in Figure 4.*

IV. A Multi-threaded Language

We introduce a concrete, multi-threaded language. Our example language comprises commands for transferring data between the shared memory and the local memory of a thread, computation commands, conditionals, and loops. Our language also provides a spawn command, which dynamically creates new threads, and a fence command, which can be used in a program to limit the effects of program-order relaxations.

The syntax of our language is defined by the grammar:

$$\begin{aligned} c = {} & \mathbf{skip}_\iota \mid \mathbf{load}_\iota\ r\ v \mid \mathbf{load}_\iota\ r\ x \mid \mathbf{store}_\iota\ x\ r \\ & \mid \mathbf{eq}_\iota\ r\ r\ r \mid \mathbf{and}_\iota\ r\ r\ r \mid \mathbf{fence}_\iota \mid \mathbf{spawn}_\iota c \\ & \mid \mathbf{if}_\iota\ r\ \mathbf{then}\ c\ \mathbf{else}\ c\ \mathbf{fi} \mid \mathbf{while}_\iota\ r\ \mathbf{do}\ c\ \mathbf{od} \mid c;\ c \end{aligned}$$

where $v \in \mathcal{V}$, $r \in \mathcal{R}$, $x \in \mathcal{X}$, and $\iota \in \mathbb{N}$. Note that each command carries a number $\iota \in \mathbb{N}$ as subscript. We assume that each subscript appears only once in a given program, such that each subscript uniquely identifies a particular occurrence of a command in the program. For instance, one could use the line number in which a command appears as subscript, given that each line contains at most one command. We use $\mathcal{C}$ to denote the set of all programs in our language.

To simplify our technical exposition in the rest of this article, we only support two commands for performing computations: the equality test "$\mathbf{eq}_\iota\ r_1\ r_2\ r_3$" and the conjunction "$\mathbf{and}_\iota\ r_1\ r_2\ r_3$". Adding further commands for computations to our language would cause no fundamental difficulties, but it would increase the length of calculi, explanations, and proofs.

Like the MEMBAR #sync instructions of SPARC [18], the fence commands in our language correspond to *full fences*. That is, the execution of a fence command is only possible if all commands that appear before the fence in program order have been executed already. Moreover, commands that appear after the fence in program order cannot be executed before the fence. Hence, fences can be used to rule out unwanted behavior by limiting the effects of program-order relaxations.

We define the operational semantics in terms of our model of computation from Sections II and III. To this end, we instantiate the set of operations by $Op = \{\text{const}, \text{eq}, \text{and}\}$ and the set of synchronization obligations by $Fe = \{\|, \nearrow_c \mid c \in \mathcal{C}\}$, where $\|$ and $\nearrow_c$ are the obligations that correspond to the commands $\mathbf{fence}_\iota$ and $\mathbf{spawn}_\iota c$, respectively.

The execution of a command is split into two steps: the assumption of an obligation and the fulfillment of this obligation. As explained in Section III-A, we use advancing paths to model snapshots during a program run. Given an advancing path $(pa, \vec{tr}) \in APa$, the assumption of an obligation ob by thread i results in the advancing path $(pa{::}[(i, ob)], \vec{tr})$. If $next_\Phi(pa, m)$ and $pa[m] = (j, ob_m)$ hold then the obligation ob_m may be fulfilled next by thread j. The fulfillment of this obligation ob_m causes the pair (j, ob_m) to be removed from pa, the obligation ob_m to be specialized to an obligation ob' by applying $specialize_\Psi$, and the effects of ob' to be propagated to the global state. If $ob' \in Ev$ holds then the fulfillment of ob' is, in addition, recorded at the end of the trace $\vec{tr}(j)$.

We use triples of the form $\langle \vec{cs}, (pa, \vec{tr}), (\vec{reg}, mem)\rangle$ to model intermediate stages of a run of a multi-threaded program

$$\frac{\begin{array}{c} i \in pre(\vec{cs}) \qquad gst = (\vec{reg}, mem) \\ \langle \vec{cs}(i), pa, \vec{reg}(i)\rangle \rightarrow_i \langle c', pa'\rangle \\ \forall n \in \{0, \ldots, |pa| - 1\}. \forall ob \in Fe. pa[n] \neq (i, ob) \end{array}}{\langle \vec{cs}, (pa, \vec{tr}), gst\rangle \Longrightarrow_{\Phi,\Psi} \langle \vec{cs}[i \mapsto c'], (pa', \vec{tr}), gst\rangle}$$

$$\frac{\begin{array}{c} next_\Phi(pa, m) \qquad pa[m] = (i, ob) \qquad ob \notin Fe \\ ob' \in specialize_\Psi(pa[0 \ldots m-1], i, ob, gst) \\ pa' = pa \setminus m \qquad \vec{tr}' = \vec{tr}[i \mapsto (\vec{tr}(i){::}[ob'])] \\ gst' = gst[i \mapsto (effect(ob', gst[i]))] \end{array}}{\langle \vec{cs}, (pa, \vec{tr}), gst\rangle \Longrightarrow_{\Phi,\Psi} \langle \vec{cs}, (pa', \vec{tr}'), gst'\rangle}$$

$$\frac{next_\Phi(pa, m) \qquad pa[m] = (i, \|) \qquad pa' = pa \setminus m}{\langle \vec{cs}, (pa, \vec{tr}), gst\rangle \Longrightarrow_{\Phi,\Psi} \langle \vec{cs}, (pa', \vec{tr}), gst\rangle}$$

$$\frac{\begin{array}{c} next_\Phi(pa, m) \qquad pa[m] = (i, \nearrow_c) \qquad pa' = pa \setminus m \\ i' = max(pre(\vec{cs})) + 1 \qquad \vec{reg}' = \vec{reg}[i' \mapsto reg] \\ \vec{cs}' = \vec{cs}[i' \mapsto c] \qquad \vec{tr}(i') = [] \qquad \forall r \in \mathcal{R}. reg(r) = 0 \end{array}}{\begin{array}{l} \langle \vec{cs}, (pa, \vec{tr}), (\vec{reg}, mem)\rangle \\ \quad \Longrightarrow_{\Phi,\Psi} \langle \vec{cs}', (pa', \vec{tr}), (\vec{reg}', mem)\rangle \end{array}}$$

Figure 5. Small steps on global configurations under (Φ, Ψ)

and refer to such triples as *global configurations*. A global configuration consists of a vector $\vec{cs} : \mathbb{N} \rightharpoonup (\mathcal{C} \cup \epsilon)$, an advancing path $(pa, \vec{tr}) \in APa$, and a global state $(\vec{reg}, mem) \in Gst$, where we use the symbol ϵ in the range of $\vec{cs}$ to model that a thread has terminated. We call a global configuration *well formed* if $\vec{cs}$, $\vec{tr}$, and $\vec{reg}$ have the same pre-image, i.e., if $pre(\vec{cs}) = pre(\vec{tr}) = pre(\vec{reg})$ holds. In the remainder of this article, all relevant global configurations will be well formed.

To capture small steps on global configurations under a memory model (Φ, Ψ), we introduce the judgment

$$\langle \vec{cs}, apa, gst\rangle \Longrightarrow_{\Phi,\Psi} \langle \vec{cs}', apa', gst'\rangle$$

The calculus for deriving this judgment is depicted in Figure 5.

The first rule in Figure 5 captures how commands are processed and how obligations are assumed. The first premise ensures that i is an active thread. In the third premise, the judgment $\langle \vec{cs}(i), pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$ is used to capture the processing of the next command in $\vec{cs}(i)$ by thread i, where c' is the command that remains to be executed by thread i and pa' is the resulting path. This judgment will be defined later in this section such that the path pa' either equals pa or equals $pa{::}[(i, ob)]$ for some obligation ob. The fourth premise ensures that a thread cannot assume new obligations if a fence obligation of this thread is pending.

The second rule in Figure 5 captures how threads fulfill obligations other than $\|$ and $\nearrow_c$. The first two premises of the rule ensure that the obligation ob of thread i at position m may be fulfilled next. In the fourth premise, ob is specialized to ob'. Recall from Section III-B, that $specialize_\Psi$ returns for an obligation $?@x \twoheadrightarrow r$ the set of all instantiations that are possible under the write-atomicity relaxations in Ψ. Otherwise, $specialize_\Psi$ returns the singleton set containing the given obligation. The last three premises remove the obligation ob from the path, append the event ob' to the trace of thread i, and update the global state according to the effect of ob'.

The third rule captures how a thread fulfills an obligation $\|$, which the thread assumed due to a fence command. A fence command prevents re-orderings across this command, hence, its name. In our operational semantics, this is realized by the combination of the fourth premise of the first rule in Figure 5, the premise $ob \notin Fe$ of the second rule in Figure 5, and the fact that if $pa(m) = (i, \|)$ and $next_\Phi(pa, m)$ hold, then pa does not contain an obligations of thread i before position m.

The last rule in Figure 5 captures how a thread fulfills an obligation $\nearrow_c$, which the thread assumed due to a spawn command. This rule models the creation of a new thread with a new identifier i' by enlarging the pre-image of $\vec{cs}$, $\vec{tr}$, and $\vec{reg}$ by i'. Like the obligation $\|$, the obligation $\nearrow_c$ also cannot be re-ordered with other obligations, for the same reasons.

