Noninterference under Weak Memory Models (Progress Report)

Technical Report TUD-CS-2014-0062

March 2014

Heiko Mantel,
Matthias Perner,
Jens Sauer

TECHNISCHE
UNIVERSITÄT
DARMSTADT

MAIS
Modeling and Analysis
of Information Systems

Noninterference under Weak Memory Models (Progress Report)

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 [Lam73], [Den76], [Coh78], [Rus81], [GM82] 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.

In this article, we study information flow security in the presence of concurrency. Prior research in this area led, e.g., to noninterference-like security properties, which are suitable for expressing information flow security for concurrent programs, and analysis techniques, which are suitable for certifying that concurrent programs have secure information flow.

Concurrency is a rich domain of foundational research problems itself and combining it with information flow security results in intriguing problems that, in order to achieve reliable security for concurrent programs, require solutions. The combination of information flow security with concurrency leads to new problems, such as how to achieve reliable information flow security without knowing how the scheduler works (see, e.g., [SS00], [MS10]), and further complicates problems that are already non-trivial in a sequential setting, such as how to control declassification (see, e.g., [SS05], [LMP12]).

The focus of this article is on the effects of relaxed consistency guarantees on information flow security. Weak memory models provide weaker consistency guarantees than the sequential consistency property [Lam79] 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 program optimizations during compilation that are not compatible with sequential consistency. For an introduction to weak memory models, we refer to [AG95], [AG96].

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 (brief: TSO), on noninterference [VM12]. They showed that noninterference under sequential consistency (brief: SC) does not imply noninterference under TSO and, vice versa, that noninterference under TSO does not imply noninterference under SC. 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 TSO, and demonstrated how this type system can be refined to a more precise one without loosing soundness for TSO. We started our research on the effects of weak memory models on noninterference independently from Vaughan and Millstein (see [Sau12] for preliminary results).

The three main, novel contributions of this article are:

- We clarify the effects of four memory models on information flow security. For each of these models, we show that noninterference under this memory model does not imply noninterference under any of the other three 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 weak memory model, i.e., TSO. 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 for verifying noninterference under weak memory models. This is the first security type system that is known to be sound for multiple memory models. We prove soundness for IBM370, PSO, TSO, and also for SC. This means, a program verified by our type system remains secure if it is ported to any environment that supports one of the four memory models. Our type system is also the first transforming type system that is suitable for relaxed consistency guarantees. Our transformation inserts fence commands into a program such that it becomes secure under relaxed consistency guarantees. Although inspired by fence-insertion techniques that establish sequentially consistent behavior of a program [AM11], our transformation does not force sequential consistency on a program that is 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. We developed this model because we did not succeed in basing our study on any of the existing models of computation for relaxed consistency guarantees. Though originally a side product, we view this model also as a valuable contribution as it was helpful for our research on information flow security and might be helpful for others, not only in security.

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 the best of our knowledge, this is just 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. We present our clarification of the correlation between noninterference and our security type system in Section V and VI, respectively. After a discussion of related work in Section VII, we conclude in Section VIII.

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 $last(as)$ for the last element in as, i.e., $last(as) = as[|as| - 1]$. 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 $pre(f)$ to denote the pre-image of f, i.e. $pre(f) = \{a \in A \mid f(a) \in B\}$. Moreover, we write $f[a \mapsto b]$ for the function f' with $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \in (pre(f) \setminus \{a\})$. 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 $|pre(\vec{a})| \in \mathbb{N}$ 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 $Mem = \mathcal{X} \to \mathcal{V}$ to model states of the global memory and functions in $Reg = \mathcal{R} \to \mathcal{V}$ 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 $\vec{Reg} = \mathcal{I} \rightharpoonup Reg$ to model states of the registers of all threads. For a given thread identifier, a vector $\vec{reg} \in \vec{Reg}$ 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 $Gst = \vec{Reg} \times Mem$ and refer to such pairs as *global states*. We use pairs from $Lst = Reg \times Mem$ 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 $gst[i]$ for the local state of thread $i \in pre(gst)$ in a global state $gst \in Gst$, i.e., if $gst = (\vec{reg}, mem)$ then $gst[i] = (\vec{reg}(i), mem) \in Lst$. We call a thread $i \in \mathcal{I}$ *active* in gst if $i \in pre(gst)$ holds, and *inactive* otherwise. Note that $gst[i]$ 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, $\vec{reg}$ for local memories in $\vec{Reg}$, 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 registers sets.

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

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

For an expression $e \in \mathcal{E}$, we use $args(e)$ 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:

$$ev := x \leftarrow v@r \mid v@x \rightarrow r \mid v@e \circlearrowright r$$

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 $\mathit{effect} : Ev \to (Lst \to Lst)$ that we define by

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

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 ($v@x \twoheadrightarrow r$) as *read events*, to events that model a register update after an operation on registers ($v@e \circlearrowright r$) as *computation events*, and to events that model the transfer of a value from a register to the global memory ($x \twoheadleftarrow v@r$) 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., [AG95]).

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 [AG95], [OSS09], [SSA+11].

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 [AG95]. 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 require events and obligations to have the same granularity as commands in the considered programming, byte-code, or machine language. That is, each event and obligation reflects the execution of a single command.

ob	$x \twoheadleftarrow v@r$	$v@x \twoheadrightarrow r$	$?@x \twoheadrightarrow r$	$v@e \circlearrowright r$	fe
$sources(ob)$	$\{r\}$	$\{x\}$	$\{x\}$	$args(e)$	$\emptyset$
$sinks(ob)$	$\{x\}$	$\{r\}$	$\{r\}$	$\{r\}$	$\emptyset$

Table I. Sources and sinks of an obligation

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.

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:

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

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 $?@x \twoheadrightarrow r$ 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 $v@x \twoheadrightarrow r$. We re-use our syntax for events to denote the corresponding obligations. That is, $v@x \twoheadrightarrow r$ is the obligation to copy v from x to r, $v@e \circlearrowright r$ is the obligation to update r to v after an operation on registers, and $x \twoheadleftarrow v@r$ 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 $\mathit{isRead}(ob)$ holds if ob has the form $?@x \twoheadrightarrow r$ or $v@x \twoheadrightarrow r$, $\mathit{isComp}(ob)$ holds if ob has the form $v@e \circlearrowright r$, and $\mathit{isWrite}(ob)$ holds if ob has the form $x \twoheadleftarrow v@r$. Moreover, we define two functions $\mathit{sinks}, \mathit{sources} : ob \to 2^{\mathcal{X} \cup \mathcal{R}}$ 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 $sources([]) = []$, $sources([ob]::obs) = sources(ob) \cup sources(obs)$, $sinks([]) = []$, and $sinks([ob]::obs) = sinks(ob) \cup sinks(obs)$.

We use the events $v@x \rightarrowtriangle r$, $v@e \circlearrowright r$, and $x \triangleleft\!\!- v@r$ 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." [Lam79]

As elaborated in [AG95], 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 obeys 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 defines conditions under which an obligation in a given path may be fulfilled before an obligation of the same thread that occurs at an earlier position in this path. A predicate ϕ takes a list of obligations obs and a position k as arguments. We assume that the last obligation in obs is the one that shall be fulfilled out of order. Nevertheless, one can use ϕ 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 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:

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

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 given 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

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

Figure 1. Program-order relaxations for read and write

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

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

the case where the obligation ob has the form $?@x \twoheadrightarrow r$:

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

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 $x \notin sinks(pa\restriction_i)$ 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:

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

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.

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

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

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., [AG95]), 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 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. Similarly, the condition $sources(obs[k]) \cap sinks(last(obs)) = \emptyset$ prevents a re-ordering if the read obligation modifies a register on which the write obligation depends. Together, these two conditions ensure 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., [AG95]). 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 source of this obligation must differ from r, 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 these obligations because they 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.

We are now ready to formalize examples of concrete memory models. In Table II we specify the consistency guarantees provided by the memory models: *sequential consistency* (brief: SC), IBM370, *total store order* (brief: TSO), and *partial store order* (brief: PSO). Our specification of consistency guarantees

Memory Model	Φ	Ψ
SC	$\emptyset$	$\emptyset$
IBM370	$\{\phi_{\mathrm{WR}}\}$	$\emptyset$
TSO	$\{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}\}$	$\{\psi_{\mathrm{ROwn}}\}$
PSO	$\{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}, \phi_{\mathrm{WW}}\}$	$\{\psi_{\mathrm{ROwn}}\}$

Table II. EXAMPLE MEMORY MODELS

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

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

Figure 4. Program-order relaxations for computation obligations

for these models is equivalent to the one presented in [AG95]. Each of these memory models already had relevance in processor design PSO, TSO, and IBM370 are supported by SPARC, modern x86 processors, and in the processor IBM370, respectively [AG95], [OSS09].

Remark 1. *In our formal definitions of program-order relaxations and of write-atomicity relaxations in Figures 1, 2, and 3, we made some design decisions that resolve ambiguities in existing informal definitions. For instance, the definitions of our predicates ϕ_{WW} and ϕ_{RR} do not rule out source/source conflicts on registers and variables, respectively. Instead, one might prefer to re-order obligations only if they involve distinct registers and variables, which is a stronger restriction. Moreover, our formal definition of $\psi^{early}_{\mathrm{ROth}}$ in Figure 3 permits a thread i to read a write to a variable x by some thread $j \in early(i)$ even even if the path contains write obligations for x by the thread i, itself. Again, one might want to rule this out by strengthening the condition with the additional conjunct $x \notin sinks(pa[0 \ldots |pa|-2]\restriction_i)$. Our observations in Section V remain valid if one requires these stronger conditions.*

Remark 2. *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. In particular, it is straightforward to adapt the results in Sections V and VI to a language with more commands for computations. However, adding further commands to our language, would increase length of calculi, explanations, and proofs. To avoid such an increase and to ensure readability, we refrain from considering a richer language in this article.

The fence command in our language corresponds to a *full fence*. The execution of a full fence is only possible if all commands that appear before the fence in program order have been executed. Moreover, a full fence prevents commands that appear after the fence in program order to be executed before the fence. 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 $Op = \{\mathrm{const}, \mathrm{eq}, \mathrm{and}\}$ and the set of synchronization obligations by $Fe = \{\|, \nearrow_c | \ c \in \mathcal{C}\}$.

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 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 $\vec{cs}$ 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, we only consider global configurations that are 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 obligations are assumed. The judgment $\langle \vec{cs}(i), pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$ in the

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

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

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

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

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

third premise captures the processing of the next command in $\vec{cs}(i)$. This judgment is explained later in this section. The fourth premise ensures that a thread cannot assume new obligations if a fence obligation of this given 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 with a value v of x that is 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 $\langle c, pa, reg\rangle \to_i \langle c', pa'\rangle$ to capture that if thread i processes the command c

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6. Processing a command and augmentating a path

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

Figure 7. Big-step semantics for commands

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 of these registers are 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 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. Information Flow Security

The novel model of computation and its instantiation with a concrete language, which we described in Sections II–IV, 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 Table II (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. 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 $mem =_L mem'$.

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. This means, if $mem =_L mem'$ holds then two global memories $mem, mem' \in Mem$ differ only in secrets.

Definition 2. *A program $c \in \mathcal{C}$ is MM-*noninterfering *(denoted by $c \in NI_{MM}$), if the following condition holds:*

$$\begin{aligned}&\forall mem_0, mem_1, mem_0' \in Mem.\\ &mem_0 =_L mem_0' \wedge \langle c, mem_0\rangle \Downarrow_{MM} mem_1\\ &\implies \exists mem_1' \in Mem.\\ &\quad mem_1 =_L mem_1' \wedge \langle c, mem_0'\rangle \Downarrow_{MM} mem_1'\end{aligned}$$

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 [VM12]. Vaughan and Millstein showed that noninterference under SC does not imply noninterference under TSO and that noninterference under TSO does not imply noninterference under SC. Our theorem demonstrates that this observation is not just a peculiarity of one specific memory model, because it can also be made for SC and IBM370 as well as for SC and PSO. Moreover, our theorem also 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. For the rest of this section, let $\mathcal{MM} = \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$.

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 $MM \in \mathcal{MM}$:

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

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.

When applying this proof technique, still some creativity is needed to determine suitable pairs of discriminating conditions (γ_l, δ_l) and to determine, for each of these pairs of conditions, 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).

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

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

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

Figure 8. Definitions of δ_1, δ_2, and δ_3

We define the conditions γ_1, γ_2, and γ_3:

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

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 $pa = [(0, \mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)]::[(0, 0@\mathrm{y} \twoheadrightarrow \mathrm{r}_2)]$. For this path, $\phi_{\mathrm{WR}}(pa,1)$ holds, because $isWrite(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$, $isRead(0@\mathrm{y} \twoheadrightarrow \mathrm{r}_2)$, $sources(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1) \cap sinks(0@\mathrm{y} \twoheadrightarrow \mathrm{r}_2) = \emptyset$ and $sinks(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1) \cap sources(0@\mathrm{y} \twoheadrightarrow \mathrm{r}_2) = \emptyset$ 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 $next_\Phi(pa,1)$, $pa[1] = (0, 0@\mathrm{y} \twoheadrightarrow \mathrm{r}_2)$, $pa[0] = (0, \mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$, $isRead(0@\mathrm{y} \twoheadrightarrow \mathrm{r}_2)$ and our assumption $\delta_1(\Phi,\Psi)$, we can conclude that $isWrite(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$ does not hold. This is a contradiction, as $\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1$ is a write obligation.

Table III 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. For γ_1, γ_2, and γ_3 the entries are an immediate consequence of the definitions of the conditions and of Table II.

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

Table III. SATISFACTION OF THE DISCRIMINATING CONDITIONS

$\mathrm{c}_1^+ :=$
$\mathbf{store}_1$ 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; $\mathbf{load}_7$ r_2 y; $\mathbf{load}_8$ r_3 z;
$\mathbf{and}_9$ r_4 r_2 r_3; $\mathbf{load}_{10}$ r_5 h;
$\mathbf{if}_{11}$ r_4
$\mathbf{then}$ $\mathbf{if}_{12}$ r_5 $\mathbf{then}$ $\mathbf{store}_{13}$ l 5 $\mathbf{else}$ $\mathbf{skip}_{14}$ $\mathbf{fi}$
$\mathbf{else}$ $\mathbf{if}_{15}$ r_5 $\mathbf{then}$ $\mathbf{skip}_{16}$ $\mathbf{else}$ $\mathbf{store}_{17}$ l 5 $\mathbf{fi}$ $\mathbf{fi}$);
$\mathbf{store}_{18}$ y 0; $\mathbf{load}_{19}$ r_1 x; $\mathbf{store}_{20}$ z r_1; $\mathbf{store}_{21}$ z 0

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

Figure 9. Programs for Lemma 1

From Table III, 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.

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

$$
\begin{array}{ll}
\gamma_1(MM) \Longrightarrow \mathrm{c}_1^+ \in NI_{MM} & \delta_1(MM) \Longrightarrow \mathrm{c}_1^+ \notin NI_{MM} \\
\gamma_1(MM) \Longrightarrow \mathrm{c}_1^- \notin NI_{MM} & \delta_1(MM) \Longrightarrow \mathrm{c}_1^- \in NI_{MM}
\end{array}
$$

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

$$
\begin{array}{ll}
\gamma_2(MM) \Longrightarrow \mathrm{c}_2^+ \in NI_{MM} & \delta_2(MM) \Longrightarrow \mathrm{c}_2^+ \notin NI_{MM} \\
\gamma_2(MM) \Longrightarrow \mathrm{c}_2^- \notin NI_{MM} & \delta_2(MM) \Longrightarrow \mathrm{c}_2^- \in NI_{MM}
\end{array}
$$

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

$$
\begin{array}{ll}
\gamma_3(MM) \Longrightarrow \mathrm{c}_3^+ \in NI_{MM} & \delta_3(MM) \Longrightarrow \mathrm{c}_3^+ \notin NI_{MM} \\
\gamma_3(MM) \Longrightarrow \mathrm{c}_3^- \notin NI_{MM} & \delta_3(MM) \Longrightarrow \mathrm{c}_3^- \in NI_{MM}
\end{array}
$$

$c_2^+ :=$
$\mathbf{store}_1$ 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
then $\mathbf{if}_{13}$ r_5 **then** $\mathbf{store}_{14}$ l 5 **else** $\mathbf{skip}_{15}$ **fi**
else $\mathbf{if}_{16}$ r_5 **then** $\mathbf{skip}_{17}$ **else** $\mathbf{store}_{18}$ l 5 **fi**
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 **then** $\mathbf{load}_{11}$ r_5 h; $\mathbf{store}_{12}$ l r_5 **else** $\mathbf{skip}_{13}$ **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
then $\mathbf{if}_{10}$ r_4 **then** $\mathbf{store}_{11}$ l 5 **else** $\mathbf{skip}_{12}$ **fi**
else $\mathbf{if}_{13}$ r_4 **then** $\mathbf{skip}_{14}$ **else** $\mathbf{store}_{15}$ l 5 **fi**
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 **then** $\mathbf{load}_8$ r_4 h; $\mathbf{store}_9$ l r_4 **else** $\mathbf{skip}_{10}$ **fi**);
$\mathbf{store}_{11}$ x 0; $\mathbf{store}_{12}$ y 1

Figure 11. Programs for Lemma 3

This concludes our proof sketch, the theorem follows from the lemmas, as explained in the outline of our proof technique at the beginning of this subsection. A more detailed proof of Theorem 1 can be found in the appendix.