We formalize how a thread processes a command in terms of the command's immediate effects on the local memory, i.e. the registers of this thread, and in terms of an obligation that the thread assumes. We use the judgment $\langle c, pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$ to capture that if thread i processes the command c in the context of a path pa, and the current local memory reg, then, afterwards, the path is pa' and either a command $c' \in \mathcal{C}$ remains to be executed or the thread has terminated (indicated by $c' = \epsilon$). The calculus for this judgment is depicted in Figure 6. The rules for $\mathbf{skip}_\iota$, conditionals and loops, leave the path unchanged. All other rules add a pair (i, ob) to the path to indicate that thread i has assumed the obligation ob. The particular obligation differs in the rules. Moreover, the rules for all commands that use values from registers, require that the path pa does not contain any unfulfilled obligations of thread i that might influence these registers. The reason for these premises in the rules is that values of registers are inserted into an obligation when the obligation is assumed, and this would be incorrect if updates to these registers were still pending. Otherwise, the rules in Figure 6 are straightforward.

We use the judgment $\langle c, mem\rangle \Downarrow_{(\Phi,\Psi)} mem'$ to model that a run of program c starting in an initial memory mem terminates with final memory mem'. The only rule for this judgment is depicted in Figure 7, where we use $\Longrightarrow^*_{(\Phi,\Psi)}$ to denote the transitive closure of the relation induced by the judgment for small steps on global configurations. The premises of the rule ensure that all registers are initialized with value 0 and that the program run starts in a well-formed global configuration. That the program, indeed, terminated is captured by the two premises $pa' = []$ and $\forall i \in pre(\vec{cs}').\vec{cs}'(i) = \epsilon$.

V. Example Memory Models

We are now ready to formalize four examples of concrete memory models. The example memory models SC, IBM370, TSO, and PSO that we present correspond to *sequential consistency*, `IBM 370`, *total store order*, and *partial store order*, respectively. Each of these memory models had relevance in processor design: partial store order, total store order, and

$$\frac{}{\langle \mathbf{skip}_\iota, pa, reg\rangle \rightarrow_i \langle \epsilon, pa\rangle}$$

$$\frac{}{\langle \mathbf{fence}_\iota, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, \|)]\rangle}$$

$$\frac{ob = v@\mathrm{const} \circlearrowright r}{\langle \mathbf{load}_\iota\ r\ v, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, ob)]\rangle}$$

$$\frac{ob = ?@x \rightarrowtriangle r}{\langle \mathbf{load}_\iota\ r\ x, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, ob)]\rangle}$$

$$\frac{ob = x \leftarrowtriangle v@r \qquad v = reg(r) \qquad r \notin sinks(pa\restriction_i)}{\langle \mathbf{store}_\iota\ x\ r, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, ob)]\rangle}$$

$$\frac{\begin{array}{c} ob = 1@\mathrm{eq}(r_2, r_3) \circlearrowright r_1 \\ reg(r_2) = reg(r_3) \qquad r_2, r_3 \notin sinks(pa\restriction_i)\end{array}}{\langle \mathbf{eq}_\iota\ r_1\ r_2\ r_3, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, ob)]\rangle}$$

$$\frac{\begin{array}{c} ob = 0@\mathrm{eq}(r_2, r_3) \circlearrowright r_1 \\ reg(r_2) \neq reg(r_3) \qquad r_2, r_3 \notin sinks(pa\restriction_i)\end{array}}{\langle \mathbf{eq}_\iota\ r_1\ r_2\ r_3, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, ob)]\rangle}$$

$$\frac{\begin{array}{c} ob = 1@\mathrm{and}(r_2, r_3) \circlearrowright r_1 \\ reg(r_2) \neq 0 \qquad reg(r_3) \neq 0 \qquad r_2, r_3 \notin sinks(pa\restriction_i)\end{array}}{\langle \mathbf{and}_\iota\ r_1\ r_2\ r_3, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, ob)]\rangle}$$

$$\frac{\begin{array}{c} ob = 0@\mathrm{and}(r_2, r_3) \circlearrowright r_1 \\ reg(r_2) = 0 \vee reg(r_3) = 0 \qquad r_2, r_3 \notin sinks(pa\restriction_i)\end{array}}{\langle \mathbf{and}_\iota\ r_1\ r_2\ r_3, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, ob)]\rangle}$$

$$\frac{reg(r) \neq 0 \qquad r \notin sinks(pa\restriction_i)}{\langle \mathbf{if}_\iota\ r\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \mathbf{fi}, pa, reg\rangle \rightarrow_i \langle c, pa\rangle}$$

$$\frac{reg(r) = 0 \qquad r \notin sinks(pa\restriction_i)}{\langle \mathbf{if}_\iota\ r\ \mathbf{then}\ c\ \mathbf{else}\ c'\ \mathbf{fi}, pa, reg\rangle \rightarrow_i \langle c', pa\rangle}$$

$$\frac{reg(r) \neq 0 \qquad r \notin sinks(pa\restriction_i)}{\begin{array}{c}\langle \mathbf{while}_\iota\ r\ \mathbf{do}\ c\ \mathbf{od}, pa, reg\rangle \\ \rightarrow_i \langle c;\ \mathbf{while}_\iota\ r\ \mathbf{do}\ c\ \mathbf{od}, pa\rangle\end{array}}$$

$$\frac{reg(r) = 0 \qquad r \notin sinks(pa\restriction_i)}{\langle \mathbf{while}_\iota\ r\ \mathbf{do}\ c\ \mathbf{od}, pa, reg\rangle \rightarrow_i \langle \epsilon, pa\rangle}$$

$$\frac{}{\langle \mathbf{spawn}_\iota c, pa, reg\rangle \rightarrow_i \langle \epsilon, pa{::}[(i, \nearrow_c)]\rangle}$$

$$\frac{\langle c, pa, reg\rangle \rightarrow_i \langle \epsilon, pa'\rangle}{\langle c;\ c', pa, reg\rangle \rightarrow_i \langle c', pa'\rangle}$$

$$\frac{\langle c, pa, reg\rangle \rightarrow_i \langle c'', pa'\rangle \qquad c'' \in \mathcal{C}}{\langle c;\ c', pa, reg\rangle \rightarrow_i \langle c'';\ c', pa'\rangle}$$

Figure 6. Processing a command and augmenting a path

$$\frac{\begin{array}{c} pre(\vec{cs}) = pre(\vec{tr}) = pre(\vec{reg}) = \{0\} \\ \vec{cs}(0) = c \qquad \vec{tr}(0) = pa = [] = pa' \\ \forall r \in \mathcal{R}.\vec{reg}(0)(r) = 0 \qquad \forall i \in pre(\vec{cs}').\vec{cs}'(i) = \epsilon \\ \langle \vec{cs}, (pa, \vec{tr}), (\vec{reg}, mem)\rangle \\ \Longrightarrow^*_{(\Phi,\Psi)} \langle \vec{cs}', (pa', \vec{tr}'), (\vec{reg}', mem')\rangle \end{array}}{\langle c, mem\rangle \Downarrow_{(\Phi,\Psi)} mem'}$$

Figure 7. Big-step semantics for commands

`IBM 370` are supported by SPARC, modern x86 processors, and in the processor `IBM 370`, respectively [11], [16].

Following Definition 1, we define each memory model as a pair (Φ, Ψ) of two sets of predicates that capture the permitted program-order relaxations and write-atomicity relaxations.

Definition 2. *For each $MM \in \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$, we define $MM = (\Phi_{MM}, \Psi_{MM})$ by the following table:*

MM	Φ_{MM}	Ψ_{MM}
SC	$\emptyset$	$\emptyset$
IBM370	$\{\phi_{\mathrm{WR}}\}$	$\emptyset$
TSO	$\{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}\}$	$\{\psi_{\mathrm{ROwn}}\}$
PSO	$\{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}, \phi_{\mathrm{WW}}\}$	$\{\psi_{\mathrm{ROwn}}\}$

Let $\mathcal{MM} = \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$ for the rest of this article. For each of the four models in $\mathcal{MM}$, our choice of relaxations of sequential consistency in Definition 2 is equivalent to the one in [11]. In the following, we argue for the adequacy of Definition 2 further by relating our definitions to the original definitions of these memory models.

Proposition 1. *The memory model* $\mathrm{SC} = (\Phi_{\mathrm{SC}}, \Psi_{\mathrm{SC}})$ *imposes the same constraints as sequential consistency [10].*

Proof sketch: As explained in Section III-B, sequential consistency is equivalent to the conjunction of two requirements: firstly, the commands of each thread must take effect in program order and, secondly, the commands of all threads must take effect in a single sequential order. These two requirements are satisfied by our memory model $\mathrm{SC} = (\Phi_{\mathrm{SC}}, \Psi_{\mathrm{SC}})$, where $\Phi_{\mathrm{SC}} = \emptyset$ and $\Psi_{\mathrm{SC}} = \emptyset$ according to Definition 2.