VI. A Sound Analysis for Weak Memory Models

In this section we propose a transforming type system. The type system inserts fences to ensure security under the four memory models SC, IBM370, TSO and PSO. The transformation does not enforce sequentially consistent behavior of the transformed program. Hence, the transformed program can still benefit from relaxed consistency guarantees to gain performance.

For the analysis we extend the domain assignment to registers. The extended domain assignment is a function $lev : (\mathcal{X} \cup \mathcal{R}) \to \{\mathbf{High}, \mathbf{Low}\}$.

The security type system is defined in Figure 12. The judgments derived using the type system have the form $pc, pt \vdash_{lev} c \diamond (pt', c')$ where $lev : (\mathcal{X} \cup \mathcal{R}) \to \{\mathbf{Low}, \mathbf{High}\}$, $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$ and $c, c' \in \mathcal{C}$. The domain assignment lev defines the policy that the transformation enforces. The program counter pc is an upper bound on the security level on which it depends whether the command is executed. The command c is transformed into the command c'. The path-types pt and pt' are a lower bound on the security levels of variables and registers for which updates might not be fulfilled yet.

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. The rule [WH] prevents that termination behavior depends on information from the security domain **High** by requiring that the condition is from the security domain **Low**. 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 not be fulfilled yet. Except for the rule [IT] and [FN] all rules might lower the path-type, but not raise it. The rule [FN] raises 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 raises 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 as the following Theorem shows.

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 all* $MM \in \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$.

Due to space restrictions, we refrain from presenting proofs in this submission. The proofs for Theorems 2, 3, and 4 can be found in the appendix.

The sound transformation does not come at the price of establishing sequential consistency as the following theorem shows.

Theorem 3. *The fact that* $pc, \mathbf{High} \vdash_{lev} c \diamond (pt, c')$ *for some* $pc, pt \in \{\mathbf{Low}, \mathbf{High}\}$ *is derivable does not imply that* $\langle c', mem\rangle \Downarrow_{MM} mem' \iff \langle c', mem\rangle \Downarrow_{\mathrm{SC}} mem'$ *for all* $MM \in \{\mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$ *holds.*

We briefly sketch the arguments with the programs and the domain assignment from 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 $mem(\mathrm{x}) = 1$ and $mem(\mathrm{y}) = 0$ can never result in a final memory mem' with $mem'(\mathrm{l}_2) = 1$. To reach such a final memory, the

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

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

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

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

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

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

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

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

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

obligations of $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$ and $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$ must both update their target registers to a non-zero value such that the obligation of $\mathbf{and}_9\ \mathrm{r}_8\ \mathrm{r}_5\ \mathrm{r}_6$ updates r_8 to 1 and the obligation of $\mathbf{store}_{11}\ \mathrm{l}_2\ \mathrm{r}_8$ updates l_2 to 1. Since the initial value of y is 0, the variable y must be updated before fulfilling the obligation of $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$. The only obligation that updates y is the obligation of $\mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$. Since SC does not permit any reordering, this implies that $\mathbf{store}_{12}\ \mathrm{x}\ \mathrm{r}_2$ must also be fulfilled before the obligation of $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$ and, consequently, also before the obligation of $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$. This means that the obligation of $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$ will update its register to 0 in this case and, consequently, the final value of l_2 is 0. However, the program-order relaxation write-to-write of PSO allows fulfilling the obligation of $\mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$ before the obligation $\mathbf{store}_{12}\ \mathrm{x}\ \mathrm{r}_2$. Consequently, it is possible that the obligation of $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$ and $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$ both update their target register to 1. Thus a final memory mem' with $mem'(\mathrm{l}_2) = 1$ is reachable. This shows that the transformed program does not have sequentially consistent behavior.

The proposed transformation is idempotent. This means that the type system can also be used for type checking a program after its transformation. The following theorem shows that the transformation is idempotent.

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

The analysis shows that it is possible to enforce noninterference without establishing sequential consistency. In addition, it shows that a program can be made secure under multiple memory models and therefore is a step towards security guarantees that are portable between memory models. At the same time the relaxed consistency guarantees can still lead to performance gains.

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

Figure 13. Transformed program without sequentially consistent behavior

VII. Related Work

Information flow analysis for concurrent programs was pioneered by Reitman and Andrews [RA79]. To present information flow analysis 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 [VSI96]. Volpano and Smith also proposed security type systems for concurrent programs together with soundness proofs [VS98], [VS99]. Many further information flow analysis for concurrent programs have been proposed since, e.g., [SS00], [Smi01], [BC02], [RS06], [MSS11]. However, the only publication that considers weak memory models so far is [VM12]. Vaughan and Millstein investigated noninterference for a weak memory model, namely TSO, for the first time and showed that noninterference under SC does not imply noninterference under TSO and vice versa.

They proposed a security type system for TSO and showed how to make this security type system more precise by adding a flow-sensitive tracking of a security type for the write buffer. Our work generalizes the result that SC and TSO cannot be ordered by implication with respect to noninterference to the memory models SC, IBM370, TSO and PSO. Like Vaughan and Millstein, we exploit the benefits of a flow-sensitive tracking of a security type for the path. In contrast to their intention of tracking the security type of the write buffer to achieve a higher precision, we use the tracking of the security type of the path to establish portable security results, i.e. security results that are sound under SC, IBM370, TSO and PSO at the same time.

The body of related work on memory consistency outside security is rich. Leslie Lamport defines in [Lam79] sequential consistency as a consistency criterion for computations on multi-processor systems. 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 give in [AG95], [AG96] an overview of different memory models with relaxed security guarantees and categorize the based on three program-order relaxations, namely write-to-read, write-to-write and write-to-read/write, a write-atomicity relaxation, namely read-others-write early, and a relaxations of both aspects, namely read-own-writes-early. Their systematic approach inspired our modular execution framework with the program-order relaxations and write-atomicity relaxations.

To understand and investigate the impact of different memory models generic frameworks have been proposed that can be instantiated for different memory models. These frameworks are defined either axiomatically, e.g. [AMSS10], [Alg10], [Alg12] or operationally, e.g. [BPS12]. For our purposes an operational model is very suitable. The operational semantics from [BPS12] for a λ-calculus with concurrency builds on the same intuition from [AG95], [AG96] of program-order and write-atomicity relaxations like our model. They introduce a temporary store that is similar to our path to collect interactions with the memory that are not fulfilled yet. To determine which interaction with the memory may be fulfilled next, they have a commutability predicate and to determine from which other threads a thread may read they have a write grain. The intention behind these concepts is similar to the intention behind our predicate $next$ and the function early. In contrast to our modular approach that provides predicates that can be combined to build up a concrete memory model, the commutability predicate must be instantiated with relations that capture permitted relaxations. Furthermore, their framework supports only references as a form of state. They recognize that for reasoning about low-level models registers should be distinguished from memory locations and mention that this can be emulated by special references that are treated differently. Nevertheless, these references might be accessed from different threads running on different processors. In our model, we introduced a clear distinction between variable that can be accessed by every thread and registers that can only be accessed by one thread. This clear distinction allows us to investigate the differences between local and shared memory more clearly.

Program transformations that establish information flow security have been proposed before, e.g., in [Aga00], [SS00], [Siv06], [KM07]. Many of such 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 so far. More generally, fence insertion has been studied in depth, e.g., in [SS88], [FLM03], [BAM07], [LW11], [KVY12], [LW13]. Many fence insertion techniques aim at establishing sequential consistency. In our transformation, we avoid to establish sequential consistency. The key motivation for relaxing sequential consistency is to gain performance. This gain is lost, if one establishes sequential consistency, despite the weak memory model, by adding fences. To minimize the insertion of fence commands, we guide our transformation by the rules of our security type system.

VIII. 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 the case where one migrates to TSO [VM12], but it was not clear for PSO and IBM370 before. In addition, we showed that one also cannot rely on the preservation of noninterference when one migrates between weak memory models. While it might not be surprising that this can happen if one moves 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 is not yet clear. It would very desirable to impose a taxonomy on memory models with respect to noninterference. All attempts to order these models by the preservation of noninterference so far, have not brought us closer to such a taxonomy.

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.

Acknowledgments.

We thank Barbara Sprick for valuable 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

[AG95] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. western research laboratory. Technical report, Research Report 95/7, 1995.

[AG96] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66 – 76, 1996.

[Aga00] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, USA, 2000. ACM Press.

[Alg10] J. Alglave. *A Shared Memory Poetics*. PhD thesis, Paris Diderot University, 2010.

[Alg12] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.

[AM11] J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, pages 50–66, 2011.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *CAV*, pages 258–272, 2010.

[BAM07] S. Burckhardt, R. Alur, and M. M.K̃. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.

[BC02] G. Boudol and I. Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.

[BPS12] G. Boudol, G. Petri, and B. Serpette. Relaxed Operational Semantics of Concurrent Programming Languages. In Bas Luttik and Michel A. Reniers, editors, Proceedings Combined 19th International Workshop on *Expressiveness in Concurrency* and 9th Workshop on *Structured Operational Semantics,* Newcastle upon Tyne, UK, September 3, 2012, volume 89 of *Electronic Proceedings in Theoretical Computer Science*, pages 19–33. Open Publishing Association, 2012.

[Coh78] E. S. Cohen. Information Transmission in Sequential Programs. *Foundations of Secure Computation*, pages 297–335, 1978. Academic Press.

[Den76] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[FLM03] X. Fang, J. Lee, and S.P̃. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, pages 285–294, 2003.

[GM82] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy (S&P)*, pages 11–20, Oakland, CA, USA, 1982. IEEE Computer Society.

[KM07] Boris Köpf and Heiko Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *International Journal of Information Security*, 6(2–3):107–131, 2007.

[KVY12] M. Kuperstein, M.T̃. Vechev, and E. Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, 2012.

[Lam73] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, pages 690–691, 1979.

[LMP12] A. Lux, H. Mantel, and M. Perner. Scheduler-Independent Declassification. In *Proceedings of the 11th International Conference on Mathematics of Program Construction (MPC)*, LNCS 7342, pages 25–47, Madrid, Spain, 2012. Springer.

[LW11] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*, pages 144–160, 2011.

[LW13] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in pso memory systems. In *TACAS*, pages 339–353, 2013.

[MS10] H. Mantel and H. Sudbrock. Flexible Scheduler-Independent Security. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, LNCS 6345, pages 116–133, Athens, Greece, 2010. Springer.

[MSS11] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 218–232, Cernay-la-Ville, France, 2011. IEEE Computer Society.

[OSS09] 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 (TPHOLs)*, pages 391–407, München, Germany, 2009. Springer.

[RA79] 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 (POPL)*, pages 283–290, San Antonio, TX, USA, 1979.

[RS06] A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 177–189, Venice, Italy, 2006.

[Rus81] J. M. Rushby. Design and Verification of Secure Systems. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asimolar, CA, USA, 1981.

[Sau12] J. Sauer. Informationsflusssicherheit in Systemen mit zwei Prozessoren. Master's thesis, TU Darmstadt, January 2012.

[Siv06] 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 (QAPL)*, number 2 in ENTCS 153, pages 241–257, Edinburgh, UK, 2006.

[Smi01] G. Smith. A New Type System for Secure Information Flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 115–125, Cape Breton, Canada, 2001.

[SS88] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–215, Cambridge, UK, 2000. IEEE Computer Society.

[SS05] A. Sabelfeld and D. Sands. Dimensions and Principles of Declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 255–269, Aix-en-Provence, France, 2005. IEEE Computer Society.

[SSA+11] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. *SIGPLAN Not.*, 46(6):175–186, 2011.

[VM12] J. A. Vaughan and T. Millstein. Secure Information Flow for Concurrent Programs under Total Store Order. In *IEEE Computer Security Foundations Symposium*, pages 19–29, 2012.

[VS98] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW)*, pages 34–43, Rockport, MA, USA, 1998.

[VS99] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. *Journal of Computer Security*, 7(2,3):231–253, 1999.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

APPENDIX

This appendix provides proofs for the main body of this report. We refer to the main body as "the article". We will recall the lemmas and theorems from the article with possibly different numbering scheme. To avoid confusions we will connect the new numbering scheme from the appendix to the numbering scheme of the article when we recall a lemma or theorem.

A. Proofs for Definitions of γ_l and δ_l

In this section we prove properties of the four memory models $\mathcal{MM} = \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$ including the conditions γ_l and δ_l that are defined in Section V of the article.

For easier reference we recall the definitions of δ_l and γ_l for all $l \in \{1, 2, 3\}$:

Definition of δ_l	Definition of γ_l
$\delta_1(\Phi, \Psi) \equiv$ $\forall pa \in Pa.\forall i \in \mathcal{I}.\forall ob \in Ob.\forall m < (\lvert pa\rvert - 1).$ $(next_\Phi(pa, m) \wedge pa[m] = (i, ob))$ $\Rightarrow \left[\begin{array}{l} \forall j \in \mathcal{I}.\forall ob' \in Ob.\forall k < m. \\ pa[k] = (j, ob') \\ \Rightarrow \left[\begin{array}{l} \left(\begin{array}{l} (isRead(ob) \wedge j = i) \\ \Rightarrow \neg isWrite(ob') \end{array}\right) \\ \wedge \left(\begin{array}{l} (isWrite(ob) \wedge j = i) \\ \Rightarrow \neg isWrite(ob') \end{array}\right) \end{array}\right] \end{array}\right]$	$\gamma_1(\Phi, \Psi) \equiv \phi_{\mathrm{WR}} \in \Phi$
$\delta_2(\Phi, \Psi) \equiv$ $\forall pa \in Pa.\forall i \in \mathcal{I}.\forall ob \in Ob.\forall m < (\lvert pa\rvert - 1).$ $(next_\Phi(pa, m) \wedge pa[m] = (i, ob))$ $\Rightarrow \left[\begin{array}{l} \forall j \in \mathcal{I}.\forall ob' \in Ob.\forall k < m. \\ pa[k] = (j, ob') \\ \Rightarrow \left[\begin{array}{l} \left(\begin{array}{l} (ob \in Fe \wedge j = i)) \\ \Rightarrow \neg isWrite(ob') \end{array}\right) \\ \wedge \left(\begin{array}{l} \forall x \in \mathcal{X}.\forall v \in \mathcal{V}.\forall r, r' \in \mathcal{R}. \\ (ob = ?@x \twoheadrightarrow r \wedge j = i) \\ \Rightarrow ob' \neq x \twoheadleftarrow v@r' \end{array}\right) \\ \wedge \left(\begin{array}{l} (isRead(ob) \wedge j = i) \\ \Rightarrow \neg isRead(ob') \end{array}\right) \\ \wedge \left(\begin{array}{l} (isWrite(ob) \wedge j = i) \\ \Rightarrow \neg isWrite(ob') \end{array}\right) \end{array}\right] \end{array}\right]$	$\gamma_2(\Phi, \Psi) \equiv \phi_{\mathrm{WR}} \in \Phi \wedge \phi_{\mathrm{ROwn}} \in \Phi$
$\delta_3(\Phi, \Psi) \equiv$ $\forall pa \in Pa.\forall i \in \mathcal{I}.\forall ob \in Ob.\forall m < (\lvert pa\rvert - 1).$ $(next_\Phi(pa, m) \wedge pa[m] = (i, ob))$ $\Rightarrow \left[\begin{array}{l} \forall j \in \mathcal{I}.\forall ob' \in Ob.\forall k < m. \\ pa[k] = (j, ob') \\ \Rightarrow \left[\begin{array}{l} \left(\begin{array}{l} (isRead(ob) \wedge j = i) \\ \Rightarrow \neg isRead(ob') \end{array}\right) \\ \wedge \left(\begin{array}{l} (isWrite(ob) \wedge j = i) \\ \Rightarrow \neg isWrite(ob') \end{array}\right) \end{array}\right] \end{array}\right]$	$\gamma_3(\Phi, \Psi) \equiv \phi_{\mathrm{WW}} \in \Phi$

Lemma 1. *The two predicates γ_1 and δ_1 are contradictory, i.e. $(\neg\gamma_1(\Phi, \Psi)) \vee (\neg\delta_1(\Phi, \Psi))$ holds.*

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

Lemma 2. *The two predicates γ_2 and δ_2 are contradictory, i.e. $(\neg\gamma_2(\Phi, \Psi)) \vee (\neg\delta_2(\Phi, \Psi))$ holds.*

Proof: We assume that both $\gamma_2(\Phi,\Psi)$ and $\delta_2(\Phi,\Psi)$ hold, and derive a contradiction. We consider the path $pa = [(0, \mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)]::[(0, ?@\mathrm{x} \twoheadrightarrow \mathrm{r}_2)]$. For this path, $\phi_{\mathrm{ROwn}}(pa, 1)$ holds, because $isWrite(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$, $isRead(?@\mathrm{x} \twoheadrightarrow \mathrm{r}_2)$, $\mathrm{r}_1 \neq \mathrm{r}_2$ hold and both obligations access the same variable x (see Figure 2 in the article). From our assumption $\gamma_2(\Phi,\Psi)$, we obtain $\phi_{\mathrm{ROwn}} \in \Phi$. Together, this implies that $next_\Phi(pa, 1)$ holds. From $next_\Phi(pa, 1)$, $pa[1] = (0, 0@? \twoheadrightarrow \mathrm{r}_2)$, $pa[0] = (0, \mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$, $isRead(0@\mathrm{y} \twoheadrightarrow \mathrm{r}_2)$ and our assumption $\delta_2(\Phi,\Psi)$, we can conclude that $(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1) \neq (\mathrm{x} \twoheadleftarrow v@r)$ holds for all $v \in \mathcal{V}$ and $r \in \mathcal{R}$. This is a contradiction, as $0 \in \mathcal{V}$ and $\mathrm{r}_1 \in \mathcal{R}$. ■

Lemma 3. *The two predicates γ_3 and δ_3 are contradictory, i.e. $(\neg\gamma_3(\Phi,\Psi)) \vee (\neg\delta_3(\Phi,\Psi))$ holds.*

Proof: We assume that both $\gamma_1(\Phi,\Psi)$ and $\delta_1(\Phi,\Psi)$ hold, and derive a contradiction. We consider the path $pa = [(0, \mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)]::[(0, \mathrm{y} \twoheadleftarrow 0@\mathrm{r}_2)]$. For this path, $\phi_{\mathrm{WW}}(pa, 1)$ holds, because $isWrite(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$, $isWrite(\mathrm{y} \twoheadleftarrow 0@\mathrm{r}_2)$ and $sinks(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1) \cap sinks(\mathrm{y} \twoheadleftarrow 0@\mathrm{r}_2) = \emptyset$ hold (see Figure 1 in the article). From our assumption $\gamma_3(\Phi,\Psi)$, we obtain $\phi_{\mathrm{WW}} \in \Phi$. Together, this implies that $next_\Phi(pa, 1)$ holds. From $next_\Phi(pa, 1)$, $pa[1] = (0, \mathrm{y} \twoheadleftarrow 0@\mathrm{r}_2)$, $pa[0] = (0, \mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$, $isWrite(\mathrm{y} \twoheadleftarrow 0@\mathrm{r}_2$ and our assumption $\delta_3(\Phi,\Psi)$, we can conclude that $isWrite(\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1)$ does not hold. This is a contradiction, as $\mathrm{x} \twoheadleftarrow 0@\mathrm{r}_1$ is a write obligation. ■

Lemma 4. *The memory model* SC *satisfies* $\delta_1(\mathrm{SC})$, $\delta_2(\mathrm{SC})$ *and* $\delta_3(\mathrm{SC})$.