The first rule in Figure 5 and the rules for sequential composition in Figure 6 together ensure that obligations are assumed in program order. If $next_{\Phi}(pa, m)$, $pa[m] = (i, ob)$, and $\Phi = \Phi_{\mathrm{SC}}$ hold then $pa[0 \ldots m-1] \restriction_i = []$ holds according to the definition of $next$ because $\Phi_{\mathrm{SC}} = \emptyset$. Since $next_{\Phi}(pa, m)$ is a premise in the second, third, and fourth rule in Figure 5, obligations are fulfilled by a thread in the order in which they have been assumed by this thread. That is, commands of each thread take effect in program order.

Moreover, if $ob' \in specialize_{\Psi}(pa[0 \ldots m-1], i, ob, gst)$ and $\Psi = \Psi_{\mathrm{SC}}$ then, according to the definition of $specialize$, either $ob' = ob$ holds or ob' results from instantiating ob with a value retrieved directly from global memory. That is, obligations can only have an effect on variables, registers, and other obligations when they are fulfilled. Hence, obligations of all threads take effect in one single order. ■

Proposition 2. *The memory model* $\mathrm{IBM370} = (\Phi_{\mathrm{IBM370}}, \Psi_{\mathrm{IBM370}})$ *imposes the same constraints as the memory model* `IBM 370` *defined in [19].*

Proof sketch: As described in [11] the memory model `IBM 370` [19] relaxes the requirement that commands of each thread must take effect in program order. More concretely, a read command of a thread may take effect before a write command of the same thread against program order if the two commands are not conflicting, i.e., the source in the read must differ from the sink in the write. This is the only relaxation of sequential consistency that the memory model `IBM 370` permits. These requirements are also satisfied by our memory model IBM370 (recall $\Phi_{\mathrm{IBM370}} = \{\phi_{\mathrm{WR}}\}$ and $\Psi_{\mathrm{IBM370}} = \emptyset$).

The first rule in Figure 5 and the rules for sequential composition in Figure 6 ensure that obligations are assumed in program order. If $next_{\Phi}(pa, m)$, $pa[m] = (i, ob)$, $\Phi = \Phi_{\mathrm{IBM370}}$, and $pa[0, \ldots, m-1] \restriction_i = obs$ with $obs \neq []$ then, according to the definitions of $next$ and ϕ_{WR}, this implies that $isRead(ob)$, $isWrite(obs[k])$ for each $k \in \{0 \ldots |obs| - 1\}$, and $sources(ob) \cap sinks(obs) = \emptyset$ hold. That is, read obligations of a thread can be fulfilled against program order before write obligations of the same thread if the sinks in these write obligations are disjoint from the source in the read obligation. All other obligations must be fulfilled in program order.

Since $\Psi_{\mathrm{IBM370}} = \emptyset$, obligations can have an effect on variables, registers, and other obligations only when they are fulfilled. The argument is identical to the one for SC. Hence, obligations of all threads take effect in one single order. ■

In [20], the memory models total store order and partial store order are defined by lists of axioms. These axioms impose constraints on the order of load and store operations wrt. a memory order $\leq$ and a program order $;^i$ for each processor i. The concepts underlying our parametric model of execution correspond to the ones in [20] as follows: our commands correspond to the operations in [20], our thread identifiers correspond to the processor identifiers in [20], a pair $(i, v@x \twoheadrightarrow r)$ in our model corresponds to a load operation L^i_x in [20], and a pair $(i, x \twoheadleftarrow v@r)$ in our model corresponds to a store operation S^i_x in [20], where v equals the value of the memory location accessed by the load and store operations, i.e., $Val[S^i_x]$ and $Val[L^i_x]$ in [20]. For arguing that our definitions of TSO and PSO satisfy the respective axioms in [20], we need to define the orders $\leq$ and $;^i$. For a given derivation of the judgment $\langle c, mem \rangle \Downarrow_{(\Phi,\Psi)} mem'$, we define $;^i$ as the order in which thread i assumes obligations in the derivation (by application of the 1st rule in Figure 5) and $\leq$ as the order in which obligations are fulfilled in the derivations (by applications of the 2nd, 3rd, and 4th rule in Figure 5).

Proposition 3. *The memory model* $\mathrm{TSO} = (\Phi_{\mathrm{TSO}}, \Psi_{\mathrm{TSO}})$ *imposes the same constraints as total store order in [20].*

Proof sketch: The memory model total store order is defined in [20] by six axioms: Order, Atomicity, Termination, Value, LoadOp, and StoreStore.

We first argue that TSO is no more restrictive than total store order. Total store order, as defined in [20], permits load operations to appear in the memory order $\leq$ before store operations against the program order $;^i$. If $pa[m] = (i, ob)$, $\Phi = \Phi_{\mathrm{TSO}}$, $pa[0, \ldots, m-1] \restriction_i = obs$ with $obs \neq []$, $isRead(ob)$, and $isWrite(obs[k])$ for each $k \in \{0 \ldots |obs| - 1\}$ then this implies $next_{\Phi}(pa, m)$ according to the definitions of $next$, ϕ_{WR}, and ϕ_{ROwn}. Since only the premise $next_{\Phi}(pa, m)$ could forbid a read obligation to be fulfilled using the second rule in Figure 5, TSO allows read obligations of a thread to be fulfilled before write obligations of the same thread against the program order like total store order in [20].

We now argue that TSO is no more permissive than total store order. The axioms Atomicity and Termination in [20] impose constraints on atomic load-store operations and on diverging runs. These two axioms are trivially fulfilled in the context of this article because our example language does not incorporate atomic load-store operations and the judgment $\langle c, mem\rangle \Downarrow_{(\Phi,\Psi)} mem'$ captures only terminating runs. The other four axioms are also fulfilled by our memory model TSO (recall $\Phi_{\mathrm{TSO}} = \{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}\}$ and $\Psi_{\mathrm{TSO}} = \{\psi_{\mathrm{ROwn}}\}$). The axiom Order in [20] requires all store and flush operations to be totally ordered by $\leq$. Our example language does not incorporate flush operations, but fence commands instead. The order $\leq$ induced by a derivation of $\langle c, mem\rangle \Downarrow_{(\Phi,\Psi)} mem'$ is a total order on write and fence obligations. The axiom Value requires that the value of a load from a location x is the value written by a most recent store to this location:

$$\begin{aligned}&\mathtt{Val}[L_x^i] = \\ &\mathtt{Val}[S_x^j \mid S_x^j = \mathtt{Max}_{\leq}[\{S_x^{j'} \mid S_x^{j'} \leq L_x^i\} \cup \{S_x^i \mid S_x^i; L_x^i\}]] \ .\end{aligned}$$

The two possible cases for determining the value of the load in this equation correspond to the two cases in our definition of *specialize*. In the first case, the value of the variable is directly retrieved from the global memory (and, hence, this value equals the value stored by the write obligation that was fulfilled last for this location) and, in the second case, the value is retrieved from the most recently assumed write obligation of the same thread that has not yet been fulfilled. The axiom LoadOp requires for each operation after a given load operation in the program order of a processor i, that this operation takes effect after the load operation. Since $\Phi_{\mathrm{TSO}} = \{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}\}$, TSO does not allow an obligation to be fulfilled before a given read obligation against the program order according to the definitions of ϕ_{WR} and ϕ_{ROwn}. The axiom StoreStore requires that store and flush operations take effect in the relative order in which they occur in the program order. Again, our definition of TSO does not allow that write or fence obligations are fulfilled against the program order according to the definitions of ϕ_{WR} and ϕ_{ROwn}. ■

Proposition 4. *The memory model* $\mathrm{PSO} = (\Phi_{\mathrm{PSO}}, \Psi_{\mathrm{PSO}})$ *imposes the same constraints as partial store order in [20].*

Proof sketch: The memory model partial store order is defined in [20] by seven axioms: Order, Atomicity, Termination, Value, LoadOp, StoreStore, and StoreEq.

We first argue that PSO is no more restrictive than partial store order. Partial store order, as defined in [20], permits load operations to appear in the memory order $\leq$ before store operations against the program order $;^i$. Moreover, two store operations may appear in the memory order $\leq$ against the program order $;^i$ if these store operations have different sinks. If $pa[m] = (i, ob)$, $\Phi = \Phi_{\mathrm{PSO}}$, $pa[0,\ldots,m-1]\lceil_i = obs$ with $obs \neq []$, $isWrite(obs[k])$ for each $k \in \{0 \ldots |obs|-1\}$ and

- either $isRead(ob)$
- or $isWrite(ob)$ and $sinks(ob) \cap sinks(obs) = \emptyset$

then this implies $next_\Phi(pa, m)$ according to the definitions of $next$ and of Φ_{PSO}, i.e. $\{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}, \phi_{\mathrm{WW}}\}$. Since only the premise $next_\Phi(pa, m)$ could forbid a read or write obligation to be fulfilled using the second rule in Figure 5, PSO allows read as well as write obligations of a thread to be fulfilled before write obligations of the same thread against the program order like partial store order in [20].

We now argue that PSO is no more permissive than partial store order. Again, the axioms Atomicity and Termination in [20] are trivially fulfilled. The axioms Order, Value, and LoadOp are the same ones as for total store order. These three axioms are fulfilled by the memory model PSO (recall $\Phi_{\mathrm{PSO}} = \{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}, \phi_{\mathrm{WW}}\}$ and $\Psi_{\mathrm{PSO}} = \{\psi_{\mathrm{ROwn}}\}$). The argument for the fulfillment of these three axioms is analogous to the one for TSO. The additional program order relaxation ϕ_{WW} might affect the order in which write obligations are fulfilled, but there still exists a total order in which write obligations are fulfilled. Due to the relaxation ϕ_{WW}, two write obligations of the same thread might be fulfilled against the program order. However, this is only permitted if the sinks of these write obligations differ, and, hence, the fulfillment of axiom Value is not affected. Finally, ϕ_{WW} does not permit any re-ordering wrt. the fulfillment of read obligations and, hence, the axiom LoadOp is fulfilled. The axiom StoreStore in the partial store order model is different from the axiom with this name in the total store order model. For partial store order, the axiom StoreStore requires that store operations that are separated by a STBAR operation take effect in program order. In our language, fence commands take the role of STBAR operations. Since Φ_{PSO} does not permit to re-order fence obligations, this variant of the axiom StoreStore is fulfilled. The axiom StoreStoreEq requires that store operations with the same sink take effect in program order. The condition $sinks(obs[k]) \cap sinks(last(obs)) = \emptyset$ in the definition of ϕ_{WW} ensures that write obligations can, indeed, only be re-ordered if they have disjoint sinks. ■

VI. Information Flow Security

The novel model of computation and its instantiation with a concrete language, which we described in Sections II–V, originated as a side-product of studying the impact of different memory models on information flow security. Two crucial features of our model of computation are its operational flavor and that it can be instantiated with different memory models. These features provide the basis for comparing the effects of different memory models on noninterference.

The main result in this section is Theorem 1. This theorem clarifies the effects of the four memory models from Definition 2 (i.e., PSO, TSO, IBM370, and SC) on noninterference. The formulation of the theorem is crisp, but proving it, was not an easy exercise. We describe the three-step proof technique that we employed as it might be interesting itself.

A. Noninterference under Weak Memory Models

We consider a termination-sensitive definition for a two-level security lattice, where the intuitive requirement is that information must not flow from the level **High** to **Low**.

We use a function $lev : \mathcal{X} \to \{\mathbf{Low}, \mathbf{High}\}$ to associate each variable in a program with one of these security levels. As usual, we assume the initial values of each variable $x \in \mathcal{X}$ with $lev(x) = \mathbf{High}$ to be secret, and the initial and final values of each $x' \in \mathcal{X}$ with $lev(x') = \mathbf{Low}$ to be public.

We define two global memories $mem, mem' \in Mem$ to be **Low**-*equal* if $lev(x) = \mathbf{Low} \implies mem(x) = mem'(x)$ holds for each $x \in \mathcal{X}$ and denote this fact by $mem =_L mem'$. This means, if $mem =_L mem'$ holds then two global memories $mem, mem' \in Mem$ differ only in secrets.

Definition 3. *A program $c \in \mathcal{C}$ is MM-*noninterfering *(denoted by $c \in NI_{MM}$), if the following condition holds:*

$$\forall mem_0, mem_1, mem'_0 \in Mem. \left(\begin{array}{l} (mem_0 =_L mem'_0 \wedge \langle c, mem_0\rangle \Downarrow_{MM} mem_1) \\ \implies \exists mem'_1 \in Mem. \\ \qquad (mem_1 =_L mem'_1 \wedge \langle c, mem'_0\rangle \Downarrow_{MM} mem'_1) \end{array} \right)$$

Note that, if $c \in NI_{MM}$ holds and c is run under the memory model MM in two initial memories that are **Low**-equal, then, after the runs of c terminate, the resulting memories are also **Low**-equal. This means that the final value of each variable $x' \in \mathcal{X}$ with $lev(x') = \mathbf{Low}$ is independent of the initial values of all variables $x \in \mathcal{X}$ with $lev(x) = \mathbf{High}$. In other words, running c cannot leak secret information.

Theorem 1. *Noninterference under MM does not imply noninterference under MM', for each pair of distinct memory models $MM, MM' \in \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$.*

In total, Theorem 1 expresses twelve non-implications for noninterference under different memory models, including the two non-implications that were proven by Vaughan and Millstein in [13]. Vaughan and Millstein showed that noninterference under sequential consistency does not imply noninterference under total store order and vice versa. Our theorem demonstrates that this observation is not just a peculiarity of one specific memory model. Beyond this, our theorem clarifies the relationship between different weak memory models.

B. Proof Sketch

For the proof of Theorem 1, we developed a three-step proof technique that we find interesting in itself.

In the first step, we define three pairs of conditions on memory models, namely (γ_1, δ_1), (γ_2, δ_2), and (γ_3, δ_3) such that the two conditions within each pair are contradictory. That is $(\neg\gamma_l) \vee (\neg\delta_l)$ holds for each $l \in \{1, 2, 3\}$.

In the second step, we show that the three pairs of conditions can be used to discriminate between any two memory models in $\mathcal{MM}$. We show for all $MM, MM' \in \mathcal{MM}$ that there exists $l \in \{1, 2, 3\}$ such that either $\gamma_l(MM) \wedge \delta_l(MM')$ or $\gamma_l(MM') \wedge \delta_l(MM)$ holds. Note that at most one of the two conditions can be true because γ_l and δ_l are contradictory. Hence, (γ_l, δ_l) discriminates between MM and MM'.

In the third step, we specify for each index $l \in \{1, 2, 3\}$ two programs $c_l^+, c_l^- \in \mathcal{C}$ and show that the following four implications hold for each $MM \in \mathcal{MM}$:

$$\begin{array}{ll} \gamma_l(MM) \implies c_l^+ \in NI_{MM} & \delta_l(MM) \implies c_l^+ \notin NI_{MM} \quad (1) \\ \gamma_l(MM) \implies c_l^- \notin NI_{MM} & \delta_l(MM) \implies c_l^- \in NI_{MM} \end{array}$$

The combination of these three steps allows one to conclude that, for each pair of two distinct memory models $(MM, MM') \in \mathcal{MM} \times \mathcal{MM}$, there exists a program $c \in \mathcal{C}$ such that $c \in NI_{MM}$ holds and $c \in NI_{MM'}$ does not hold. This proposition is equivalent to the one in Theorem 1.

Before developing this proof technique, we started to prove Theorem 1 by providing two programs c, c' for each pair of memory models MM, MM' and by showing that the programs are discriminating for these memory models, i.e., that

$$c \in NI_{MM} \wedge c \notin NI_{MM'} \text{ and } c' \in NI_{MM'} \wedge c' \notin NI_{MM}$$

hold. This led to proofs that shared many similarities. Our proof technique can be viewed as a systematic solution to factor out these similarities, thus reducing both, the size of proofs and the effort to construct them. Our proof technique can also be viewed as a systematic solutions to uniformly structure proofs that pairs of programs are discriminating. Finally, we found the conditions (γ_l, δ_l) helpful for finding pairs of discriminating programs. However, creativity is also needed with our proof technique, namely to determine suitable pairs of discriminating conditions (γ_l, δ_l) and to determine, for each of these pairs, a suitable pair of programs c_l^+, c_l^-.

In the remainder of this section, we elaborate the three steps in more detail. In particular, we provide formal definitions of the three pairs of conditions, show that they discriminate the memory models in $\mathcal{MM}$, and present, for each pair of conditions, two programs that fulfill the implications in (1).

We define the conditions γ_1, γ_2, and γ_3:

$$\begin{array}{rcl} \gamma_1(\Phi, \Psi) & \equiv & \phi_{\mathrm{WR}} \in \Phi \\ \gamma_2(\Phi, \Psi) & \equiv & \phi_{\mathrm{WR}} \in \Phi \wedge \phi_{\mathrm{ROwn}} \in \Phi \\ \gamma_3(\Phi, \Psi) & \equiv & \phi_{\mathrm{WW}} \in \Phi \end{array}$$

We define the conditions δ_1, δ_2, and δ_3 in Figure 8. With our definitions of (γ_1, δ_1), (γ_2, δ_2), and (γ_3, δ_3), the disjunction $(\neg\gamma_l) \vee (\neg\delta_l)$ holds for each $l \in \{1, 2, 3\}$. That is, the two conditions within each pair are contradictory.