Proof: From Table II in the article we get $\Phi = \emptyset$. From $\Phi = \emptyset$ we obtain by Figures 1 and 2 in the article and the definition of $next_\Phi$ that $next_\Phi(pa, k)$ for $pa[k] = (i, ob)$ can only evaluate to true, if $i \neq j$ holds for $pa[m] = (j, ob')$. From this we conclude that

- $next_\Phi(pa, k) \wedge isRead(ob) \wedge j = i \implies \neg isWrite(ob')$ holds for $pa[k] = (i, ob)$ and $pa[m] = (j, ob')$ for all $m < k$, and
- $next_\Phi(pa, k) \wedge ob \in Fe \wedge j = i \implies \neg isWrite(ob')$ holds for $pa[k] = (i, ob)$ and $pa[m] = (j, ob')$ for all $m < k$, and
- $next_\Phi(pa, k) \wedge pa[k] = (i, ?@x \twoheadrightarrow r) \implies pa[m] \neq (i, x \twoheadleftarrow v@r')$ holds for all $x \in \mathcal{X}$, $v \in \mathcal{V}$, , $r, r' \in \mathcal{R}$ and $m < k$, and
- $next_\Phi(pa, k) \wedge isWrite(ob) \wedge j = i \implies \neg isWrite(ob')$ holds for $pa[k] = (j, ob)$ and $pa[m] = (j, ob')$ for all $m < k$, and
- $next_\Phi(pa, k) \wedge isRead(ob) \wedge j = i \implies \neg isRead(ob')$ holds for $pa[k] = (j, ob)$ and $pa[m] = (j, ob')$ for all $m < k$.

Hence, $\delta_1(\mathrm{SC})$, $\delta_2(\mathrm{SC})$ and $\delta_3(\mathrm{SC})$ holds. ■

Lemma 5. *The memory model* IBM370 *satisfies* $\gamma_1(\mathrm{IBM370})$, $\delta_2(\mathrm{IBM370})$ *and* $\delta_3(\mathrm{IBM370})$.

Proof: From Table II in the article follows directly that $\phi_{\mathrm{WR}} \in \Phi$ holds for IBM370.

From Table II in the article we get $\Phi = \{\phi_{\mathrm{WR}}\}$. From $\Phi = \{\phi_{\mathrm{WR}}\}$ we obtain by Figures 1 and 2 in the article and the definition of $next_\Phi$ that $next_\Phi(pa, k)$ can only evaluate to true, if one of two conditions holds for each position m with $m < k$: Either $pa[k] = (i, ob)$ and $pa[m] = (j, ob')$ with $i \neq j$ holds, or $pa[k] = (i, ob)$ and $pa[m] = (j, ob')$ with $isRead(ob)$, $isWrite(ob')$ and $sources(ob) \cap sinks(ob') = \emptyset$ and $sinks(ob) \cap sources(ob') = \emptyset$ holds. From this we conclude that

- $next_\Phi(pa, k) \wedge ob \in Fe \wedge j = i \implies \neg isWrite(ob')$ holds for $pa[k] = (i, ob)$ and $pa[m] = (j, ob')$ for all $m < k$, and
- $next_\Phi(pa, k) \wedge pa[k] = (i, ?@x \twoheadrightarrow r) \implies pa[m] \neq (i, x \twoheadleftarrow v@r')$ holds for all $x \in \mathcal{X}$, $v \in \mathcal{V}$, $r, r' \in \mathcal{R}$ and $m < k$, and
- $next_\Phi(pa, k) \wedge isWrite(ob) \wedge j = i \implies \neg isWrite(ob')$ holds for $pa[k] = (j, ob)$ and $pa[m] = (j, ob')$ for all $m < k$, and
- $next_\Phi(pa, k) \wedge isRead(ob) \wedge j = i \implies \neg isRead(ob')$ holds for $pa[k] = (j, ob)$ and $pa[m] = (j, ob')$ for all $m < k$.

Hence, $\delta_2(\mathrm{IBM370})$ and $\delta_3(\mathrm{IBM370})$ holds. ■

Lemma 6. *The memory model* TSO *satisfies* $\gamma_1(\mathrm{TSO})$, $\gamma_2(\mathrm{TSO})$ *and* $\delta_3(\mathrm{TSO})$.

Proof: From Table II in the article follows directly that $\phi_{\mathrm{WR}} \in \Phi$ and $\phi_{\mathrm{ROwn}} \in \Phi$ hold for TSO. Hence, $\gamma_1(\mathrm{TSO})$ and $\gamma_2(\mathrm{TSO})$.

From Table II in the article we get $\Phi = \{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}\}$. From $\Phi = \{\phi_{\mathrm{WR}}, \phi_{\mathrm{ROwn}}\}$ we obtain by Figures 1 and 2 in the article and the definition of $next_\Phi$ that $next_\Phi(pa, k)$ can only evaluate to true, if one of two conditions holds for each position m with $m < k$: Either $pa[k] = (i, ob)$ and $pa[m] = (j, ob')$ with $i \neq j$ holds, or $pa[k] = (i, ob)$ and $pa[m] = (j, ob')$ with $isRead(ob)$ and $isWrite(ob')$ holds. From this we conclude that

- $next_\Phi(pa, k) \wedge isWrite(ob) \wedge j = i \implies \neg isWrite(ob')$ holds for $pa[k] = (j, ob)$ and $pa[m] = (j, ob')$ for all $m < k$, and

- $next_\Phi(pa,k) \wedge isRead(ob) \wedge j = i \implies \neg isRead(ob')$ holds for $pa[k] = (j, ob)$ and $pa[m] = (j, ob')$ for all $m < k$.

Hence, $\delta_3(\text{TSO})$ holds. ■

Lemma 7. *The memory model* PSO *satisfies* $\gamma_1(\text{PSO})$, $\gamma_2(\text{PSO})$ *and* $\gamma_3(\text{PSO})$.

Proof: From Table II in the article follows directly that $\phi_{\text{WR}} \in \Phi$, $\phi_{\text{ROwn}} \in \Phi$ and $\phi_{\text{WW}} \in \Phi$ hold for PSO. Hence, $\gamma_1(\text{PSO})$, $\gamma_2(\text{PSO})$ and $\gamma_3(\text{PSO})$ hold. ■

Corollary 1. *The properties discriminate the memory models* SC, IBM370, TSO *and* PSO, *i.e. the following three propositions hold:*

- $\delta_1(\text{SC})$ *and* $\neg\gamma_1(\text{SC})$ *while* $\neg\delta_1(MM)$ *and* $\gamma_1(MM)$ *for all* $MM \in \{\text{IBM370}, \text{TSO}, \text{PSO}\}$,
- $\delta_2(MM)$ *and* $\neg\gamma_2(MM)$ *for* $MM \in \{\text{SC}, \text{IBM370}\}$ *while* $\neg\delta_2(MM)$ *and* $\gamma_2(MM)$ *for all* $MM \in \{\text{TSO}, \text{PSO}\}$,
- $\delta_3(MM)$ *and* $\neg\gamma_3(MM)$ *for* $MM \in \{\text{SC}, \text{IBM370}, \text{TSO}\}$ *while* $\neg\delta_3(\text{PSO})$ *and* $\gamma_3(\text{PSO})$.

Proof: This follows immediately from Lemmas 4, 5, 6 and 7 that show which memory model satisfies which condition and Lemma 1, 2 and 3 that show that γ_l and δ_l for each $l \in \{1, 2, 3\}$ are contradictory. ■

Lemma 8. *For each* $MM \in \mathcal{MM}$ *the following proposition holds:*
$next_\Phi(pa,k) \wedge pa[k] = (i, ob) \wedge pa[m] = (i, ob') \wedge isWrite(ob) \wedge isWrite(ob') \implies sinks(ob) \cap sinks(ob') = \emptyset$ *for all* $pa \in Pa$, $i \in \mathcal{I}$, $ob, obs' \in Ob$, $k < (|pa| - 1)$ *and* $m < k$.

Proof: For $MM \in \{\text{SC}, \text{IBM370}, \text{TSO}\}$, the proposition follows directly from $\delta_3(MM)$ (Corollary 1), because the left side of the implication cannot be fulfilled.

For $MM = \text{PSO}$ we get from Table II in the article that $\Phi = \{\phi_{\text{WR}}, \phi_{\text{WW}}, \phi_{\text{ROwn}}\}$. The only possibility to fulfill the left side of the implication is with ϕ_{WW}, because ϕ_{WR} and ϕ_{ROwn} both require that $isRead(ob)$ and $isWrite(ob')$ holds for $pa[k] = (i, ob)$ and $pa[m] = (i, ob')$ according to Figure 2 in the article. From Figure 2 in the article we know that ϕ_{WW} only evaluates to true if $sinks(ob) \cap sinks(ob') = \emptyset$ holds for $pa[k] = (i, ob)$ and $pa[m] = (i, ob')$. Hence, the proposition holds for $MM = \text{PSO}$. ■

B. Proofs for Comparing Weak Memory Models wrt. Noninterference

In this section we prove that the programs programs in Section V of the article discriminate the memory models with respect to noninterference.

We split each of the three Lemmas 1-3 from the article into 4 Lemmas, one Lemma for each implication. The correspondence between the Lemmas is as follows:

Lemma in the article	Lemmas in this Document
Lemma 1	Lemma 9, 10, 11, 12
Lemma 2	Lemma 13, 14, 15, 16
Lemma 3	Lemma 17, 18, 19, 20

For easier reference we recall the definitions of c_1^+ and c_1^- in Figure 1.

```
c1+ :=
store1 x 1; store2 y 1; store3 z 0; store4 l 0;
spawn5(
 store6 x 0; load7 r2 y; load8 r3 z;
 and9 r4 r2 r3; load10 r5 h;
 if11 r4
 then if12 r5 then store13 l 5 else skip14 fi
 else if15 r5 then skip16 else store17 l 5 fi fi);
store18 y 0; load19 r1 x; store20 z r1; store21 z 0
```

```
c1- :=
store1 x 1; store2 y 1; store3 z 0;
spawn4(
 store5 x 0; load6 r2 y; load7 r3 z; and8 r4 r2 r3;
 if9 r4 then load10 r5 h; store11 l r5 else skip12 fi);
store13 y 0; load14 r1 x; store15 z r1
```

Figure 1. Programs for Lemmas 9, 10, 11, 12 (Lemma 1 in the article)

Lemma 9. *The following proposition holds for the domain assignment* lev *with* $lev(\text{h})$ *and* $lev(x) = \textbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\text{h}\}$*:* $\gamma_1(MM) \implies \text{c}_1^+ \in NI_{MM}$.

Proof: Only the final value of the **Low**-variable l can depend on the initial value of a **High**-variable, because the only other variables that are updated are the variables x, y and z, and the final value of x, y and z is definitely 0. Independent of the

initial memory, and in particular independent of the initial value of h, the final value of l can be either 0 or 5. Consequently, the program c_1^+ satisfies MM-Noninterference. In the following we present the arguments in detail.

The final value of x and y is definitely 0, because the obligations of $\mathbf{store}_6\ \mathrm{x}\ 0$ and $\mathbf{store}_{18}\ \mathrm{y}\ 0$ respectively cause the last updates of x, and y, and both updates set their respective variable to 0. The obligations of $\mathbf{store}_6\ \mathrm{x}\ 0$ and $\mathbf{store}_{18}\ \mathrm{y}\ 0$ cause the last updates of x and y, because only the obligations of $\mathbf{store}_1\ \mathrm{x}\ 1$ and $\mathbf{store}_2\ \mathrm{y}\ 1$ cause further updates of x and y, and these two obligations must be fulfilled before the obligation of $\mathbf{spawn}_5$ while the obligations of $\mathbf{store}_6\ \mathrm{x}\ 0$ and $\mathbf{store}_{18}\ \mathrm{y}\ 0$ must be fulfilled after the obligation of $\mathbf{spawn}_5$. The final value of z is definitely 0, because the obligation of $\mathbf{store}_{21}\ \mathrm{z}\ 0$ causes the last update of z, because only the obligations of $\mathbf{store}_3\ \mathrm{z}\ 0$ and $\mathbf{store}_{20}\ \mathrm{z}\ \mathrm{r}_1$ cause further updates of z, and these two obligations must be fulfilled before the obligation of $\mathbf{store}_{21}\ \mathrm{z}\ 0$. The obligations of $\mathbf{store}_3\ \mathrm{z}\ 0$ and $\mathbf{store}_{20}\ \mathrm{z}\ \mathrm{r}_1$ must be fulfilled before the obligation of $\mathbf{store}_{21}\ \mathrm{z}\ 1$, because the obligations of $\mathbf{store}_3\ \mathrm{z}\ 0$ and $\mathbf{store}_{20}\ \mathrm{z}\ \mathrm{r}_1$ are caused by the same thread and access the same variable as $\mathbf{store}_{21}\ \mathrm{z}\ 1$ (Lemma 8).

The final value of h is in $\{0, 5\}$, because the obligation of $\mathbf{store}_4\ \mathrm{l}\ 0$ is definitely processed in every program run and the only other updates of l are $\mathbf{store}_{13}\ \mathrm{l}\ 5$ and $\mathbf{store}_{17}\ \mathrm{l}\ 5$.

We show that the final value of l can always be 0. Let mem_1 be arbitrary. We distinguish two cases based on the initial value of h.

Case (Initial value of h equals 0):

Let $t = 1, 2, 3, 4, 5, \mathbf{19}, 6, 7, \mathbf{18}, 20, 8, 9, 10, 21$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by the sequence, because of four reasons. First, all obligations, except the obligations of $\mathbf{load}_{19}\ \mathrm{r}_1\ \mathrm{x}$ and $\mathbf{store}_{18}\ \mathrm{y}\ 0$, are fulfilled in the order in which their instructions are executed. Second, $\gamma_1(MM)$ holds and therefore $\phi_{\mathrm{WR}} \in \Phi$ for MM. The obligation ob of $\mathbf{store}_{18}\ \mathrm{y}\ 0$ may be fulfilled after the obligation ob' of $\mathbf{load}_{19}\ \mathrm{r}_1\ \mathrm{x}$ due to ϕ_{WR}, because $isWrite(ob)$ evaluates to true, $isRead(ob')$ evaluates to true, $sources(ob) \cap sinks(ob') = \emptyset$ and $sinks(ob) \cap sources(ob') = \emptyset$ holds. Third, the $\mathbf{then}$-branch of $\mathbf{if}_{11}$ is taken, and, fourth, the $\mathbf{else}$-branch of $\mathbf{if}_{12}$ is taken.

The $\mathbf{then}$-branch of $\mathbf{if}_{11}\ \mathrm{r}_4$ is taken, because r_4 is 1 when executing $\mathbf{if}_{11}$. The value of r_4 is 1, because obligation of $\mathbf{and}_9\ \mathrm{r}_4\ \mathrm{r}_2\ \mathrm{r}_3$ causes the most recent update of r_4 and the value of r_2 and r_3 is 1 when executing $\mathbf{and}_9\ \mathrm{r}_4\ \mathrm{r}_2\ \mathrm{r}_3$. The value of r_2 is 1, because the obligation of $\mathbf{load}_7\ \mathrm{r}_2\ \mathrm{y}$ causes the most recent update of r_2 and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_2\ \mathrm{y}\ 1$ causes the most recent update of y and the update sets y to 1. The value of r_3 is 1, because the obligation of $\mathbf{load}_8\ \mathrm{r}_3\ \mathrm{z}$ causes the most recent update of r_3 and the update sets r_3 to the value of z. The value of z is 1, because the obligation of $\mathbf{store}_{20}\ \mathrm{z}\ \mathrm{r}_1$ causes the most recent update of z and the update sets z to the value of r_1. The value of r_1 is 1, because the obligation of $\mathbf{load}_{19}\ \mathrm{r}_1\ \mathrm{x}$ causes the most recent update of r_1 and the update sets r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1\ \mathrm{x}\ 1$ causes the most recent update of x and the update sets x to 1.