Here, we show this for $l = 1$ only, the other two cases are similar: We assume that both $\gamma_1(\Phi, \Psi)$ and $\delta_1(\Phi, \Psi)$ hold, and derive a contradiction. We consider the path $pa = [(0, \mathtt{x} \leftarrow 0@\mathtt{r}_1)]{::}[(0, 0@\mathtt{y} \twoheadrightarrow \mathtt{r}_2)]$. For this path, $\phi_{\mathrm{WR}}(pa, 1)$ holds, because $isWrite(\mathtt{x} \leftarrow 0@\mathtt{r}_1)$, $isRead(0@\mathtt{y} \twoheadrightarrow \mathtt{r}_2)$, and $sinks(\mathtt{x} \leftarrow 0@\mathtt{r}_1) \cap sources(0@\mathtt{y} \twoheadrightarrow \mathtt{r}_2) = \emptyset$ hold (see Figure 1). From our assumption $\gamma_1(\Phi, \Psi)$, we obtain $\phi_{\mathrm{WR}} \in \Phi$. Together, this implies that $next_\Phi(pa, 1)$ holds. From $next_\Phi(pa, 1)$, $pa[1] = (0, 0@\mathtt{y} \twoheadrightarrow \mathtt{r}_2)$, $pa[0] = (0, \mathtt{x} \leftarrow 0@\mathtt{r}_1)$, $isRead(0@\mathtt{y} \twoheadrightarrow \mathtt{r}_2)$ and our assumption $\delta_1(\Phi, \Psi)$, we can conclude that $isWrite(\mathtt{x} \leftarrow 0@\mathtt{r}_1)$ does not hold. This is a contradiction, as $\mathtt{x} \leftarrow 0@\mathtt{r}_1$ is a write obligation.

Table II shows which of our conditions γ_1, γ_2, γ_3, δ_1, δ_2, and δ_3 are satisfied by which of the memory models in $\mathcal{MM}$. The argument for each entry in this table is straightforward.

$$\begin{array}{l}\delta_1(\Phi,\Psi)\equiv\\ \forall pa\in Pa.\forall i\in\mathcal{I}.\forall ob\in Ob.\forall m<(|pa|-1).\\ (next_\Phi(pa,m)\wedge pa[m]=(i,ob))\\ \Rightarrow\left[\begin{array}{l}\forall j\in\mathcal{I}.\forall ob'\in Ob.\forall k<m.\\ pa[k]=(j,ob')\\ \Rightarrow\left(\begin{array}{l}(isRead(ob)\wedge j=i)\\ \Rightarrow\neg isWrite(ob')\end{array}\right)\end{array}\right]\end{array}$$

$$\begin{array}{l}\delta_2(\Phi,\Psi)\equiv\\ \forall pa\in Pa.\forall i\in\mathcal{I}.\forall ob\in Ob.\forall m<(|pa|-1).\\ (next_\Phi(pa,m)\wedge pa[m]=(i,ob))\\ \Rightarrow\left[\begin{array}{l}\forall j\in\mathcal{I}.\forall ob'\in Ob.\forall k<m.\\ pa[k]=(j,ob')\\ \Rightarrow\left[\begin{array}{l}\left(\begin{array}{l}(ob\in Fe\wedge j=i))\\ \Rightarrow\neg isWrite(ob')\end{array}\right)\\ \wedge\left(\begin{array}{l}\forall x\in\mathcal{X}.\forall v\in\mathcal{V}.\forall r,r'\in\mathcal{R}.\\ (ob={?}@x\multimap r\wedge j=i)\\ \Rightarrow ob'\neq x\leftarrow v@r'\end{array}\right)\\ \wedge\left(\begin{array}{l}(isRead(ob)\wedge j=i)\\ \Rightarrow\neg isRead(ob')\end{array}\right)\\ \wedge\left(\begin{array}{l}(isWrite(ob)\wedge j=i)\\ \Rightarrow\neg isWrite(ob')\end{array}\right)\end{array}\right]\end{array}\right]\end{array}$$

$$\begin{array}{l}\delta_3(\Phi,\Psi)\equiv\\ \forall pa\in Pa.\forall i\in\mathcal{I}.\forall ob\in Ob.\forall m<(|pa|-1).\\ (next_\Phi(pa,m)\wedge pa[m]=(i,ob))\\ \Rightarrow\left[\begin{array}{l}\forall j\in\mathcal{I}.\forall ob'\in Ob.\forall k<m.\\ pa[k]=(j,ob')\\ \Rightarrow\left[\begin{array}{l}\left(\begin{array}{l}(isRead(ob)\wedge j=i)\\ \Rightarrow\neg isRead(ob')\end{array}\right)\\ \wedge\left(\begin{array}{l}(isWrite(ob)\wedge j=i)\\ \Rightarrow\neg isWrite(ob')\end{array}\right)\end{array}\right]\end{array}\right]\end{array}$$

Figure 8. Definitions of δ_1, δ_2, and δ_3

MM	$\gamma_1(MM)$	$\gamma_2(MM)$	$\gamma_3(MM)$	$\delta_1(MM)$	$\delta_2(MM)$	$\delta_3(MM)$
SC				✓	✓	✓
IBM370	✓				✓	✓
TSO	✓	✓				✓
PSO	✓	✓	✓			

Table II
SATISFACTION OF THE DISCRIMINATING CONDITIONS

For γ_1, γ_2, and γ_3 the entries are an immediate consequence of the definitions of the conditions and of Definition 2.

From Table II, one can see that for all $MM, MM' \in \mathcal{MM}$ with $MM \neq MM'$, there is a $l \in \{1,2,3\}$ such that either $\gamma_l(MM) \wedge \delta_l(MM')$ or $\gamma_l(MM') \wedge \delta_l(MM)$ holds. This means that our choice of (γ_1,δ_1), (γ_2,δ_2), and (γ_3,δ_3) is suitable for discriminating the memory models in $\mathcal{MM}$.

The following three lemmas refer to the programs c_1^+, c_1^-, c_2^+, c_2^-, c_3^+, and c_3^-, in Figures 9, 10, and 11. Each thread in these programs starts with an initialization phase, which is omitted in the figures for brevity. In this initialization phase, each constant value used in the program is

c_1^+ :=
store$_1$ x 1; **store**$_2$ y 1; **store**$_3$ z 0; **store**$_4$ l 0;
spawn$_5$(
 store$_6$ x 0; **load**$_7$ r$_2$ y; **load**$_8$ r$_3$ z;
 and$_9$ r$_4$ r$_2$ r$_3$; **load**$_{10}$ r$_5$ h;
 if$_{11}$ r$_4$
 then if$_{12}$ r$_5$ **then store**$_{13}$ l 5 **else skip**$_{14}$ **fi**
 else if$_{15}$ r$_5$ **then skip**$_{16}$ **else store**$_{17}$ l 5 **fi fi**);
store$_{18}$ y 0; **load**$_{19}$ r$_1$ x; **store**$_{20}$ z r$_1$; **store**$_{21}$ z 0

c_1^- :=
store$_1$ x 1; **store**$_2$ y 1; **store**$_3$ z 0;
spawn$_4$(
 store$_5$ x 0; **load**$_6$ r$_2$ y; **load**$_7$ r$_3$ z; **and**$_8$ r$_4$ r$_2$ r$_3$;
 if$_9$ r$_4$ **then load**$_{10}$ r$_5$ h; **store**$_{11}$ l r$_5$ **else skip**$_{12}$ **fi**);
store$_{13}$ y 0; **load**$_{14}$ r$_1$ x; **store**$_{15}$ z r$_1$

Figure 9. Programs for Lemma 1

read into a dedicated register, which is not overwritten by the program afterwards. This allows us to use these registers in store operations for writing constant values into variables. Instead of referring to the dedicated registers, we directly use the corresponding constant values in store operations, i.e., **store**$_\iota$ x v denotes **store**$_\iota$ x r_v where r_v is the dedicated register for value v.

Lemma 1. *For the domain assignment lev with lev*(h) = **High** *and lev*(x) = **Low** *for all* $x \in \mathcal{X} \setminus \{\mathtt{h}\}$*, the following four propositions hold for each* $MM \in \mathcal{MM}$*:*

$$\gamma_1(MM) \Longrightarrow \mathrm{c}_1^+ \in NI_{MM} \quad \delta_1(MM) \Longrightarrow \mathrm{c}_1^+ \notin NI_{MM}$$
$$\gamma_1(MM) \Longrightarrow \mathrm{c}_1^- \notin NI_{MM} \quad \delta_1(MM) \Longrightarrow \mathrm{c}_1^- \in NI_{MM}$$

Lemma 2. *For the domain assignment lev with lev*(h) = **High** *and lev*(x) = **Low** *for all* $x \in \mathcal{X} \setminus \{\mathtt{h}\}$*, the following four propositions hold for each* $MM \in \mathcal{MM}$*:*

$$\gamma_2(MM) \Longrightarrow \mathrm{c}_2^+ \in NI_{MM} \quad \delta_2(MM) \Longrightarrow \mathrm{c}_2^+ \notin NI_{MM}$$
$$\gamma_2(MM) \Longrightarrow \mathrm{c}_2^- \notin NI_{MM} \quad \delta_2(MM) \Longrightarrow \mathrm{c}_2^- \in NI_{MM}$$

Lemma 3. *For the domain assignment lev with lev*(h) = **High** *and lev*(x) = **Low** *for all* $x \in \mathcal{X} \setminus \{\mathtt{h}\}$*, the following four propositions hold for each* $MM \in \mathcal{MM}$*:*

$$\gamma_3(MM) \Longrightarrow \mathrm{c}_3^+ \in NI_{MM} \quad \delta_3(MM) \Longrightarrow \mathrm{c}_3^+ \notin NI_{MM}$$
$$\gamma_3(MM) \Longrightarrow \mathrm{c}_3^- \notin NI_{MM} \quad \delta_3(MM) \Longrightarrow \mathrm{c}_3^- \in NI_{MM}$$

This concludes our proof sketch. The theorem follows from the three lemmas, as explained in the outline of our proof technique at the beginning of this subsection. A more detailed proof of Theorem 1 is made available under `http://www.mais.informatik.tu-darmstadt.de/assets/publications/CSF2014-mps.pdf` .

```
$c_2^+$ :=
$\mathbf{store}_1$ x 1; $\mathbf{store}_2$ y 1; $\mathbf{store}_3$ z 0; $\mathbf{store}_4$ l 0;
$\mathbf{spawn}_5$(
 $\mathbf{store}_6$ x 0; $\mathbf{fence}_7$; $\mathbf{load}_8$ $r_2$ y; $\mathbf{load}_9$ $r_3$ z;
 $\mathbf{and}_{10}$ $r_4$ $r_2$ $r_3$; $\mathbf{load}_{11}$ $r_5$ h;
 $\mathbf{if}_{12}$ $r_4$
 $\mathbf{then}$ $\mathbf{if}_{13}$ $r_5$ $\mathbf{then}$ $\mathbf{store}_{14}$ l 5 $\mathbf{else}$ $\mathbf{skip}_{15}$ $\mathbf{fi}$
 $\mathbf{else}$ $\mathbf{if}_{16}$ $r_5$ $\mathbf{then}$ $\mathbf{skip}_{17}$ $\mathbf{else}$ $\mathbf{store}_{18}$ l 5 $\mathbf{fi}$
 $\mathbf{fi}$);
$\mathbf{store}_{19}$ y 0; $\mathbf{load}_{20}$ $r_2$ y;
$\mathbf{load}_{21}$ $r_1$ x; $\mathbf{store}_{22}$ z $r_1$; $\mathbf{store}_{23}$ z 0
```

```
$c_2^-$ :=
$\mathbf{store}_1$ x 1; $\mathbf{store}_2$ y 1; $\mathbf{store}_3$ z 0;
$\mathbf{spawn}_4$(
 $\mathbf{store}_5$ x 0; $\mathbf{fence}_6$;
 $\mathbf{load}_7$ $r_2$ y; $\mathbf{load}_8$ $r_3$ z; $\mathbf{and}_9$ $r_4$ $r_2$ $r_3$
 $\mathbf{if}_{10}$ $r_4$ $\mathbf{then}$ $\mathbf{load}_{11}$ $r_5$ h; $\mathbf{store}_{12}$ l $r_5$ $\mathbf{else}$ $\mathbf{skip}_{13}$ $\mathbf{fi}$);
$\mathbf{store}_{14}$ y 0; $\mathbf{load}_{15}$ $r_2$ y; $\mathbf{load}_{16}$ $r_1$ x; $\mathbf{store}_{17}$ z $r_1$
```

Figure 10. Programs for Lemma 2

```
$c_3^+$ :=
$\mathbf{store}_1$ x 1; $\mathbf{store}_2$ y 0; $\mathbf{store}_3$ l 0;
$\mathbf{spawn}_4$(
 $\mathbf{load}_5$ $r_2$ y; $\mathbf{load}_6$ $r_1$ x; $\mathbf{and}_7$ $r_3$ $r_1$ $r_2$; $\mathbf{load}_8$ $r_4$ h;
 $\mathbf{if}_9$ $r_3$
 $\mathbf{then}$ $\mathbf{if}_{10}$ $r_4$ $\mathbf{then}$ $\mathbf{store}_{11}$ l 5 $\mathbf{else}$ $\mathbf{skip}_{12}$ $\mathbf{fi}$
 $\mathbf{else}$ $\mathbf{if}_{13}$ $r_4$ $\mathbf{then}$ $\mathbf{skip}_{14}$ $\mathbf{else}$ $\mathbf{store}_{15}$ l 5 $\mathbf{fi}$
 $\mathbf{fi}$);
$\mathbf{store}_{16}$ x 0; $\mathbf{store}_{17}$ y 1
```

```
$c_3^-$ :=
$\mathbf{store}_1$ x 1; $\mathbf{store}_2$ y 0;
$\mathbf{spawn}_3$(
 $\mathbf{load}_4$ $r_2$ y; $\mathbf{load}_5$ $r_1$ x; $\mathbf{and}_6$ $r_3$ $r_1$ $r_2$;
 $\mathbf{if}_7$ $r_3$ $\mathbf{then}$ $\mathbf{load}_8$ $r_4$ h; $\mathbf{store}_9$ l $r_4$ $\mathbf{else}$ $\mathbf{skip}_{10}$ $\mathbf{fi}$);
$\mathbf{store}_{11}$ x 0; $\mathbf{store}_{12}$ y 1
```

Figure 11. Programs for Lemma 3

VII. A Sound Analysis for Weak Memory Models

In this section we present our transforming security type system. This type system inserts fences to ensure security under the four memory models SC, IBM370, TSO, and PSO.

To capture that a command $c \in \mathcal{C}$ is transformed to $c' \in \mathcal{C}$, we introduce the judgment $pc, pt \vdash_{lev} c \diamond (pt', c')$, where $lev : (\mathcal{X} \cup \mathcal{R}) \to \{\mathbf{Low}, \mathbf{High}\}$ and $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$. Note that we use lev to assign security levels to both, variables and registers in this section. The domain assignment lev defines the policy that the transformation shall establish. The program counter pc is an upper bound on the security level on which it depends whether the command c is executed. The path-types pt and pt' are lower bounds on the security levels of variables and registers for which updates might be pending.

Figure 12 defines the rules of our transforming type system. The rules [LC], [LX], [OP] and [ST] prevent direct and indirect information leaks by checking that the security domains of the sources of the command and the program counter are lower than the security domain of the targets of the update.

The rule [IH] prevents indirect leaks by checking that each of the branches of the **if** command is type-able with a **High** program counter. Being the only rule for while, [WL] ensures that termination behavior does not depend on information from the security domain **High**. The rules [IL] and [SQ] propagate the analysis into the branches of **if** commands and into the components of a sequential composition, respectively.

We use the path-types pt and pt' to track the lower bound of the security levels for which obligations might be pending. All rules, except for the rule [IT] and [FN], might lower the path-type, but cannot raise it. The rule [FN] may raise the path-type, because a fence ensures that the path is empty before assuming the next obligation. The rule [IT] inserts a **fence** before the **if** and, hence, may raise the path-type. This ensures that the path does not contain any updates of **Low** variables or registers when entering a branching that depends on a **High** register.

The transformation results in a program that is noninterfering under the memory models SC, IBM370, TSO and PSO:

Theorem 2. *If $pc, \mathbf{High} \vdash_{lev} c \diamond (pt, c')$ is derivable for some $pc, pt \in \{\mathbf{Low}, \mathbf{High}\}$, then $c' \in NI_{MM}$ for each $MM \in \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$.*

Due to space restrictions, the proofs of Theorems 2, 3, and 4 are not presented in this article. They are available under `http://www.mais.informatik.tu-darmstadt.de/assets/publications/CSF2014-mps.pdf` .

The soundness of the transformation does not come at the price of always establishing sequentially consistent behavior:

Theorem 3. *There are c, c', pc, pt, lev, mem and mem' such that $pc, \mathbf{High} \vdash_{lev} c \diamond (pt, c')$ and $\langle c', mem \rangle \Downarrow_{\mathrm{PSO}} mem'$ are derivable although $\langle c', mem \rangle \Downarrow_{\mathrm{SC}} mem'$ is not derivable.*

Proof sketch: Consider the programs c and c' and the domain assignment lev in Figure 13. The judgment $\mathbf{Low}, \mathbf{High} \vdash_{lev} c \diamond (\mathbf{Low}, c')$ is derivable with the rules [SQ], [LX], [LC], [SP], [ST], [OP], [IT], [FN], and [SK].