The $\mathbf{else}$-branch of $\mathbf{if}_{12}\ \mathrm{r}_5$ is taken, because r_5 is 0 when executing $\mathbf{if}_{12}$. The value of r_5 is 0, because the obligation of $\mathbf{load}_{10}\ \mathrm{r}_5\ \mathrm{h}$ causes the most recent update of r_5 and the update sets r_5 to the value of h. The value of h is 0, because there is no update of h in the program and the initial value of h is 0 by assumption of this case.

Since the $\mathbf{else}$-branch of $\mathbf{if}_{12}$ is taken the only obligation that causes an update of l is the obligation of $\mathbf{store}_4\ \mathrm{l}\ 0$. Thus the final value of l is 0.

Case (Initial value of h does not equal 0):

Let $t = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 19, 20, 21$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by the sequence, because of three reasons. First, all obligations are fulfilled in the order in which their instructions are executed. Second, the $\mathbf{else}$-branch of $\mathbf{if}_{11}$ is taken and, third, the $\mathbf{then}$-branch of $\mathbf{if}_{15}$ is taken.

The $\mathbf{else}$-branch of $\mathbf{if}_{11}\ \mathrm{r}_4$ is taken, because r_4 is 0 when executing $\mathbf{if}_{11}$. The value of r_4 is 0, because the obligation of $\mathbf{and}_9\ \mathrm{r}_4\ \mathrm{r}_2\ \mathrm{r}_3$ causes the most recent update of r_4 and the value of r_3 is 0 when executing $\mathbf{and}_9\ \mathrm{r}_4\ \mathrm{r}_2\ \mathrm{r}_3$. The value of r_3 is 0, because the obligation of $\mathbf{load}_8\ \mathrm{r}_3\ \mathrm{z}$ causes the most recent update of r_3 and the update sets r_3 to the value of z. The value of z is 0, because the obligation of $\mathbf{store}_3\ \mathrm{z}\ 0$ causes the most recent update of z and the update sets z to 0.

The $\mathbf{then}$-branch of $\mathbf{if}_{15}\ \mathrm{r}_5$ is taken, because the value of r_5 is not 0 when executing $\mathbf{if}_{15}$. The value of r_5 is not 0, because the obligation of $\mathbf{load}_{10}\ \mathrm{r}_5\ \mathrm{h}$ causes the most recent update of r_5 and the update sets r_5 to the value of h. The value of h is not 0, because there is no update of h in the program and the initial value of h is different from 0 by assumption of this case.

Since the $\mathbf{then}$-branch of $\mathbf{if}_{15}$ is taken, the only update of l is the event of $\mathbf{store}_4\ \mathrm{l}\ 0$. Thus the final value of l is 0.

We show that the final value of l can always be 5. Let mem_1 be arbitrary. We distinguish two cases based on the initial value of h.

Case (Initial value of h equals 0):

Let $t = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 18, 19, 20, 21$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by the sequence, because of three

reasons. First, all obligations are fulfilled in the order in which their instructions are executed. Second, the **else**-branch of $\mathbf{if}_{11}$ is taken and, third, the **else**-branch of $\mathbf{if}_{15}$ is taken.
The **else**-branch of $\mathbf{if}_{11}$ r_4 is taken, because r_4 is 0 when executing $\mathbf{if}_{11}$. The value of r_4 is 0, because the obligation of $\mathbf{and}_9$ r_4 r_2 r_3 causes the most recent update of r_4 and the value of r_3 is 0 when executing $\mathbf{and}_9$ r_4 r_2 r_3. The value of r_3 is 0, because the obligation of $\mathbf{load}_8$ r_3 z causes the most recent update of r_3 and the update sets r_3 to the value of z. The value of z is 0, because the obligation of $\mathbf{store}_3$ z 0 causes the most recent update of z and the update sets z to 0.
The **else**-branch of $\mathbf{if}_{15}$ r_5 is taken, because r_5 is 0 when executing $\mathbf{if}_{15}$. The value of r_5 is 0, because the obligation of $\mathbf{load}_{10}$ r_5 h causes the most recent update of r_5 and the update sets r_5 to the value of h. The value of h is 0, because there is no update of h in the program and the initial value of h is 0 by assumption of this case.
Since the **else**-branch of $\mathbf{if}_{15}$ is taken, the obligation of $\mathbf{store}_{17}$ l 5 is fulfilled as the last update of l. Thus the final value of l is 5.

Case (Initial value of h does not equal 0):

Let $t = 1, 2, 3, 4, 5, \mathbf{19}, 6, 7, \mathbf{18}, 20, 8, 9, 10, 13, 21$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by the sequence, because of four reasons. First, all obligations, except the obligations of $\mathbf{load}_{19}$ r_1 x and $\mathbf{store}_{18}$ y 0, are fulfilled in the order in which their instructions are executed. Second, $\gamma_1(MM)$ holds and therefore $\phi_{\mathrm{WR}} \in \Phi$ for MM. The obligation ob of $\mathbf{store}_{18}$ y 0 may be fulfilled after the obligation ob' of $\mathbf{load}_{19}$ r_1 x due to ϕ_{WR}, because $isWrite(ob)$ evaluates to true, $isRead(ob')$ evaluates to true, $sources(ob) \cap sinks(ob') = \emptyset$ and $sinks(ob) \cap sources(ob') = \emptyset$ holds. Third, the **then**-branch of $\mathbf{if}_{11}$ is taken, and, fourth, the **then**-branch of $\mathbf{if}_{12}$ is taken.
The **then**-branch of $\mathbf{if}_{11}$ r_4 is taken, because r_4 is 1 when executing $\mathbf{if}_{11}$. The value of r_4 is 1, because obligation of $\mathbf{and}_9$ r_4 r_2 r_3 causes the most recent update of r_4 and the value of r_2 and r_3 is 1 when executing $\mathbf{and}_9$ r_4 r_2 r_3. The value of r_2 is 1, because the obligation of $\mathbf{load}_7$ r_2 y causes the most recent update of r_2 and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_2$ y 1 causes the most recent update of y and the update sets y to 1. The value of r_3 is 1, because the obligation of $\mathbf{load}_8$ r_3 z causes the most recent update of r_3 and the update sets r_3 to the value of z. The value of z is 1, because the obligation of $\mathbf{store}_{20}$ z r_1 causes the most recent update of z and the update sets z to the value of r_1. The value of r_1 is 1, because the obligation of $\mathbf{load}_{19}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of x and the update sets x to 1.
The **then**-branch of $\mathbf{if}_{12}$ r_5 is taken, because the value of r_5 is not 0 when executing $\mathbf{if}_{12}$. The value of r_5 is not 0, because the obligation of $\mathbf{load}_{10}$ r_5 h causes the most recent update of r_5 and the update sets r_5 to the value of h. The value of h is not 0, because there is no update of h in the program and the initial value of h is different from 0 by assumption of this case.
Since the **then**-branch of $\mathbf{if}_{12}$ is taken, the obligation of $\mathbf{store}_{13}$ l 5 is fulfilled as the last update of l. Thus the final value of l is 5.

That means independent of the initial value of the **High**-variable h, the final values of x, y and z are 0, the final value of l can be either 0 or 5, and the values of all other variables remain unchanged. Hence, for all pairs of **Low**-equal initial memories **Low**-equal final memories are reachable. Consequently, the program c_1^+ satisfies MM-Noninterference, if $\gamma_1(MM)$ holds. ■

Lemma 10. *The following proposition holds for the domain assignment lev with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\delta_1(MM) \implies \mathrm{c}_1^+ \notin NI_{MM}$.

Proof: The final value of the **Low**-variable l can only be 5 if the initial value of the **High**-variable h is 0, because $\mathbf{store}_4$ l 0 definitely updates l to 0 and there are only two instructions that can update l to 5. The two instructions are $\mathbf{store}_{13}$ l 5 and $\mathbf{store}_{17}$ l 5. The instruction $\mathbf{store}_{13}$ l 5 is in the **then**-branch of $\mathbf{if}_{11}$. The **then**-branch of $\mathbf{if}_{11}$ is dead code. The instruction $\mathbf{store}_{17}$ l 5 is in the **else**-branch of $\mathbf{if}_{15}$. The **else**-branch of $\mathbf{if}_{15}$ is only reachable if the initial value of h is 0. Thus the program does not satisfy MM-Noninterference. In the following we present the arguments in detail.

The **else**-branch of $\mathbf{if}_{15}$ r_5 is only reachable if the initial value of h is 0, because the obligation of $\mathbf{load}_{10}$ r_5 h is fulfilled before $\mathbf{if}_{15}$ is executed, the obligation of $\mathbf{load}_{10}$ r_5 h causes an update of r_5 to the initial value of h, and there is no other update of r_5. The obligation of $\mathbf{load}_{10}$ r_5 h causes an update of r_5 to the initial value of h, because there is no update of h in the program.

The **then**-branch of $\mathbf{if}_{11}$ is dead code, because the value of r_4 is always 0. The value of r_4 is always 0, because it is initialized with 0 and only the obligation of $\mathbf{and}_9$ r_4 r_2 r_3 causes an update of r_4. The obligation of $\mathbf{and}_9$ r_4 r_2 r_3 causes an update of r_4 to 0, because either r_2 or r_3 is always 0. This is due to the fact that r_2 and r_3 are initialized with 0 and only the obligations of $\mathbf{load}_7$ r_2 y and $\mathbf{load}_8$ r_3 z respectively cause an update of r_2 and r_3. At least one of the obligations causes an update of its respective register to 0. Which of the two instructions updates its register to 0 depends on the obligation that is fulfilled directly after the obligation of $\mathbf{spawn}_5$. Either the obligation of $\mathbf{store}_6$ x 0 or the obligation of $\mathbf{store}_{18}$ y 0 must be fulfilled directly after the obligation of $\mathbf{spawn}_5$. This is due to the fact that these two instructions belong to different threads and the obligations of their subsequent instructions cannot be fulfilled before the obligations of these two instructions. The subsequent instruction of $\mathbf{store}_6$ x 0 is $\mathbf{load}_7$ r_2 y and $isRead$ holds for the obligation of $\mathbf{load}_7$ r_2 y. Since $\delta_1(MM)$ holds, $next_\Phi$ cannot hold for the

obligation of $\mathbf{load}_7$ r_2 y, because *isWrite* holds for the obligation of $\mathbf{store}_6$ x 0. The subsequent instruction of $\mathbf{store}_{18}$ y 0 is $\mathbf{load}_{19}$ r_1 x and *isRead* holds for the obligation of $\mathbf{load}_{19}$ r_1 x. Since $\delta_1(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{load}_{19}$ r_1 x, because *isWrite* holds for the obligation of $\mathbf{store}_{18}$ y 0.

We distinguish two cases based on the obligation that is fulfilled after the obligation of $\mathbf{spawn}_5$ and show that either r_2 or r_3 is 0.

Case ($\mathbf{store}_6$ x 0):

In this case, the value of r_3 is always 0, because r_3 is initialized with 0, only the obligation of $\mathbf{load}_8$ r_3 z causes an update of r_3 and the update caused by the obligation sets r_3 to the value of z. The value of z is 0, because the obligation of either $\mathbf{store}_3$ z 0 or $\mathbf{store}_{21}$ z 0 or $\mathbf{store}_{20}$ z r_1 causes the most recent update of z. All three obligations update z to 0. The obligation of $\mathbf{store}_{20}$ z r_1 causes an update of z to 0, because the obligation of $\mathbf{load}_{19}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The value of x is 0, because the obligation of $\mathbf{store}_6$ x 0 causes the most recent update of x and the update sets x to 0. The obligation of $\mathbf{store}_6$ x 0 causes the most recent update of x for $\mathbf{load}_{19}$ r_1 x, because both obligations must be fulfilled after the obligation of $\mathbf{spawn}_5$, the obligation of $\mathbf{store}_6$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_5$ due to the assumption of this case, and there is no other obligation that causes an update of x that can be fulfilled after the obligation of $\mathbf{spawn}_5$.

Case ($\mathbf{store}_{18}$ y 0):

In this case, the value of r_2 is always 0, because r_2 is initialized with 0, only the obligation of $\mathbf{load}_7$ r_2 y causes an update of r_2, and the update caused by the obligation sets r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_{18}$ y 0 causes the most recent update of y and the update sets y to 0. The obligation of $\mathbf{store}_{18}$ y 0 causes the most recent update for the obligation of $\mathbf{load}_7$ r_2 y, because both obligations are fulfilled after the obligation of $\mathbf{spawn}_5$, the obligation of $\mathbf{store}_{18}$ y 0 is fulfilled directly after $\mathbf{spawn}_5$ due to the assumption of this case, and there is no other obligation that causes an update of y that can be fulfilled after the obligation of $\mathbf{spawn}_5$.

Based on these observations we construct a concrete counter example: We choose initial memories mem_1 and mem_1' such that $mem_1(x) = 0$ for all $x \in \mathcal{X}$, $mem_1'(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $mem_1'(\mathrm{h}) = 23$. The memories mem_1 and mem_1' satisfy $mem_1 =_L mem_1'$, because $mem_1(x) = 0 = mem_1'(x)$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $lev(\mathrm{h}) = \mathbf{High}$. From mem_1, a final memory mem_2 is reachable with $mem_2(\mathrm{l}) = 5$, because the initial value of h is 0. All final memories mem_2' that are reachable from mem_1' satisfy $mem_2'(\mathrm{l}) \neq 5$, because the initial value of h is 23 and not 0. Thus, for all final memories mem_2' that are reachable from mem_1', we have $mem_2 \neq_L mem_2'$, because $mem_2(\mathrm{l}) = 5 \neq mem_2'(\mathrm{l})$ and $lev(\mathrm{l}) = \mathbf{Low}$.

Consequently, the program c_1^+ does not satisfy MM-Noninterference, if $\delta_1(MM)$. ■

Lemma 11. *The following proposition holds for the domain assignment lev with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\gamma_1(MM) \implies \mathrm{c}_1^- \notin NI_{MM}$.

Proof: The two instructions $\mathbf{load}_{10}$ and $\mathbf{store}_{11}$ in the **then**-branch of $\mathbf{if}_9$ form a direct leak, because $\mathbf{load}_{10}$ reads the value of the **High**-variable h into r_5 and $\mathbf{store}_{11}$ subsequently writes the value of r_5 into the **Low**-variable l. Due to this direct leak and the fact that this **then**-branch is reachable, the program c_1^- does not satisfy MM-noninterference. In the following we present the arguments in detail.

Let $t = 1, 2, 3, 4, \mathbf{14}, 5, 6, \mathbf{13}, 15, 7, 8$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers up to reaching $\mathbf{if}_9$ in the order given by t, because all obligations, except the obligations of $\mathbf{store}_{13}$ y 0 and $\mathbf{load}_{14}$ x r_1 are fulfilled in the order in which their instructions are executed. Since $\gamma_1(MM)$ holds, $\phi_{\mathrm{WR}} \in \Phi$ for MM. Thus, the obligation ob of $\mathbf{store}_{13}$ y 0 may be fulfilled after the obligation ob' of $\mathbf{load}_{14}$ x r_1 due to ϕ_{WR}, because $isWrite(ob)$ evaluates to true, $isRead(ob')$ evaluates to true, $sources(ob) \cap sinks(ob') = \emptyset$ and $sinks(ob) \cap sources(ob') = \emptyset$ holds. We will now argue why the instructions $\mathbf{load}_{10}$ and $\mathbf{store}_{11}$ will be executed and their obligations will get fulfilled after t.

The sequence t always leads to a register state reg_9 with $reg_9(\mathrm{r}_2) = 1$, $reg_9(\mathrm{r}_3) = 1$ and $reg_9(\mathrm{r}_4) = 1$, because of three reasons. First, the obligation of $\mathbf{load}_6$ r_2 y causes the last update of r_2 and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_2$ y 1 causes the most recent update of y and the update sets y to 1. Second, the obligation of $\mathbf{load}_7$ r_3 z causes the last update of r_3 and the update sets r_3 to the value of z. The value of z is 1, because the obligation of $\mathbf{store}_{15}$ z r_1 causes the most recent update of z and the update sets z to the value of r_1. The value of r_1 is 1, because the obligation of $\mathbf{load}_{14}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of x and the update sets x to 1. Thus the obligation of $\mathbf{load}_7$ r_3 z causes an update of r_3 to 1. Third, the obligation of $\mathbf{and}_8$ r_4 r_2 r_3 causes the last update of r_4. The obligation of $\mathbf{and}_8$ r_4 r_2 r_3 causes an update of r_4 to 1, because the obligations of $\mathbf{load}_6$ r_2 y and $\mathbf{load}_7$ r_3 z respectively cause the most recent updates of r_2 and r_3, and the updates caused by these two obligations set their respective registers to 1 (as we have argued before). Consequently, the **then**-branch of $\mathbf{if}_9$ r_4 is taken after the sequence t. This means that the instructions $\mathbf{load}_{10}$ r_5 h and $\mathbf{store}_{11}$ l r_5 are executed and their obligations are fulfilled after t.

We choose an initial memory mem_1 with $mem_1(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $mem_1(\mathrm{h}) = 5$. The final value of l is 5, because the obligation of $\mathbf{store}_{11}$ l r_5 causes an update of l to the value of r_5. The value of r_5 is 5, because the obligation of $\mathbf{load}_{10}$ r_5 h causes an update of r_5 to the value of h. The value of h is 5, because there is no update to h in the program and the initial value of h is 5, i.e. $mem_1(\mathrm{h}) = 5$, according to the initial memory that we have chosen. The obligation of $\mathbf{load}_{10}$ r_5 h must be fulfilled before the obligation of $\mathbf{store}_{11}$ l r_5, because both obligations are caused by the same thread and access r_5.

We now show that there is an initial memory mem_1' with $mem_1 =_L mem_1'$ for which 5 is not a possible final value for l. We choose an initial value mem_1' with $mem_1'(x) = 0$ for all $x \in \mathcal{X}$. From the definition of **Low**-equality we know that $mem_1 =_L mem_1'$, because $mem_1(x) = mem_1'(x)$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $lev(\mathrm{h}) = \mathbf{High}$. For the initial memory mem_1' the final value of l must be in $\{0, 1\}$, because the initial value of all variables is 0, all constants that appear in the program are either 0 or 1, and the only computation, i.e. **and** has $\{0, 1\}$ as range of values. Consequently, the final value of l cannot be 5. Hence, $mem_2'(\mathrm{l}) \neq 5 = mem_2(\mathrm{l})$ holds for all final memories mem_2' that are reachable from mem_1'.

This means that there is no final memory reachable from mem_1' that is **Low**-equal to mem_2. Consequently, the program c_1^- does not satisfy MM-Noninterference, if $\gamma_1(MM)$ holds. ■

Lemma 12. *The following proposition holds for the domain assignment lev with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\delta_1(MM) \implies \mathrm{c}_1^- \in NI_{MM}$.

Proof: Only $\mathbf{load}_{10}$ in the **then**-branch of $\mathbf{if}_9$ reads a **High**-variable. The **then**-branch of $\mathbf{if}_9$ is dead code. Since the only instruction that reads a **High**-variable is dead code, the program c_1^- satisfies MM-noninterference. In the following we present the arguments in detail.

The instruction $\mathbf{load}_{10}$ is dead code, because it is in the **then**-branch of $\mathbf{if}_9$ r_4 and the value of r_4 of the spawned thread is always 0. The value of r_4 is always 0, because it is initialized with 0, only the obligation of $\mathbf{and}_8$ r_4 r_2 r_3 updates r_4 and the update sets r_4 to 0. The update caused by the obligation of $\mathbf{and}_8$ r_4 r_2 r_3 sets r_4 to 0, because the value of either r_2 or r_3 is 0 in all possible sequences for fulfilling obligations when executing $\mathbf{and}_8$.

We now show that the value of either r_2 or r_3 is always 0 in all possible sequences for fulfilling obligations. All possible sequences for fulfilling obligations up to (inclusively) the obligation of $\mathbf{spawn}_4$ have the same effect on the global state, because each of the obligations of $\mathbf{store}_1$ x 1, $\mathbf{store}_2$ y 1 and $\mathbf{store}_3$ z 0 causes an update of a different variable, and all of these three obligations must be fulfilled before the obligation of $\mathbf{spawn}_4$ is fulfilled. Only two obligations can be fulfilled directly after the obligation of $\mathbf{spawn}_4$: Either the obligation of $\mathbf{store}_5$ x 0 or the obligation of $\mathbf{store}_{13}$ y 0 must be fulfilled directly after the obligation of $\mathbf{spawn}_4$. This is due to the fact that these two instructions belong to different threads and the obligations of their subsequent instructions cannot be fulfilled before the obligations of these two instructions. The subsequent instruction of $\mathbf{store}_5$ x 0 is $\mathbf{load}_6$ r_2 y and *isRead* holds for the obligation of $\mathbf{load}_6$ r_2 y. Since $\delta_1(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{load}_6$ r_2 y, because *isWrite* holds for the obligation of $\mathbf{store}_5$ x 0. The subsequent instruction of $\mathbf{store}_{13}$ y 0 is $\mathbf{load}_{14}$ r_1 x and *isRead* holds for the obligation of $\mathbf{load}_{14}$ r_1 x. Since $\delta_1(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{load}_{14}$ r_1 x, because *isWrite* holds for the obligation of $\mathbf{store}_{13}$ y 0.

We distinguish two cases. In the first case, the obligation of $\mathbf{store}_5$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$. In the second case, the obligation of $\mathbf{store}_{13}$ y 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$.

Case ($\mathbf{store}_5$ x 0):

In this case, the value of r_3 is always 0, because r_3 is initialized with 0 and the only update of r_3 is $\mathbf{load}_7$ r_3 z. The obligation of $\mathbf{load}_7$ r_3 z always causes an update of r_3 to 0. We now show why this holds: The only remaining update of z after fulfilling the obligation of $\mathbf{spawn}_4$ is $\mathbf{store}_{15}$ z r_1. We distinguish two cases based on the order in which the obligations of $\mathbf{store}_{15}$ z r_1 and $\mathbf{load}_7$ r_3 z are fulfilled.

Case ($\mathbf{load}_7$ r_3 z before $\mathbf{store}_{15}$ z r_1):

In this case, the obligation $\mathbf{load}_7$ r_3 z is specialized with the value of z from the obligation of $\mathbf{store}_3$ z 0, because the obligation of $\mathbf{store}_3$ z 0 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligation of $\mathbf{load}_7$ r_3 z must be fulfilled after $\mathbf{spawn}_4$, and the only other obligation that causes an update of z, i.e. the obligation of $\mathbf{store}_{15}$ z r_1, is fulfilled after the obligation of $\mathbf{load}_7$ r_3 z due to the assumption of this case. Thus the obligation of $\mathbf{load}_7$ r_3 z causes an update of r_3 to 0 in this case.

Case ($\mathbf{load}_7$ r_3 z after $\mathbf{store}_{15}$ z r_1):

In this case, the obligation of $\mathbf{load}_7$ r_3 z is specialized with the value of z from the obligation of $\mathbf{store}_{15}$ z r_1, because the only other obligation that causes an update of z, i.e. the obligation of $\mathbf{store}_3$ z 0, must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligation of $\mathbf{store}_{15}$ z r_1 must be fulfilled after the obligation of $\mathbf{spawn}_4$ and the obligation of $\mathbf{store}_{15}$ z r_1 is fulfilled before the obligation of $\mathbf{load}_7$ r_3 z due to the assumption of this case.

The obligation of $\mathbf{store}_{15}$ z r_1 causes an update of z to the value of r_1. The value of r_1 is 0, because the obligation of $\mathbf{load}_{14}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The obligation of $\mathbf{load}_{14}$ r_1 x causes the most recent update of r_1, because there is no other update of r_1 in the program and the obligation must be fulfilled before the obligation of $\mathbf{store}_{15}$ z r_1. The obligation

of $\mathbf{load}_{14}$ r_1 x must be fulfilled before the obligation of $\mathbf{store}_{15}$ z r_1, because both obligations are caused by the same thread and access r_1.
The obligation of $\mathbf{load}_{14}$ r_1 x causes an update of r_1 to the value of x. The value of x is 0, because the obligation of $\mathbf{store}_5$ x 0 causes the most recent update of x and the update sets x to 0. The obligation of $\mathbf{store}_5$ x 0 causes the most recent update of x for the obligation of $\mathbf{load}_{14}$ r_1 x, because both obligations must be fulfilled after the obligation of $\mathbf{spawn}_4$ and the obligation of $\mathbf{store}_5$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$ due to the assumption of this case. Thus the obligation of $\mathbf{load}_7$ r_3 z causes an update of r_3 to 0 in this case.

Case ($\mathbf{store}_{13}$ y 0):

In this case, the value of r_2 is always 0, because r_2 is initialized with 0 and the only update of r_2 is $\mathbf{load}_6$ r_2 y. The obligation of $\mathbf{load}_6$ r_2 y causes an update of r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_{13}$ y 0 causes the most recent update of y and the update sets y to 0. The obligation of $\mathbf{store}_{13}$ y 0 causes the most recent update of y, because only the obligation of $\mathbf{store}_2$ y 1 causes a further update of y, the obligation of $\mathbf{store}_2$ y 1 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligation of $\mathbf{store}_{13}$ y 0 and $\mathbf{load}_6$ r_2 y must be fulfilled after the obligation of $\mathbf{spawn}_4$, and the obligation of $\mathbf{store}_{13}$ y 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$ due to the assumption of this case. Thus the obligation of $\mathbf{load}_6$ r_2 y causes an update of r_2 to 0 in this case.

Since $\mathbf{load}_{10}$ r_5 h is dead code and no other instruction reads a **High**-variable the program c_1^- does not read information from **High**-variables in any program run. Consequently, the program c_1^- satisfies MM-Noninterference, if $\delta_1(MM)$ holds. ■

For easier reference we recall the definitions of c_2^+ and c_2^- in Figure 2.

$\mathrm{c}_2^+ :=$
$\mathbf{store}_1$ 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
 then $\mathbf{if}_{13}$ r_5 **then** $\mathbf{store}_{14}$ l 5 **else** $\mathbf{skip}_{15}$ **fi**
 else $\mathbf{if}_{16}$ r_5 **then** $\mathbf{skip}_{17}$ **else** $\mathbf{store}_{18}$ l 5 **fi**
 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

$\mathrm{c}_2^- :=$
$\mathbf{store}_1$ 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 **then** $\mathbf{load}_{11}$ r_5 h; $\mathbf{store}_{12}$ l r_5 **else** $\mathbf{skip}_{13}$ **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 2. Programs for Lemmas 13, 14, 15, 16 (Lemma 2 in the article)

Lemma 13. *The following proposition holds for the domain assignment lev with $lev(\mathrm{h})$ and $lev(x) = \mathbf{Low}$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$:* $\gamma_2(MM) \implies \mathrm{c}_2^+ \in NI_{MM}$.

Proof: Only the final value of the **Low**-variable l can depend on the initial value of a **High**-variable, because the only other variables that are updated are the variables x, y and z, and the final value of x, y and z is definitely 0. Independent of the initial memory, and in particular independent of the initial value of h, the final value of l can be either 0 or 5. Consequently, the program c_2^+ satisfies MM-Noninterference. In the following we present the arguments in detail.

The final value of x, y and z is definitely 0, because the obligations of $\mathbf{store}_6$ x 0, $\mathbf{store}_{19}$ y 0 and $\mathbf{store}_{23}$ z 0 respectively cause the last updates of x, y and z and all three updates set their respective variable to 0. The obligations of $\mathbf{store}_6$ x 0 and $\mathbf{store}_{19}$ y 0 cause the last updates of x and y, because only the obligations of $\mathbf{store}_1$ x 1 and $\mathbf{store}_2$ y 1 cause further updates of x and y and these two obligations must be fulfilled before the obligation of $\mathbf{spawn}_5$ while the obligations of $\mathbf{store}_6$ x 0 and $\mathbf{store}_{19}$ y 0 must be fulfilled after the obligation of $\mathbf{spawn}_5$. The obligation of $\mathbf{store}_{23}$ z 0 causes the last update of z, because only the obligation of $\mathbf{store}_3$ z 0 and $\mathbf{store}_{22}$ z r_1 cause further updates of z and these two obligations must be fulfilled before the obligation of $\mathbf{store}_{23}$ z 0. The obligations of $\mathbf{store}_3$ z 0 and $\mathbf{store}_{22}$ z r_1 must be fulfilled before the obligation of $\mathbf{store}_{23}$ z 0, because the obligations of $\mathbf{store}_3$ z 0 and $\mathbf{store}_{22}$ z r_1 are caused by the same thread and access the variable as the obligation of $\mathbf{store}_{23}$ z 0 (Lemma 8).

The final value of h is in $\{0, 5\}$, because the obligation of $\mathbf{store}_4$ l 0 is definitely fulfilled in every program run and the only other updates of l are $\mathbf{store}_{15}$ l 5 and $\mathbf{store}_{18}$ l 5.

We show that the final value of l can always be 0. Let mem_1 be arbitrary. We distinguish two cases based on the initial value of h.

Case (Initial value of h equals 0):

Let $t = 1, 2, 3, 4, 5, \mathbf{20}, \mathbf{21}, 6, 7, 8, \mathbf{19}, 22, 9, 10, 11, 23$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by t, because of four reasons. First, all obligations, except the obligations of $\mathbf{load}_{20}$ r_2 y, $\mathbf{load}_{21}$ r_1 x and $\mathbf{store}_{19}$ y 0 are fulfilled in the order in which their instructions are executed. Second, $\gamma_2(MM)$ holds and therefore $\phi_{\mathrm{WR}} \in \Phi$ and $\phi_{\mathrm{ROwn}} \in \Phi$ for MM. The obligation ob of $\mathbf{store}_{19}$ y 0 may be fulfilled after the obligation ob' of $\mathbf{load}_{20}$ r_2 y due to $\phi_{\mathrm{ROwn}} \in \Phi$, because $isWrite(ob)$ evaluates to true, $isRead(ob')$ evaluates to true and both access the same variable. The obligation ob of $\mathbf{store}_{19}$ y 0 may also be fulfilled after the obligation ob'' of $\mathbf{load}_{21}$ r_1 x due to ϕ_{WR}, because $isWrite(ob)$ evaluates to true, $isRead(ob'')$ evaluates to true $sources(ob) \cap sinks(ob'') = \emptyset$ and $sinks(ob) \cap sources(ob'') = \emptyset$ hold. Third, the **then**-branch of $\mathbf{if}_{12}$ is taken, and, fourth, the **else**-branch of $\mathbf{if}_{13}$ is taken.
The **then**-branch of $\mathbf{if}_{12}$ r_4 is taken, because r_4 is 1 when executing $\mathbf{if}_{12}$. The value of r_4 is 1, because the obligation of $\mathbf{and}_{10}$ r_4 r_2 r_3 causes the most recent update of r_4 and the the value of r_2 and r_3 is 1 when executing $\mathbf{and}_{10}$ r_4 r_2 r_3. The value of r_2 is 1, because the obligation of $\mathbf{load}_8$ r_2 y causes the most recent update of r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_2$ y 1 causes the most recent update of y to 1. The value of r_3 is 1, because the obligation of $\mathbf{load}_9$ r_3 z causes the most recent update of r_3 to the value of z. The value of z is 1, because the obligation of $\mathbf{store}_{22}$ z r_1 causes the most recent update of z to the value of r_1. The value of r_1 is 1, because the obligation of $\mathbf{load}_{21}$ r_1 x causes the most recent update of r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of x to 1.
The **else**-branch of $\mathbf{if}_{13}$ r_5 is taken, because r_5 is 0 when executing $\mathbf{if}_{13}$. The value of r_5 is 0, because the obligation of $\mathbf{load}_{11}$ r_5 h causes the most recent update of r_5 to the value of h. The value of h is 0, because there is no update of h in the program and the initial value of h is 0 by assumption of this case.
Since the **else**-branch of $\mathbf{if}_{13}$ is taken the only obligation that causes an update of l is the obligation of $\mathbf{store}_3$ l 0. Thus the final value of l is 0.

Case (Initial value of h does not equal 0):
Let $t = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 19, 20, 21, 22, 23$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by t, because of three reasons. First, all obligation are processed in the order in which their instructions are executed. Second, the **else**-branch of $\mathbf{if}_{12}$ is taken and, third, the **then**-branch of $\mathbf{if}_{16}$ is taken.
The **else**-branch of $\mathbf{if}_{12}$ r_4 is taken, because r_4 is 0 when executing $\mathbf{if}_{12}$. The value of r_4 is 0, because the obligation of $\mathbf{and}_{10}$ r_4 r_2 r_3 causes the most recent update of r_4 and the value of r_3 is 0 when executing $\mathbf{and}_{10}$ r_4 r_2 r_3. The value of r_3 is 0, because the obligation of $\mathbf{load}_9$ r_3 z causes the most recent update of r_3 and the update sets r_3 to the value of z. The value of z is 0, because the obligation of $\mathbf{store}_3$ z 0 causes the most recent update of z and the update set z to 0.
The **then**-branch of $\mathbf{if}_{16}$ r_5 is taken, because r_5 is not 0 when executing $\mathbf{if}_{16}$. The value of r_5 is not 0, because the obligation of $\mathbf{load}_{11}$ r_5 h causes the most recent update of r_5 and the update sets r_5 to the value of h. The value of h is not 0, because there is no update to h in the program and the initial value of h is different from 0 by assumption of this case.
Since the **then**-branch of $\mathbf{if}_{16}$ is taken the only obligation that causes an update of l is the obligation of $\mathbf{store}_4$ l 0. Thus the final value of l is 0.

We show that the final value of l can always be 5. Let mem_1 be arbitrary. We distinguish two cases based on the initial value of h.

Case (Initial value of h equals 0):
Let $t = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 18, 19, 20, 21, 22, 23$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by t, because of three reasons. First, all obligation are processed in the order in which their instructions are executed. Second, the **else**-branch of $\mathbf{if}_{12}$ is taken and, third, the **else**-branch of $\mathbf{if}_{16}$ is taken.
The **else**-branch of $\mathbf{if}_{12}$ r_4 is taken, because r_4 is 0 when executing $\mathbf{if}_{12}$. The value of r_4 is 0, because the obligation of $\mathbf{and}_{10}$ r_4 r_2 r_3 causes the most recent update of r_4 and the value of r_3 is 0 when executing $\mathbf{and}_{10}$ r_4 r_2 r_3. The value of r_3 is 0, because the obligation of $\mathbf{load}_9$ r_3 z causes the most recent update of r_3 and the update sets r_3 to the value of z. The value of z is 0, because the obligation of $\mathbf{store}_3$ z 0 causes the most recent update of z and the update set z to 0.
The **else**-branch of $\mathbf{if}_{16}$ r_5 is taken, because r_5 is 0 when executing $\mathbf{if}_{16}$. The value of r_5 is 0, because the obligation of $\mathbf{load}_{11}$ r_5 h causes the most recent update of r_5 to the value of h. The value of h is 0, because there is no update of h in the program and the initial value of h is 0 by assumption of this case.
Since the **else**-branch of $\mathbf{if}_{16}$ is taken, the obligation of $\mathbf{store}_{18}$ l 5 is fulfilled as the last update of l. Thus the final value of l is 5.