For the program c' final memories are reachable under PSO that are not reachable under SC. Running c' under SC with an initial memory mem where $mem(\mathtt{x}) = 1$ and $mem(\mathtt{y}) = 0$ can never result in a final memory mem' with $mem'(\mathtt{l}_2) = 1$. To reach such a final memory, the obligations of $\mathbf{load}_6$ $\mathtt{r}_5$ y and $\mathbf{load}_7$ $\mathtt{r}_6$ x must both update their target registers to a non-zero value such that the obligation of $\mathbf{and}_9$ $\mathtt{r}_8$ $\mathtt{r}_5$ $\mathtt{r}_6$ updates $\mathtt{r}_8$ to 1 and the obligation of $\mathbf{store}_{11}$ $\mathtt{l}_2$ $\mathtt{r}_8$ updates $\mathtt{l}_2$ to 1. Since the initial value of y is 0, the variable y must be updated before fulfilling the obligation of $\mathbf{load}_6$ $\mathtt{r}_5$ y. The only obligation that updates y is the obligation of $\mathbf{store}_{13}$ y $\mathtt{r}_3$. Since SC does not permit any reordering, this implies that $\mathbf{store}_{12}$ x $\mathtt{r}_2$ must also be fulfilled before the obligation of $\mathbf{load}_6$ $\mathtt{r}_5$ y and, consequently, also before the obligation of $\mathbf{load}_7$ $\mathtt{r}_6$ x.

$$[\text{SK}]\ \frac{}{pc, pt \vdash_{lev} \mathbf{skip}_\iota \diamond (pt, \mathbf{skip}_\iota)} \qquad [\text{FN}]\ \frac{}{pc, pt \vdash_{lev} \mathbf{fence}_\iota \diamond (\mathbf{High}, \mathbf{fence}_\iota)}$$

$$[\text{LC}]\ \frac{v \in \mathcal{V} \qquad pc \sqsubseteq lev(r)}{pc, pt \vdash_{lev} \mathbf{load}_\iota\ r\ v \diamond (pt \sqcap lev(r), \mathbf{load}_\iota\ r\ v)} \qquad [\text{LX}]\ \frac{x \in \mathcal{X} \qquad lev(x) \sqcup pc \sqsubseteq lev(r)}{pc, pt \vdash_{lev} \mathbf{load}_\iota\ r\ x \diamond (pt \sqcap lev(r), \mathbf{load}_\iota\ r\ x)}$$

$$[\text{OP}]\ \frac{op \in \{\mathbf{and}, \mathbf{eq}\} \qquad lev(r_2) \sqcup lev(r_3) \sqcup pc \sqsubseteq lev(r_1)}{pc, pt \vdash_{lev} op_\iota\ r_1\ r_2\ r_3 \diamond (pt \sqcap lev(r_1), op_\iota\ r_1\ r_2\ r_3)} \qquad [\text{ST}]\ \frac{lev(r) \sqcup pc \sqsubseteq lev(x)}{pc, pt \vdash_{lev} \mathbf{store}_\iota\ x\ r \diamond (pt \sqcap lev(x), \mathbf{store}_\iota\ x\ r)}$$

$$[\text{SP}]\ \frac{pc, pt' \vdash_{lev} c \diamond (pt'', c')}{\mathbf{Low}, pt \vdash_{lev} \mathbf{spawn}_\iota c \diamond (\mathbf{Low}, \mathbf{spawn}_\iota c')} \qquad [\text{SQ}]\ \frac{pc, pt \vdash_{lev} c_1 \diamond (pt', c_1') \qquad pc, pt' \vdash_{lev} c_2 \diamond (pt'', c_2')}{pc, pt \vdash_{lev} c_1;\ c_2 \diamond (pt'', c_1';\ c_2')}$$

$$[\text{IL}]\ \frac{lev(r) = \mathbf{Low} \qquad pc, pt \vdash_{lev} c_1 \diamond (pt', c_1') \qquad pc, pt \vdash_{lev} c_2 \diamond (pt'', c_2')}{pc, pt \vdash_{lev} \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \diamond (pt' \sqcap pt'', \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi})}$$

$$[\text{IH}]\ \frac{lev(r) = pt = \mathbf{High} \qquad \mathbf{High}, pt \vdash_{lev} c_1 \diamond (pt, c_1') \qquad \mathbf{High}, pt \vdash_{lev} c_2 \diamond (pt, c_2')}{pc, pt \vdash_{lev} \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \diamond (\mathbf{High}, \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi})}$$

$$[\text{IT}]\ \frac{lev(r) = \mathbf{High} \quad pt = \mathbf{Low} \quad \iota' \text{ is fresh} \quad \mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High}, c_1') \quad \mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High}, c_2')}{pc, pt \vdash_{lev} \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \diamond (\mathbf{High}, \mathbf{fence}_{\iota'};\ \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi})}$$

$$[\text{WL}]\ \frac{pc = lev(r) = \mathbf{Low} \qquad pc, \mathbf{Low} \vdash_{lev} c \diamond (pt', c')}{pc, pt \vdash_{lev} \mathbf{while}_\iota\ r\ \mathbf{do}\ c\ \mathbf{od} \diamond (pt \sqcap pt', \mathbf{while}_\iota\ r\ \mathbf{do}\ c'\ \mathbf{od})}$$

Figure 12. Transforming security type system for SC, IBM370, TSO, and PSO

This means that the obligation of $\mathbf{load}_7\ \mathtt{r}_6\ \mathtt{x}$ will update its register to 0 in this case and, consequently, the final value of $\mathtt{l}_2$ is 0. However, the program-order relaxation write-to-write of PSO allows fulfilling the obligation of $\mathbf{store}_{13}\ \mathtt{y}\ \mathtt{r}_3$ before the obligation $\mathbf{store}_{12}\ \mathtt{x}\ \mathtt{r}_2$. Consequently, it is possible that the obligation of $\mathbf{load}_6\ \mathtt{r}_5\ \mathtt{y}$ and $\mathbf{load}_7\ \mathtt{r}_6\ \mathtt{x}$ both update their target register to 1. Thus a final memory mem' with $mem'(\mathtt{l}_2) = 1$ is reachable. This shows that the transformed program c' does not have sequentially consistent behavior. ■

$c := c_1;\ \mathbf{if}_{14}\ \mathtt{r}_1\ \mathbf{then}\ \mathbf{fence}_{15}\ \mathbf{else}\ \mathbf{skip}_{16}\ \mathbf{fi};\ c_2$
$c' := c_1; \mathbf{fence}_{18}; \mathbf{if}_{14}\ \mathtt{r}_1\ \mathbf{then}\ \mathbf{fence}_{15}\ \mathbf{else}\ \mathbf{skip}_{16}\ \mathbf{fi};\ c_2$
where
$c_1 :=$
$\mathbf{load}_1\ \mathtt{r}_1\ \mathtt{h};\ \mathbf{load}_2\ \mathtt{r}_2\ 0;\ \mathbf{load}_3\ \mathtt{r}_3\ 1;$
$\mathbf{spawn}_4($
$\quad\mathbf{load}_5\ \mathtt{r}_4\ \mathtt{z};\ \mathbf{load}_6\ \mathtt{r}_5\ \mathtt{y};\ \mathbf{load}_7\ \mathtt{r}_6\ \mathtt{x};\ \mathbf{and}_8\ \mathtt{r}_7\ \mathtt{r}_4\ \mathtt{r}_6;$
$\quad\mathbf{and}_9\ \mathtt{r}_8\ \mathtt{r}_5\ \mathtt{r}_6;\ \mathbf{store}_{10}\ \mathtt{l}_1\ \mathtt{r}_7;\ \mathbf{store}_{11}\ \mathtt{l}_2\ \mathtt{r}_8);$
$\mathbf{store}_{12}\ \mathtt{x}\ \mathtt{r}_2;\ \mathbf{store}_{13}\ \mathtt{y}\ \mathtt{r}_3$
$c_2 := \mathbf{store}_{17}\ \mathtt{z}\ \mathtt{r}_3$
$lev(\mathtt{h}) = lev(\mathtt{r}_1) = \mathbf{High},$
$lev(x) = \mathbf{Low}$ for all $x \in \mathcal{X} \setminus \{\mathtt{h}\}$
$lev(r) = \mathbf{Low}$ for all $r \in \mathcal{R} \setminus \{\mathtt{r}_1\}.$

Figure 13. Transformed program without sequentially consistent behavior

Figure 12 defines a transformation that is idempotent.

Theorem 4. *Let* $c, c' \in \mathcal{C}$ *and* $pc, pt \in \{\mathbf{Low}, \mathbf{High}\}$. *If* $pc, \mathbf{High} \vdash_{lev} c \diamond (pt, c')$ *is derivable, then* $pc, \mathbf{High} \vdash_{lev} c' \diamond (pt, c')$ *is also derivable.*

Theorems 2 and 3 show that it is possible to enforce noninterference under multiple memory models without having to establish sequential consistency. This means that programs resulting from the application of our type system may still gain performance from relaxed consistency guarantees.

Theorem 4 shows that our transforming security type system accepts all programs that resulted from an application of this type system. Moreover, a repeated application of our transforming security type system does not result in an additional increase of the size of a program.

VIII. Related Work

Work on information flow analysis for concurrent programs was pioneered by Reitman and Andrews [21]. To present information flow analyses in the form of type systems together with a soundness proof against a declarative noninterference-like condition has become popular since the seminal work by Volpano, Smith, and Irvine [22]. Volpano and Smith also proposed security type systems for concurrent programs together with soundness proofs [23], [24]. Many further information flow analysis for concurrent programs have been proposed since, e.g., [6], [25], [26], [27], [28].

To our knowledge, the only prior publication on information flow security that considers weak memory models is [13]. Vaughan and Millstein investigated noninterference for the memory model total store order. They showed that noninterference under sequential consistency does not imply noninterference under total store order and vice versa. Our work generalizes their result to further memory models.

Vaughan and Millstein proposed a security type system that is sound for total store order and showed how to make this security type system more precise by adding a flow-sensitive tracking of a security type for their write buffer. In our security type system, we perform a flow sensitive tracking of the security type of the path. However, we track this security type for a different purpose, namely to establish portable security guarantees, i.e. security guarantees that are valid for all four memory models sequential consistency, `IBM 370`, total store order, and partial store order.