Case (Initial value of h does not equal 0):
Let $t = 1, 2, 3, 4, 5, \mathbf{20}, \mathbf{21}, 6, 7, 8, \mathbf{19}, 22, 9, 10, 11, 14, 23$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by the t, because of four reasons. First, all obligations, except the obligations of $\mathbf{load}_{20}$ r_2 y, $\mathbf{load}_{21}$ r_1 x and $\mathbf{store}_{19}$ y 0 are fulfilled in the order in which their instructions are executed. Second, $\gamma_2(MM)$ holds and therefore $\phi_{\mathrm{WR}} \in \Phi$ and $\phi_{\mathrm{ROwn}} \in \Phi$ for MM. The

obligation ob of $\mathbf{store}_{19}$ y 0 may be fulfilled after the obligation ob' of $\mathbf{load}_{20}$ r_2 y due to $\phi_{\mathrm{ROwn}} \in \Phi$, because $isWrite(ob)$ evaluates to true, $isRead(ob')$ evaluates to true and both access the same variable. The obligation ob of $\mathbf{store}_{19}$ y 0 may also be fulfilled after the obligation ob'' of $\mathbf{load}_{21}$ r_1 x due to ϕ_{WR}, because $isWrite(ob)$ evaluates to true, $isRead(ob'')$ evaluates to true $sources(ob) \cap sinks(ob'') = \emptyset$ and $sinks(ob) \cap sources(ob'') = \emptyset$ hold. Third, the **then**-branch of $\mathbf{if}_{12}$ is taken, and, fourth, the **then**-branch of $\mathbf{if}_{13}$ is taken.
The **then**-branch of $\mathbf{if}_{12}$ r_4 is taken, because r_4 is 1 when executing $\mathbf{if}_{12}$. The value of r_4 is 1, because the obligation of $\mathbf{and}_{10}$ r_4 r_2 r_3 causes the most recent update of r_4 and the value of r_2 and r_3 is 1 when executing $\mathbf{and}_{10}$ r_4 r_2 r_3. The value of r_2 is 1, because the obligation of $\mathbf{load}_8$ r_2 y causes the most recent update of r_2 and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_2$ y 1 causes the most recent update of y and the update sets y to 1. The value of r_3 is 1, because the obligation of $\mathbf{load}_9$ r_3 z causes the most recent update of r_3 and the update sets r_3 to the value of z. The value of z is 1, because the obligation of $\mathbf{store}_{22}$ z r_1 causes the most recent update of z and the update sets z to the value of r_1. The value of r_1 is 1, because the obligation of $\mathbf{load}_{21}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of x and the update sets x to 1.
The **then**-branch of $\mathbf{if}_{13}$ r_5 is taken, because r_5 is not 0 when executing $\mathbf{if}_{13}$. The value of r_5 is not 0, because the obligation of $\mathbf{load}_{11}$ r_5 h causes the most recent update of r_5 and the update sets r_5 to the value of h. The value of h is not 0, because there is no update to h in the program and the initial value of h is different from 0 by assumption of this case.
Since the **then**-branch of $\mathbf{if}_{13}$ is taken, the obligation of $\mathbf{store}_{14}$ l 5 is fulfilled as the last update of l. Thus the final value of l is 5.

That means independent of the initial value of the **High**-variable h, the final value of x, y and z is 0, the final value of l can be either 0 or 5, and the values of all other variables remain unchanged. Hence, for all pairs of **Low**-equal initial memories **Low**-equal final memories are reachable. Consequently, the program c_2^+ satisfies MM-Noninterference, if $\gamma_2(MM)$ holds. ■

Lemma 14. *The following proposition holds for the domain assignment lev with $lev(\mathrm{h})$ and $lev(x) = \mathbf{Low}$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$:* $\delta_2(MM) \implies c_2^+ \notin NI_{MM}$.

Proof: The final value of the **Low**-variable l can only be 5 if the initial value of the **High**-variable h is 0, because $\mathbf{store}_4$ l 0 definitely updates l to 0 and there are only two instructions that can update l to 5. The two instructions are $\mathbf{store}_{14}$ l 5 and $\mathbf{store}_{18}$ l 5. The instruction $\mathbf{store}_{14}$ l 5 is in the **then**-branch of $\mathbf{if}_{12}$. The **then**-branch of $\mathbf{if}_{12}$ is dead code. The instruction $\mathbf{store}_{18}$ l 5 is in the **else**-branch of $\mathbf{if}_{16}$. The **else**-branch of $\mathbf{if}_{16}$ is only reachable if the initial value of h is 0. Thus the program does not satisfy MM-Noninterference. In the following we present the arguments in detail.

The **else**-branch of $\mathbf{if}_{16}$ r_5 is only reachable if the initial value of h is 0, because the obligation of $\mathbf{load}_{11}$ r_5 h is fulfilled before $\mathbf{if}_{16}$ is executed, the obligation of $\mathbf{load}_{11}$ r_5 h causes an update of r_5 to the initial value of h. The obligation of $\mathbf{load}_{11}$ r_5 h causes an update of r_5 to the initial value of h, because there is no update of h in the program.

The **then**-branch of $\mathbf{if}_{12}$ r_4 is dead code, because the value of r_4 is always 0. The value of r_4 is always 0, because it is initialized with 0 and only the obligation of $\mathbf{and}_{10}$ r_4 r_2 r_3 causes an update of r_4. The obligation of $\mathbf{and}_{10}$ r_4 r_2 r_3 causes an update of r_4 to 0, because either r_2 or r_3 is always 0 when executing $\mathbf{and}_{10}$. This is due to the fact that both r_2 and r_3 are initialized with 0 and only the obligations of $\mathbf{load}_8$ r_2 y and $\mathbf{load}_9$ r_3 z respectively cause an update of r_2 and r_3. At least one of the obligations causes an update of its respective register to 0. Which of the two obligations causes an update to 0 depends on the obligation that is fulfilled directly after the obligation of $\mathbf{spawn}_5$: Either the obligation of $\mathbf{store}_6$ x 0 or the obligation of $\mathbf{store}_{19}$ y 0 must be fulfilled directly after the obligation of $\mathbf{spawn}_5$ This is due to the fact that these two instructions belong to different threads and the obligations of their subsequent instructions cannot be fulfilled before the obligations of these two instructions. The subsequent instruction of $\mathbf{store}_6$ x 0 is $\mathbf{fence}_7$ and $ob \in Fe$ holds for the obligation ob of $\mathbf{fence}_7$. Since $\delta_2(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{fence}_7$, because $isWrite$ holds for the obligation of $\mathbf{store}_6$ x 0. The subsequent instruction of $\mathbf{store}_{19}$ y 0 is $\mathbf{load}_{20}$ r_2 y and $isRead$ holds for the obligation of $\mathbf{load}_{20}$ r_2 y. Since $\delta_2(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{load}_{20}$ r_2 y, because $isWrite$ holds for the obligation of $\mathbf{store}_{19}$ y 0 and both obligations access the same variable, i.e. y. We distinguish these two cases based on the obligation that is fulfilled after the obligation of $\mathbf{spawn}_5$ and show that either r_2 or r_3 is 0.

Case ($\mathbf{store}_6$ x 0):
In this case, the value of r_3 is always 0, because r_3 is initialized with 0, only the obligation of $\mathbf{load}_9$ r_3 z causes an update of r_3 and the update caused by the obligation sets r_3 to the value of z. The value of z is 0, because the obligation of either $\mathbf{store}_3$ z 0 or $\mathbf{store}_{23}$ z 0 or $\mathbf{store}_{22}$ z r_1 causes the most recent update of z. All three events update z to 0. The obligation of $\mathbf{store}_{22}$ z r_1 sets z to the value of r_1. The value of r_1 is 0, because the obligation of $\mathbf{load}_{21}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The value of x is 0, because the obligation of $\mathbf{store}_6$ x 0 causes the most recent update of x and the update sets x to 0. The obligation of $\mathbf{store}_6$ x 0 causes the most recent update of x for $\mathbf{load}_{21}$ r_1 x, because both obligations must be fulfilled after the obligation of $\mathbf{spawn}_5$, the obligation of $\mathbf{store}_6$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_5$ due to the assumption of this case, and there is no other obligation that causes an update of x that can be fulfilled after $\mathbf{spawn}_5$. Thus the obligation of $\mathbf{store}_{22}$ z r_1 causes an update of r_1 to 0.

Case ($\mathbf{store}_{19}$ y 0):

In this case, the value of r_2 is always 0, because r_2 is initialized with 0, only the obligation of $\mathbf{load}_8$ r_2 y causes an update of r_2, and the update caused by the obligation sets r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_{19}$ y 0 causes the most recent update of y and the update sets y to 0. The obligation of $\mathbf{store}_{19}$ y 0 causes the most recent update of y for the obligation of $\mathbf{load}_8$ r_2 y, because both obligations are fulfilled after the obligation of $\mathbf{spawn}_5$, the obligation of $\mathbf{store}_{19}$ y 0 is fulfilled directly after the obligation of $\mathbf{spawn}_5$ due to the assumption of this case, and there is no other obligation that causes an update of y that can be fulfilled after the obligation of $\mathbf{spawn}_5$.

Based on these observations we construct a concrete counter example: We choose initial memories mem_1 and mem_1' such that $mem_1(x) = 0$ for all $x \in \mathcal{X}$, $mem_1'(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $mem_1'(\mathrm{h}) = 23$. The global memories mem_1 and mem_1' satisfy $mem_1 =_L mem_1'$, because $mem_1(x) = 0 = mem_1'(x)$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $lev(\mathrm{h}) = \mathbf{High}$. From mem_1, a final memory mem_2 is reachable with $mem_2(\mathrm{l}) = 5$, because the initial value of h is 0. All final memories mem_2' that are reachable from mem_1' satisfy $mem_2'(\mathrm{l}) \neq 5$, because the initial value of h is 23 and not 0. Thus, for all final memories mem_2' that are reachable from mem_1', we have $mem_2 \neq_L mem_2'$, because $mem_2(\mathrm{l}) = 5 \neq mem_2'(\mathrm{l})$ and $lev(\mathrm{l}) = \mathbf{Low}$.

Consequently, the program c_2^+ does not satisfy MM-Noninterference, if $\delta_2(MM)$ holds. ■

Lemma 15. *The following proposition holds for the domain assignment lev with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\gamma_2(MM) \implies \mathrm{c}_2^- \notin NI_{MM}$.

Proof: The two instructions $\mathbf{load}_{11}$ and $\mathbf{store}_{12}$ in the **then**-branch of $\mathbf{if}_{10}$ form a direct leak, because $\mathbf{load}_{11}$ reads the value of the **High**-variable h into r_5, and $\mathbf{store}_{12}$ subsequently writes the value of r_5 into the **Low**-variable l. Due to this direct leak and the fact that this **then**-branch is reachable, the program c_2^- does not satisfy MM-noninterference. In the following we present the arguments in detail.

Let $t = 1, 2, 3, 4, \mathbf{15}, \mathbf{16}, 5, 6, 7, \mathbf{14}, 17, 8, 9$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers up to reaching $\mathbf{if}_{10}$ in the order given by t, because all obligations except the obligations of $\mathbf{store}_{14}$ y 0, $\mathbf{load}_{15}$ r_2 y and $\mathbf{load}_{16}$ r_1 x, are fulfilled in the order in which their instructions are executed. Since $\gamma_2(MM)$ holds, $\phi_{\mathrm{WR}} \in \Phi$ and $\phi_{\mathrm{ROwn}} \in \Phi$ for MM. Thus, the obligation ob of $\mathbf{store}_{14}$ y 0 may be fulfilled after the obligation ob' of $\mathbf{load}_{15}$ r_2 y due to ϕ_{ROwn}, because $isWrite(ob)$ evaluates to true, $isRead(ob')$ evaluates to true and both access the same variable. The obligation ob of $\mathbf{store}_{14}$ y 0 may also be fulfilled after the obligation ob'' of $\mathbf{load}_{16}$ r_1 x due to ϕ_{WR}, because $isWrite(ob)$ evaluates to true, $isRead(ob'')$ evaluates to true $sources(ob) \cap sinks(ob'') = \emptyset$ and $sinks(ob) \cap sources(ob'') = \emptyset$ hold. We will now argue why the instructions $\mathbf{load}_{11}$ and $\mathbf{store}_{12}$ will be executed and their obligations will get fulfilled after t.

The sequence t always leads to a register state reg_{10} with $reg_{10}(\mathrm{r}_2) = 1$, $reg_{10}(\mathrm{r}_3) = 1$ and $reg_{10}(\mathrm{r}_4) = 1$, because of three reasons. First, the obligation of $\mathbf{load}_7$ r_2 y causes the last update of r_2 and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_2$ y 1 causes the most recent update of y and the update sets y to 1. Second, the obligation of $\mathbf{load}_8$ r_3 z causes the last update of r_3 and the update sets r_3 to the value of z. The value of z is 1, because the obligation of $\mathbf{store}_{17}$ z r_1 causes the most recent update of z and the update sets z to the value of r_1. The value of r_1 is 1, because the obligation of $\mathbf{load}_{16}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The value of x is 1, because the the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of x and the update sets x to 1. Thus the obligation of $\mathbf{load}_8$ r_3 z reads 1 into r_3. Third, the obligation of $\mathbf{and}_9$ r_4 r_2 r_3 causes the last update of r_4. The obligation of $\mathbf{and}_9$ r_4 r_2 r_3 causes an update of r_4 to 1, because the obligations of $\mathbf{load}_7$ r_2 y and $\mathbf{load}_8$ r_3 z respectively cause the most recent updates of r_2 and r_3, and the updates caused by these two obligations cause an update of their respective registers to 1 (as we have argued before). Consequently, the **then**-branch of $\mathbf{if}_{10}$ r_4 is taken after t. This means that the instructions $\mathbf{load}_{11}$ r_5 h and $\mathbf{store}_{12}$ l r_5 are executed and their obligations fulfilled after t.

We choose an initial memory mem_1 with $mem_1(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $mem_1(\mathrm{h}) = 5$. The final value of l is 5, because the obligation of $\mathbf{store}_{12}$ l r_5 causes an update of l to the value of r_5. The value of r_5 is 5, because the obligation of $\mathbf{load}_{11}$ r_5 h causes an update of r_5 to the value of h. The value of h is 5, there is no update of h in the program and the initial value of h is 5, i.e. $mem_1(\mathrm{h}) = 5$, according to the initial memory that we have chosen. The obligation of $\mathbf{load}_{11}$ r_5 h must be fulfilled before the obligation of $\mathbf{store}_{12}$ l r_5, because both obligations are caused by the same thread and access r_5.

We now show that there is an initial memory mem_1' with $mem_1 =_L mem_1'$ for which 5 is not a possible final value for l. We choose an initial value mem_1' with $mem_1'(x) = 0$ for all $x \in \mathcal{X}$. From the definition of **Low**-equality we know that $mem_1 =_L mem_1'$, because $mem_1(x) = mem_1'(x)$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $lev(\mathrm{h}) = \mathbf{High}$. For the initial memory mem_1' the final value of l must be in $\{0, 1\}$, because the initial value of all variables is 0, all constants that appear in the program are either 0 or 1, and the only computation, i.e. **and** has $\{0, 1\}$ as range of values. Consequently, the final value of l cannot be 5. Hence, $mem_2'(\mathrm{l}) \neq 5 = mem_2(\mathrm{l})$ holds for all final memories mem_2' that are reachable from mem_1'.

This means that there is no final memory reachable from mem_1' that is **Low**-equal to mem_2. Consequently, the program c_2^- does not satisfy MM-Noninterference, if $\gamma_2(MM)$ holds. ■

Lemma 16. *The following proposition holds for the domain assignment* lev *with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\delta_2(MM) \implies \mathrm{c}_2^- \in NI_{MM}$.

Proof: Only $\mathbf{load}_{11}$ in the $\mathbf{then}$-branch of $\mathbf{if}_9$ reads a $\mathbf{High}$-variable. The $\mathbf{then}$-branch of $\mathbf{if}_9$ is dead code. Since the only instruction that reads a $\mathbf{High}$-variable is dead code, the program c_2^- satisfies MM-noninterference. In the following we present the arguments in detail.

The instruction $\mathbf{load}_{11}$ is dead code, because it is in the $\mathbf{then}$-branch of $\mathbf{if}_{10}$ r_4 and the value of r_4 of the spawned thread is always 0. The value of r_4 is always 0, because it is initialized with 0, only the obligation of $\mathbf{and}_9$ r_4 r_2 r_3 causes an update of r_4 and the update sets r_4 to 0. The update caused by the obligation of $\mathbf{and}_9$ r_4 r_2 r_3 sets r_4 to 0, because the value of either r_2 or r_3 is 0 in all possible sequences for fulfilling obligations when executing $\mathbf{and}_9$.

We now show that the value of either r_2 or r_3 is always 0 in all possible sequences for fulfilling obligations. All possible sequences for fulfilling obligations up to (inclusively) the obligation of $\mathbf{spawn}_4$ have the same effect on the global state, because each of the obligations of $\mathbf{store}_1$ x 1, $\mathbf{store}_2$ y 1 and $\mathbf{store}_3$ z 0 causes an update of a different variable, and the obligations of all three instructions must be fulfilled before the obligation of $\mathbf{spawn}_4$. Only two events can be fulfilled directly after the event of $\mathbf{spawn}_4$: The obligation of either $\mathbf{store}_5$ x 0 or $\mathbf{store}_{14}$ y 0 must be fulfilled directly after the obligation of $\mathbf{spawn}_4$. This is due to the fact that these two instructions belong to different threads and the obligations of their subsequent instructions cannot be fulfilled before the obligations of these two instructions. The subsequent instruction of $\mathbf{store}_5$ x 0 is $\mathbf{fence}_6$ and $ob \in Fe$ holds for the obligation ob of $\mathbf{fence}_6$. Since $\delta_2(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{fence}_6$, because $isWrite$ holds for the obligation of $\mathbf{store}_5$ x 0. The subsequent instruction of $\mathbf{store}_{14}$ y 0 is $\mathbf{load}_{15}$ r_2 y and $isRead$ holds for the obligation of $\mathbf{load}_{15}$ r_2 y. Since $\delta_2(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{load}_{15}$ r_2 y, because $isWrite$ holds for the obligation of $\mathbf{store}_{14}$ y 0 and both obligations access the same variable, i.e. y. We distinguish two cases. In the first case, the obligation of $\mathbf{store}_5$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$. In the second case, the obligation of $\mathbf{store}_{14}$ y 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$.

Case ($\mathbf{store}_5$ x 0):

In this case, the value of register r_3 is always 0, because the register r_3 is initialized with 0 and the only update of r_3 is $\mathbf{load}_8$ r_3 z. The obligation of $\mathbf{load}_8$ r_3 z always causes an update of r_4 to 0. We now show why this holds: The only remaining updates of z after fulfilling the obligation of $\mathbf{spawn}_4$ is $\mathbf{store}_{17}$ z r_1. We distinguish two cases based on the order in which the obligations of $\mathbf{load}_8$ r_3 z and $\mathbf{store}_{17}$ z r_1 are fulfilled.

Case ($\mathbf{load}_8$ r_3 z before $\mathbf{store}_{17}$ z r_1):

In this case, the obligation of $\mathbf{load}_8$ r_3 z is specialized with the value of z from the obligation of $\mathbf{store}_3$ z 0, because the obligation of $\mathbf{store}_3$ z 0 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligation of $\mathbf{load}_8$ r_3 z must be fulfilled after the obligation of $\mathbf{spawn}_4$, and the only other obligation that causes an update of z, i.e. the obligation of $\mathbf{store}_{17}$ z r_1, is fulfilled after the obligation of $\mathbf{load}_8$ r_3 z due to the assumption of this case. Thus the obligation of $\mathbf{load}_8$ r_3 z causes an update of r_3 to 0 in this case.

Case ($\mathbf{load}_8$ r_3 z after $\mathbf{store}_{17}$ z r_1):

In this case, the obligation of $\mathbf{load}_8$ r_3 z is specialized with the value of z from the obligation of $\mathbf{store}_{17}$ z r_1, because the only other obligation that causes an update of z, i.e. the obligation of $\mathbf{store}_3$ z 0, must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligation of $\mathbf{store}_{17}$ z r_1 must be fulfilled after the obligation of $\mathbf{spawn}_4$ and the obligation of $\mathbf{store}_{17}$ z r_1 is fulfilled before the obligation of $\mathbf{load}_8$ r_3 z due to the assumption of this case.

The obligation of $\mathbf{store}_{17}$ z r_1 causes an update of z to the value of r_1. The value of r_1 is 0, because the obligation of $\mathbf{load}_{16}$ r_1 x causes the most recent update of r_1 and the update sets r_1 to the value of x. The obligation of $\mathbf{load}_{16}$ r_1 x causes the most recent update of r_1, because there is no other update of r_1 and the obligation of $\mathbf{load}_{16}$ r_1 x must be fulfilled before the obligation of $\mathbf{store}_{17}$ z r_1. The obligation of $\mathbf{load}_{16}$ r_1 x must be fulfilled before the obligation of $\mathbf{store}_{17}$ z r_1, because both obligations are caused by the same thread and access r_1.

The obligation of $\mathbf{load}_{16}$ r_1 x causes an update of r_1 to the value of x. The value of x is 0, because the obligation of $\mathbf{store}_5$ x 0 causes the most recent update of x and the update sets x to 0. The obligation of $\mathbf{store}_5$ x 0 causes the most recent update of x for the obligation of $\mathbf{load}_{16}$ r_1 x, because both obligations must be fulfilled after the obligation of $\mathbf{spawn}_4$ and the obligation of $\mathbf{store}_5$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$ due to the assumption of this case. Thus the obligation of $\mathbf{load}_8$ r_3 z causes an update of r_3 to 0 in this case.

Case ($\mathbf{store}_{14}$ y 0):

In this case, the value of register r_2 is always 0, because the register r_2 is initialized with 0 and the only update of r_2 is $\mathbf{load}_7$ r_2 y. The obligation of $\mathbf{load}_7$ r_2 y causes an update of r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_{14}$ y 0 causes the most recent update of y and the update sets y to 0. The obligation of $\mathbf{store}_{14}$ y 0 causes the most recent update of y for the obligation of $\mathbf{load}_7$ r_2 y, because only the obligation of $\mathbf{store}_2$ y 1 causes a further update of y, the obligation of $\mathbf{store}_2$ y 1 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligations of $\mathbf{store}_{14}$ y 0 and $\mathbf{load}_7$ r_2 y must be fulfilled after the obligation of $\mathbf{spawn}_4$,

and the obligation of $\mathbf{store}_{14}$ y 0 is fulfilled before the obligation of $\mathbf{load}_7$ r_2 y due to the assumption of this case. Thus the obligation of $\mathbf{load}_7$ r_2 y causes an update of r_2 to 0 in this case.

Since $\mathbf{load}_{11}$ r_5 h is dead code and no other instruction reads a **High**-variable the program c_2^- does not read information from **High**-variables in any program run. Consequently, the program c_2^- satisfies MM-Noninterference, if $\delta_2(MM)$. ■

For easier reference we recall the definitions of c_3^+ and c_3^- in Figure 3.

$\mathrm{c}_3^+ :=$
$\mathbf{store}_1$ 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
then $\mathbf{if}_{10}$ r_4 **then** $\mathbf{store}_{11}$ l 5 **else** $\mathbf{skip}_{12}$ **fi**
else $\mathbf{if}_{13}$ r_4 **then** $\mathbf{skip}_{14}$ **else** $\mathbf{store}_{15}$ l 5 **fi**
fi);
$\mathbf{store}_{16}$ x 0; $\mathbf{store}_{17}$ y 1

$\mathrm{c}_3^- :=$
$\mathbf{store}_1$ 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 **then** $\mathbf{load}_8$ r_4 h; $\mathbf{store}_9$ l r_4 **else** $\mathbf{skip}_{10}$ **fi**);
$\mathbf{store}_{11}$ x 0; $\mathbf{store}_{12}$ y 1

Figure 3. Programs for Lemmas 17, 18, 19, 20 (Lemma 3 in the article)

Lemma 17. *The following proposition holds for the domain assignment* lev *with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\gamma_3(MM) \implies \mathrm{c}_3^+ \in NI_{MM}$.

Proof: Only the final value of **Low**-variable l can depend on the initial value of a **High**-variable, because the only other variables that are updated are the variables x and y, and the final values of x and y are definitely 0 and 1, respectively. Independent of the initial memory, and in particular independent of the initial value of h, the final value of l can be either 0 or 5. Consequently, the program c_3^+ satisfies MM-Noninterference. In the following we present the arguments in detail.

The final values of x and y are definitely 0 and 1, respectively, because the obligations of $\mathbf{store}_{16}$ x 0 and $\mathbf{store}_{17}$ y 1 cause the last updates of x and y, and these updates set x and y to 0 and 1, respectively. The obligations of $\mathbf{store}_{16}$ x 0 and $\mathbf{store}_{17}$ y 1 cause the last updates of x and y, because only the obligations of $\mathbf{store}_1$ x 1 and $\mathbf{store}_2$ y 0 cause further updates of x and y, and the obligations of $\mathbf{store}_1$ x 1 and $\mathbf{store}_2$ y 0 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligations of $\mathbf{store}_{16}$ x 0 and $\mathbf{store}_{17}$ y 1 must be fulfilled after the event of $\mathbf{spawn}_4$.

The final value of h is in $\{0, 5\}$, because the obligation of $\mathbf{store}_3$ l 0 is definitely fulfilled in every program run and the only other updates of l are $\mathbf{store}_{11}$ l 5 and $\mathbf{store}_{15}$ l 5.

We show that the final value of l can always be 0. Let mem_1 be arbitrary. We distinguish two cases based on the initial value of h.

Case (Initial value of h equals 0):

Let $t = 1, 2, 3, 4, \mathbf{17}, 5, 6, 7, 8, \mathbf{16}$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by t, because of four reasons. First, all obligations, except the obligations of $\mathbf{store}_{17}$ y 1 and $\mathbf{store}_{16}$ x 0 are fulfilled in the order in which their instructions are executed. Second, $\gamma_3(MM)$ holds and therefore $\phi_{\mathrm{WW}} \in \Phi$ for MM. The obligation ob of $\mathbf{store}_{16}$ x 0 may be fulfilled after the obligation ob' of $\mathbf{store}_{17}$ y 1 due to ϕ_{WW}, because $isWrite(ob)$ evaluates to true, $isWrite(ob')$ evaluates to true, $sinks(ob) \cap sinks(ob') = \emptyset$ holds. Third, the **then**-branch of $\mathbf{if}_9$ is taken, and, fourth, the **else**-branch of $\mathbf{if}_{10}$ is taken.

The **then**-branch of $\mathbf{if}_9$ r_3 is taken, because r_3 is 1 when executing $\mathbf{if}_9$. The value of r_3 is 1, because the obligation of $\mathbf{and}_7$ r_3 r_1 r_2 causes the most recent update of r_3 and the value of r_1 and r_2 is 1 when executing $\mathbf{and}_7$ r_3 r_1 r_2. The value of r_1 is 1, because the obligation of $\mathbf{load}_6$ r_1 x causes the most recent update of r_1, and the update sets r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of x and the update sets x to 1. The value of r_2 is 1, because the obligation of $\mathbf{load}_5$ r_2 y causes the most recent update, and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_{17}$ y 1 causes the most recent update of y and the update sets y to 1.

The **else**-branch of $\mathbf{if}_{10}$ r_4 is taken, because r_4 is 0 when executing $\mathbf{if}_{10}$. The value of r_4 is 0, because the obligation of $\mathbf{load}_8$ r_4 h causes the most recent update of r_4, and the udpate sets r_4 to the value of h. The value of h is 0, because there is no update of h in the program and the initial value of h is 0 by assumption of this case.

Since the **else**-branch of $\mathbf{if}_{10}$ is taken, the only obligation causing an update of l is the obligation of $\mathbf{store}_3$ l 0. Thus the final value of l is 0.

Case (Initial value of h does not equal 0):

Let $t = 1, 2, 3, 4, 5, 6, 7, 8, 16, 17$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by t, because of three reasons. First, all obligations

are processed in the order in which their instructions are executed. Second, the **else**-branch of $\mathbf{if}_9$ is taken and, third, the **then**-branch of $\mathbf{if}_{13}$ is taken.
The **else**-branch of $\mathbf{if}_9$ r_3 is taken, because r_3 is 0 when executing $\mathbf{if}_9$. The value of r_3 is 0, because the obligation of $\mathbf{and}_7$ r_3 r_1 r_2 causes the most recent update of r_3, and the value of r_2 is 0 when executing $\mathbf{and}_7$ r_3 r_1 r_2. The value of r_2 is 0, because the obligation of $\mathbf{load}_5$ r_2 y causes the most recent update of r_2, and the update sets r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_2$ y 0 causes the most recent update of y, and the update sets y to 0.
The **then**-branch of $\mathbf{if}_{13}$ r_4 is taken, because r_4 is not 0 when executing $\mathbf{if}_{13}$. The value of r_4 is not 0, because the obligation of $\mathbf{load}_8$ r_4 h causes the most recent update of r_4, and the udpate sets r_4 to the value of h. The value of h is not 0, because there is no update of h in the program and the initial value of h is different from 0 by assumption of this case.
Since the **then**-branch of $\mathbf{if}_{13}$ is taken the only obligation causing and update of l is the obligation of $\mathbf{store}_3$ l 0. Thus the final value of l is 0.

We show that the final value of l can always be 5. Let mem_1 be arbitrary. We distinguish two cases based on the initial value of h.

Case (Initial value of h equals 0):
Let $t = 1, 2, 3, 4, 5, 6, 7, 8, 15, 16, 17$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these numbers as identifiers in the order given by t, because of three reasons. First, all obligations are fulfilled in the order in which their instructions are executed. Second, the **else**-branch of $\mathbf{if}_9$ is taken and, third, the **else**-branch of $\mathbf{if}_{13}$ is taken.
The **else**-branch of $\mathbf{if}_9$ r_3 is taken, because r_3 is 0 when executing $\mathbf{if}_9$. The value of r_3 is 0, because the obligation of $\mathbf{and}_7$ r_3 r_1 r_2 causes the most recent update of r_3, and the value of r_2 is 0 when executing $\mathbf{and}_7$ r_3 r_1 r_2. The value of r_2 is 0, because the obligation of $\mathbf{load}_5$ r_2 y causes the most recent update of r_2, and the update sets r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_2$ y 0 causes the most recent update of y, and the update sets y to 0.
The **else**-branch of $\mathbf{if}_{13}$ r_4 is taken, because r_4 is 0 when executing $\mathbf{if}_{13}$. The value of r_4 is 0, because the obligation of $\mathbf{load}_8$ r_4 h causes the most recent update of r_4, and the udpate sets r_4 to the value of h. The value of h is 0, because there is no update of h in the program and the initial value of h is 0 by assumption of this case.
Since the **else**-branch of $\mathbf{if}_{13}$ is taken, the obligation of $\mathbf{store}_{15}$ l 5 is fulfilled as the last update of l. Thus the final value of l is 5.

Case (Initial value of h does not equal 0):
Let $t = 1, 2, 3, 4, \mathbf{17}, 5, 6, 7, 8, 11, \mathbf{16}$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with with these numbers as identifiers in the order given by t, because of four reasons. First, all obligations, except the obligations of $\mathbf{store}_{17}$ y 1 and $\mathbf{store}_{16}$ x 0, are fulfilled in the order in which their instructions are executed. Second, $\gamma_3(MM)$ holds and therefore $\phi_{\mathrm{WW}} \in \Phi$ for MM. The obligation ob of $\mathbf{store}_{16}$ x 0 may be fulfilled after the obligation ob' of $\mathbf{store}_{17}$ y 1 due to ϕ_{WW}, because $isWrite(ob)$ evaluates to true, $isWrite(ob')$ evaluates to true, $sinks(ob) \cap sinks(ob') = \emptyset$ holds. Third, the **then**-branch of $\mathbf{if}_9$ is taken, and, fourth, the **then**-branch of $\mathbf{if}_{10}$ is taken.
The **then**-branch of $\mathbf{if}_9$ r_3 is taken, because r_3 is 1 when executing $\mathbf{if}_9$. The value of r_3 is 1, because the obligation of $\mathbf{and}_7$ r_3 r_1 r_2 causes the most recent update of r_3 and the value of r_1 and r_2 is 1 when executing $\mathbf{and}_7$ r_3 r_1 r_2. The value of r_1 is 1, because the obligation of $\mathbf{load}_6$ r_1 x causes the most recent update of r_1, and the update sets r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of x and the update sets x to 1. The value of r_2 is 1, because the obligation of $\mathbf{load}_5$ r_2 y causes the most recent update of r_2, and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_{17}$ y 1 causes the most recent update of y and the update sets y to 1.
The **then**-branch of $\mathbf{if}_{10}$ r_4 is taken, because r_4 is not 0 when executing $\mathbf{if}_{10}$. The value of r_4 is not 0, because the obligation of $\mathbf{load}_8$ r_4 h causes the most recent update of r_4, and the udpate sets r_4 to the value of h. The value of h is not 0, because there is no update of h in the program and the initial value of h is different from 0 by assumption of this case.
Since the **then**-branch of $\mathbf{if}_{10}$ is taken, the obligation of $\mathbf{store}_{11}$ l 5 is fulfilled as the last update of l. Thus the final value of l is 5.

That means independent of the initial value of the **High**-variable h, the final values of x and y respectively are 0 and 1, the final value of l can be either 0 or 5, and the values of all other variables remain unchanged. Hence, for all pairs of **Low**-equal initial memories **Low**-equal final memories are reachable. Consequently, the program c_3^+ satisfies MM-Noninterference, if $\gamma_3(MM)$. ∎

Lemma 18. *The following proposition holds for the domain assignment* lev *with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\delta_3(MM) \implies \mathrm{c}_3^+ \notin NI_{MM}$.

Proof: The final value of the **Low**-variable l can only be 5 if the initial value of the **High**-variable h is 0, because $\mathbf{store}_3$ l 0

definitely updates l to 0 and there are only two instructions that can update l to 5. The two instructions are $\mathbf{store}_{11}$ l 5 and $\mathbf{store}_{15}$ l 5. The instruction $\mathbf{store}_{11}$ l 5 is in the **then**-branch of $\mathbf{if}_9$. The **then**-branch of $\mathbf{if}_9$ is dead code. The instruction $\mathbf{store}_{15}$ l 5 is in the **else**-branch of $\mathbf{if}_{13}$. The **else**-branch of $\mathbf{if}_{13}$ is only reachable if the initial value of h is 0. Thus the program does not satisfy MM-Noninterference. In the following we present the arguments in detail.