The body of related work on memory consistency outside security is rich. Lamport defined sequential consistency as a consistency criterion for computations on multi-processor systems in [10]. While sequential consistency is very intuitive, it reduces the possibilities for effective use of hardware and compiler optimizations. To overcome this, memory models with relaxed consistency guarantees were developed. Adve and Gharachorloo gave in [11], [12] an informal overview on memory models with relaxed security guarantees and proposed a taxonomy based on three program-order relaxations (write-to-read, write-to-write, and write-to-read/write), the write-atomicity relaxation read-others-write early, and the relaxation read-own-writes-early. Their work inspired the modular representation of relaxations of sequential consistency in our model of computation. This modularity enables us to clearly distinguish between the various program-order relaxations and write-atomicity relaxations.

To investigate and compare which program runs are possible under different memory models, generic execution models have been proposed. One prominent framework is the one by Alglave et al [29], [30], [15]. Like the original definition of total store order and partial store order in [20], this framework is defined in an axiomatic style. Program runs are captured by read and write events, and by relations on these events. Event structures and execution witnesses are constructed from such events and relations. A given pair of an event structure and an execution witness describes one or more terminated program runs. That is, two different program runs might have the same representation. However, two given program runs might also have different representations in terms of event structures and execution witnesses. Not being able to distinguish program runs based on their representation, complicates the definition of noninterference and makes it difficult to prove noninterference in a compositional fashion. In contrast, our model of computation provides a unique representation of each program run and an explicit representation of intermediate states. The latter facilitates reasoning about noninterference compositionally in terms of individual computation steps.

An operational approach as it is taken for instance in [31] provides easier access to intermediate states. Boudol and Petri's execution model in [31] is tied to a particular memory model, which is defined by a combination of the relaxations write-to-read, write-to-write, read-own-writes-early, and read-others-writes-early. In [32], Boudol, Petri and Serpette propose a generic execution model with a similar flavor. This execution model provides a small-step semantics for different memory models. The programming language considered is a λ-calculus with concurrency features including thread creation, but without recursion or loops. To describe weak memory models a temporary store is introduced in [32] that, similar to our concept of paths, buffers reads and writes until they take effect. The key concepts for describing permitted relaxations of sequential consistency, a commutability predicate and a write grain, build on the intuition of program-order and write-atomicity relaxations from [11], [12], similar to our predicates for program-order and write-atomicity relaxations. However, in contrast to our modular approach that allows one to compose pre-defined predicates to define a memory model, there is no support for defining the commutability predicate in [32]. Another important difference is that our states assign values to both memory locations and registers, while states in [32] assign values to memory locations only. Instead of referring to registers, a write event in [32] refers to the read event on which the value to be written depends until this value can be retrieved. As also pointed out in [32] for reasoning about low-level models an explicit representation of registers is desirable.

Program transformations that establish information flow security have been proposed before, e.g., in [33], [6], [34], [35]. Many of these transformations aim at the elimination of internal timing leaks in concurrent programs. To the best of our knowledge, fence insertion techniques have not been applied in the area of information flow security before.

Fence insertion techniques themselves, however, have been studied in depth, e.g., in [36], [37], [38], [39], [40], [41]. Many fence insertion techniques aim at establishing sequential consistency. In our transformation, we avoid to establish sequential consistency as this would reduce the benefits of weak memory models. One motivation for relaxing sequential consistency is to gain performance, but this gain is lost, if sequentially consistent behavior is established, despite the weak memory model. To minimize the insertion of fences, we guide our transformation by the rules of our type system.

IX. Conclusion

The aim of our research was to better clarify the impact of weak memory models on information flow security. In this article, we showed that one cannot rely on the preservation of noninterference if one gives up sequential consistency. This was already known for total store order [13], but it was not clear for the memory models partial store order and `IBM 370` before. In addition, we showed that one cannot rely on the preservation of noninterference when migrating between weak memory models. While it might not be surprising that this can happen if one migrates from one weak memory model to another, we found it surprising that noninterference is not preserved no matter which two memory models one considers and no matter in which direction one migrates. In this article, we studied information flow security under four memory models. There are further relaxations of sequential consistency, whose impact on noninterference remains to be clarified.

The transforming type system that we presented is, to our knowledge, the first solution for soundly establishing noninterference under multiple weak memory models. At this point, we just employ a simple fence-insertion technique. To eliminate further insecurities, it would be desirable to integrate more sophisticated program modifications, however, without endangering sound enforcement of noninterference.

Altogether the study of information flow security under relaxed consistency guarantees has just begun.

ACKNOWLEDGMENTS.

We thank Andrei Sabelfeld, Barbara Sprick, and the reviewers for valuable criticism and constructive comments. This work was funded by the DFG under the project RSCP (MA 3326/4-2) in the priority program RS^3 (SPP 1496).

REFERENCES

[1] B. W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[2] D. E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[3] E. S. Cohen, "Information Transmission in Sequential Programs," *Foundations of Secure Computation*, pp. 297–335, 1978.

[4] J. M. Rushby, "Design and Verification of Secure Systems," in *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pp. 12–21, 1981.

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

[6] A. Sabelfeld and D. Sands, "Probabilistic Noninterference for Multi-threaded Programs," in *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pp. 200–215, 2000.

[7] H. Mantel and H. Sudbrock, "Flexible Scheduler-Independent Security," in *Proceedings of the 15th European Symposium on Research in Computer Security*, pp. 116–133, 2010.

[8] A. Sabelfeld and D. Sands, "Dimensions and Principles of Declassification," in *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pp. 255–269, 2005.

[9] A. Lux, H. Mantel, and M. Perner, "Scheduler-Independent Declassification," in *Proceedings of the 11th International Conference on Mathematics of Program Construction*, pp. 25–47, 2012.

[10] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, pp. 690–691, 1979.

[11] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial. Western Research Laboratory," tech. rep., Research Report 95/7, 1995.

[12] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66 – 76, 1996.

[13] J. A. Vaughan and T. Millstein, "Secure Information Flow for Concurrent Programs under Total Store Order," in *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pp. 19–29, 2012.

[14] J. Alglave and L. Maranget, "Stability in weak memory models," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, pp. 50–66, 2011.

[15] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design*, vol. 41, no. 2, pp. 178–210, 2012.

[16] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: x86-TSO," in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pp. 391–407, 2009.

[17] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER multiprocessors," in *Proceedings of the 32nd ACM Conference on Programming Language Design and Implementation*, pp. 175–186, 2011.

[18] "The SPARC Architecture Manual, Version 9," 1994.

[19] "IBM System/370 Principles of Operation," 1983.

[20] "The SPARC Architecture Manual, Version 8," 1991.

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

[22] D. Volpano, G. Smith, and C. Irvine, "A Sound Type System for Secure Flow Analysis," *Journal of Computer Security*, vol. 4(3), pp. 1–21, 1996.

[23] D. Volpano and G. Smith, "Probabilistic Noninterference in a Concurrent Language," in *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pp. 34–43, 1998.

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

[25] G. Smith, "A New Type System for Secure Information Flow," in *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pp. 115–125, 2001.

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

[27] A. Russo and A. Sabelfeld, "Securing Interaction between Threads and the Scheduler," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pp. 177–189, 2006.

[28] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and Guarantees for Compositional Noninterference," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium*, pp. 218–232, 2011.

[29] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, pp. 258–272, 2010.

[30] J. Alglave, *A Shared Memory Poetics*. PhD thesis, Paris Diderot University, 2010.

[31] G. Boudol and G. Petri, "Relaxed memory models: an operational approach," in *Proceedings of the 36th ACM Symposium on Principles of Programming Languages*, pp. 392–403, 2009.

[32] G. Boudol, G. Petri, and B. Serpette, "Relaxed Operational Semantics of Concurrent Programming Languages," in *Proceedings of Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics*, pp. 19–33, 2012.

[33] J. Agat, "Transforming out Timing Leaks," in *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pp. 40–53, 2000.

[34] I. Siveroni, "Filling Out the Gaps: A Padding Algorithm for Transforming Out Timing Leaks," in *Proceedings of the 3rd Workshop on Quantitative Aspects of Programming Languages*, no. 2 in ENTCS 153, pp. 241–257, 2006.

[35] B. Köpf and H. Mantel, "Transformational Typing and Unification for Automatically Correcting Insecure Programs," *International Journal of Information Security*, vol. 6, no. 2–3, pp. 107–131, 2007.

[36] D. Shasha and M. Snir, "Efficient and Correct Execution of Parallel Programs that Share Memory," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 2, pp. 282–312, 1988.

[37] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *Proceedings of the 17th Annual International Conference on Supercomputing*, pp. 285–294, 2003.

[38] S. Burckhardt, R. Alur, and M. M. K. Martin, "CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 12–21, 2007.

[39] A. Linden and P. Wolper, "A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems," in *SPIN*, pp. 144–160, 2011.

[40] M. Kuperstein, M. T. Vechev, and E. Yahav, "Automatic Inference of Memory Fences," *SIGACT News*, vol. 43, no. 2, pp. 108–123, 2012.

[41] A. Linden and P. Wolper, "A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 339–353, 2013.