The **else**-branch of $\mathbf{if}_{13}$ r_4 is only reachable if the initial value of h is 0, because the obligation of $\mathbf{load}_8$ r_4 h is fulfilled before $\mathbf{if}_{13}$ is executed, the obligation of $\mathbf{load}_8$ r_4 h causes an update of r_4 to the initial value of h, and there is no other update of r_5. The obligation of $\mathbf{load}_8$ r_4 h causes an update of r_4 to the initial value of h, because there is no update of h in the program.

The **then**-branch of $\mathbf{if}_9$ is dead code, because the value of r_3 is always 0. The value of r_3 is always 0, because r_3 is initialized with 0 and only the obligation of $\mathbf{and}_7$ r_3 r_1 r_2 causes an update of r_3. The obligation of $\mathbf{and}_7$ r_3 r_1 r_2 always causes an update of r_3 to 0, because either r_1 or r_2 is alsways 0 when executing $\mathbf{and}_7$. This is due to the fact that both registers are initialized with 0 and only the obligations of $\mathbf{load}_6$ r_1 x and $\mathbf{load}_5$ r_2 y respectively cause an update of x and y. At least one of the obligations causes an update of its respective register to 0. Which of the two obligations causes an udpate of its respective register to 0 depends on the obligation that is fulfilled directly after the obligation of $\mathbf{spawn}_4$. Either the obligation of $\mathbf{load}_5$ r_2 y or the obligation of $\mathbf{store}_{16}$ x 0 must be fulfilled directly after the obligation of $\mathbf{spawn}_4$. This is due to the fact that these two instructions belong to different threads and the obligations of their subsequent instructions cannot be fulfilled before the obligations of these two instructions. The subsequent instruction of $\mathbf{load}_5$ r_2 y is $\mathbf{load}_6$ r_1 x and $isRead$ holds for the obligation of $\mathbf{load}_6$ r_1 x. Since $\delta_3(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{load}_6$ r_1 x, because $isRead$ also holds for the obligation of $\mathbf{load}_5$ r_2 y. The subsequent instruction of $\mathbf{store}_{16}$ x 0 is $\mathbf{store}_{17}$ y 1 and $isWrite$ holds for the obligation of $\mathbf{store}_{17}$ y 1. Since $\delta_3(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{store}_{17}$ y 1, because $isWrite$ also holds for the obligation of $\mathbf{store}_{16}$ x 0.

We distinguish two cases based on the obligation that is fulfilled after the obligation of $\mathbf{spawn}_4$ and show that either r_1 or r_2 is always 0.

Case ($\mathbf{load}_5$ r_2 y):
> In this case, the value of r_2 is always 0, because r_2 is initialized with 0, only the obligation of $\mathbf{load}_5$ r_2 y causes an update of r_2 and the update sets r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_2$ y 0 causes the most recent update of y and the update sets y to 0. The obligation of $\mathbf{store}_2$ y 0 causes the most recent update of y, because only the obligation of $\mathbf{store}_{17}$ y 1 causes a further update of y, the obligation of $\mathbf{store}_2$ y 0 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligations of $\mathbf{load}_5$ r_2 y and $\mathbf{store}_{17}$ y 1 must be fulfilled after the obligation of $\mathbf{spawn}_4$, and the obligation of $\mathbf{load}_5$ r_2 y is fulfilled directly after the obligation of $\mathbf{spawn}_4$ according to the assumption of this case.

Case ($\mathbf{store}_{16}$ x 0):
> In this case, the value of r_1 is always 0, because r_1 is initialized with 0, only the obligation of $\mathbf{load}_6$ r_1 x causes an update of r_1, and the update sets r_1 to the value of x. The value of x is 0, because the obligation of $\mathbf{store}_{16}$ x 0 causes the most recent udpate of x and the update sets x to 0. The obligation of $\mathbf{store}_{16}$ x 0 causes the most recent udpate of x, because only the obligation of $\mathbf{store}_1$ x 1 causes a further update of x, the obligation of $\mathbf{store}_1$ x 1 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligations of $\mathbf{load}_6$ r_1 x and $\mathbf{store}_{16}$ x 0 must be fulfilled after the obligation of $\mathbf{spawn}_4$, and the obligation of $\mathbf{store}_{16}$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$ according to the assumption of this case.

Based on these observations we construct a concrete counter example: We choose initial memories mem_1 and mem_1' such that $mem_1(x) = 0$ for all $x \in \mathcal{X}$, $mem_1'(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $mem_1'(\mathrm{h}) = 23$. The memories mem_1 and mem_1' satisfy $mem_1 =_L mem_1'$, because $mem_1(x) = 0 = mem_1'(x)$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $lev(\mathrm{h}) = \mathbf{High}$. From mem_1 a final memory mem_2 is reachable with $mem_2(\mathrm{l}) = 5$, because the initial value of h is 0. All final memories mem_2' that are reachable from mem_1' satisfy $mem_2'(\mathrm{l}) \neq 5$, because the initial value of h is 23 and not 0. Thus, for all final memories mem_2' that are reachable from mem_1', we have $mem_2 \neq_L mem_2'$, because $mem_2(\mathrm{l}) = 5 \neq mem_2'(\mathrm{l})$ and $lev(\mathrm{l}) = \mathbf{Low}$.

Consequently, the program c_3^+ does not satisfy MM-Noninterference, if $\delta_3(MM)$ holds. ■

Lemma 19. *The following proposition holds for the domain assignment lev with $lev(\mathrm{h})$ and $lev(x) = \mathbf{Low}$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$:* $\gamma_3(MM) \implies \mathrm{c}_3^- \notin NI_{MM}$.

Proof: The two instructions $\mathbf{load}_8$ and $\mathbf{store}_9$ in the **then**-branch of $\mathbf{if}_7$ form a direct leak, because $\mathbf{load}_8$ reads the value of the **High**-variable h into r_4, and $\mathbf{store}_9$ subsequently writes the value of r_4 into the **Low**-variable l. Due to this direct leak and the fact that this **then**-branch is reachable, the program c_3^- does not satisfy MM-noninterference. In the following we present the arguments in detail.

Let $t = 1, 2, 3, \mathbf{12}, 4, 5, 6, \mathbf{11}$ be a sequence of natural numbers. It is possible to fulfill the obligations of instructions with these natural numbers as identifiers up to reaching $\mathbf{if}_7$ in the order given by t, because all obligations, except the obligations of $\mathbf{store}_{11}$ y 1 and $\mathbf{store}_{12}$ y 1, are fulfilled in the order in which their instructions are executed. Since $\gamma_3(MM)$ holds, $\phi_{\mathrm{WW}} \in \Phi$ for MM. Thus, the obligation ob of $\mathbf{store}_{11}$ x 0 may be fulfilled after the obligation ob' of $\mathbf{store}_{12}$ y 1 due to ϕ_{WW}, because

$isWrite(ob)$ evaluates to true, $isWrite(ob')$ evaluates to true and $sinks(ob) \cap sinks(ob') = \emptyset$ holds. We will now argue why the instructions $\mathbf{load}_8$ and $\mathbf{store}_9$ will be executed and their obligations fulfilled after t.

The sequence t always leads to a register state reg_7 with $reg_7(\mathrm{r}_1) = 1$, $reg_7(\mathrm{r}_2) = 1$ and $reg_7(\mathrm{r}_3) = 1$, because of three reasons. First, the obligation of $\mathbf{load}_4$ r_2 y causes the last update of r_2 and the update sets r_2 to the value of y. The value of y is 1, because the obligation of $\mathbf{store}_{12}$ y 1 causes the most recent update of y and the update sets y to 1. Second, the obligation of $\mathbf{load}_5$ r_1 x causes the last update of r_1 and the update sets r_1 to the value of x. The value of x is 1, because the obligation of $\mathbf{store}_1$ x 1 causes the most recent update of r_1 and the update sets x to 1. Third, the obligation of $\mathbf{and}_6$ r_3 r_1 r_2 causes the last update of r_3 and the update sets r_3 to 1, because the obligations of $\mathbf{load}_5$ r_1 x and $\mathbf{load}_4$ r_2 y respectively cause the most recent updates of r_1 and r_2 and both updates set their respective registers to 1 (as we have argued before). Consequently, the $\mathbf{then}$-branch of $\mathbf{if}_7$ r_3 is taken after t. This means that the instructions $\mathbf{load}_8$ r_4 h and $\mathbf{store}_9$ l r_4 are executed and their obligations fulfilled after t.

We choose an initial memory mem_1 with $mem_1(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $mem_1(\mathrm{h}) = 5$. The final value of l is 5, because only the obligation of $\mathbf{store}_9$ l r_4 causes an update of l and the update sets l to the value of r_4. The value of r_4 is 5, because the obligation of $\mathbf{load}_8$ r_4 h causes an update of r_4 and the update sets r_4 to the initial value of h. The update sets r_4 to the initial value of h, because no instruction in the program updates h. The initial value of h is 5, i.e. $mem_1(\mathrm{h}) = 5$, according to the initial memory we have chosen. The obligation of $\mathbf{load}_8$ r_4 h must be fulfilled before the obligation of $\mathbf{store}_9$ l r_4, because both obligations are caused by the same thread and access r_4.

We now show that there is an initial memory mem_1' with $mem_1 =_L mem_1'$ for which 5 is not a possible final value for l. We choose an initial value mem_1' with $mem_1'(x) = 0$ for all $x \in \mathcal{X}$. From the definition of $\mathbf{Low}$-equality we know that $mem_1 =_L mem_1'$, because $mem_1(x) = mem_1'(x)$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$ and $lev(\mathrm{h}) = \mathbf{High}$. For the initial memory mem_1' the final value of l must be in $\{0, 1\}$, because the initial value of all variables is 0, all constants that appear in the program are either 0 or 1, and the only computation, i.e. $\mathbf{and}$ has $\{0, 1\}$ as range of values. Consequently, the final value of l cannot be 5. Hence, $mem_2'(\mathrm{l}) \neq 5 = mem_2(\mathrm{l})$ holds for all final memories mem_2' that are reachable from mem_1'.

This means that there is no final memory reachable from mem_1' that is $\mathbf{Low}$-equal to mem_2. Consequently, the program c_3^- does not satisfy MM-Noninterference, if $\gamma_3(MM)$ holds. ■

Lemma 20. *The following proposition holds for the domain assignment lev with* $lev(\mathrm{h})$ *and* $lev(x) = \mathbf{Low}$ *for all* $x \in \mathcal{X} \setminus \{\mathrm{h}\}$*:* $\delta_3(MM) \implies \mathrm{c}_3^- \in NI_{MM}$.

Proof: Only $\mathbf{load}_8$ in the $\mathbf{then}$-branch of $\mathbf{if}_7$ reads a $\mathbf{High}$-variable. The $\mathbf{then}$-branch of $\mathbf{if}_7$ is dead code. Since the only instruction that reads a $\mathbf{High}$-variable is dead code, the program c_3^- satisfies MM-noninterference. In the following we present the arguments in detail.

The instruction $\mathbf{load}_8$ is dead code, because it is in the $\mathbf{then}$-branch of $\mathbf{if}_7$ r_3 and the value of r_3 of the spawned thread is always 0. The value of r_3 is always 0, because it is initialized with 0 and only the obligation of $\mathbf{and}_6$ r_3 r_1 r_2 causes an update of r_3. The obligation of $\mathbf{and}_6$ r_3 r_1 r_2 causes an update of r_3 to 0, because the value of either r_1 or r_2 is always 0 in all possible sequences for fulfilling obligations.

We now show that the value of either r_1 or r_2 is always 0 in all possible sequences for fulfilling obligations. All possible sequences for fulfilling obligations up to (inclusively) the obligation of $\mathbf{spawn}_3$ have the same effect on the global state, because each of the obligations of $\mathbf{store}_1$ x 1 and $\mathbf{store}_2$ y 0 causes an update of a different variable and both obligations must be fulfilled before the obligation of $\mathbf{spawn}_3$. Only two obligations can be fulfilled directly after the obligation of $\mathbf{spawn}_3$: The obligation of either $\mathbf{load}_4$ r_2 y or $\mathbf{store}_{11}$ x 0 must be fulfilled directly after the obligation of $\mathbf{spawn}_3$. This is due to the fact that these two instructions belong to different threads and the obligations of their subsequent instructions cannot be fulfilled before the obligations of these two instructions. The subsequent instruction of $\mathbf{load}_4$ r_2 y is $\mathbf{load}_5$ r_1 x and $isRead$ holds for the obligation of $\mathbf{load}_5$ r_1 x. Since $\delta_3(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{load}_5$ r_1 x, because $isRead$ also holds for the obligation of $\mathbf{load}_4$ r_2 y. The subsequent instruction of $\mathbf{store}_{11}$ x 0 is $\mathbf{store}_{12}$ y 1 and $isWrite$ holds for the obligation of $\mathbf{store}_{12}$ y 1. Since $\delta_3(MM)$ holds, $next_\Phi$ cannot hold for the obligation of $\mathbf{store}_{12}$ y 1, because $isWrite$ also holds for the obligation of $\mathbf{store}_{11}$ x 0.

We distinguish two cases. In the first case, the obligation of $\mathbf{load}_4$ r_2 y is fulfilled directly after the obligation of $\mathbf{spawn}_3$. In the second case, the obligation of $\mathbf{store}_{11}$ x 0 is fulfilled directly after the obligation of $\mathbf{spawn}_4$.

Case ($\mathbf{load}_4$ r_2 y):

In this case, the value of r_2 is always 0, because r_2 is initialized with 0, only the obligation of $\mathbf{load}_4$ r_2 y causes an update of r_2, and the update sets r_2 to the value of y. The value of y is 0, because the obligation of $\mathbf{store}_2$ y 0 causes the most recent update of y and the update sets y to 0. The obligation of $\mathbf{store}_2$ y 0 causes the most recent update of y, because the obligation of $\mathbf{store}_2$ y 0 must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligation of $\mathbf{load}_4$ r_2 y and $\mathbf{store}_{12}$ y 1 must be fulfilled after the obligation of $\mathbf{spawn}_4$, and the obligation of $\mathbf{load}_4$ r_2 y is fulfilled directly after the obligation of $\mathbf{spawn}_4$ according to the assumption of this case.

Case ($\mathbf{store}_{11}$ x 0):

In this case, the the value of r_1 is always 0, because r_1 is initialized with 0, only the obligation of $\mathbf{load}_5\ \mathrm{r}_1\ \mathrm{x}$ causes an update of r_1, and the update sets r_1 to the value of x. The value of x is 0, because the obligation of $\mathbf{store}_{11}\ \mathrm{x}\ 0$ causes the most recent update of x, and the update sets x to 0. The obligation of $\mathbf{store}_{11}\ \mathrm{x}\ 0$ causes the most recent update of x, because the obligation of $\mathbf{store}_1\ \mathrm{x}\ 0$ must be fulfilled before the obligation of $\mathbf{spawn}_4$ while the obligations of $\mathbf{load}_5\ \mathrm{r}_1\ \mathrm{x}$ and $\mathbf{store}_{11}\ \mathrm{x}\ 0$ must be fulfilled after the obligation of $\mathbf{spawn}_4$, and the obligation of $\mathbf{store}_{11}\ \mathrm{x}\ 0$ is fulfilled directly after the obligation of $\mathbf{spawn}_4$ according to the assumption of this case.

Since $\mathbf{load}_8\ \mathrm{h}\ \mathrm{r}_4$ is dead code and no other instruction reads a **High**-variable the program does not read information from **High**-variables in any program run. Consequently, the program c_3^- satisfies MM-Noninterference, if $\delta_3(MM)$ holds. ■

For easier reference we recall Theorem 1 from the article:

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

Proof: This follows from Lemmas 4, 5, 6 and 7 that show which of the discrimnating conditions from δ_l and γ_l with $l \in \{1,2,3\}$ each of the memory models satisfy and Lemmas 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 and 20 that provide concrete counter examples against each implication. ■

C. Proofs for Program Typing

In this section we recall the transforming type system from the article and introduce a type-check system that we use to back the soundness proof of our transformation. We show that a program that is transformed with our transforming type system results in a program that is typeable with the type-check system. The transforming type system from the article is recalled in Figure 4. The type-check system is defined in Figure 5. Note that the rules [CSK], [CFN], [CLC], [CLX], [COP], [CST], [CSP],

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

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

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

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

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

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

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

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

Figure 4. Transforming security type system for SC, IBM370, TSO, and PSO

[CSQ], [CIL], [CIH] and [CWH] for type-checking have the same preconditions like the transformation rules [SK], [FN], [LC], [LX], [OP], [ST], [SP], [SQ], [IL], [IH] and [WH]. There is no corresponding rule for the transformation rule [IT], because this is the rule that inserts the fences in our transformation. There are additional rules for type-checking the terminated instruction ϵ without any constraints, for typing an instruction with a lower final path-type if it is type-checkable with a higher final path-type, for type-checking a list of obligations to check that all obligations adhere to the information-flow policy and to check whether the path-type constitues a lower bound on the security level of variables and registers that will be updated due to obligations in the path.

The following lemma shows that a program that is obtained with the transformation (Figure 4) is typeable with the type-check system (Figure 5).

Lemma 21 (Transformed Programs can be Typechecked)**.** *If $pc, pt \vdash_{lev} c \diamond (pt', c')$ is derivable, then $pc, pt \vdash_{lev} c' \diamond (pt')$ is derivable.*

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

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

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

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

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

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

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

$$[\text{SB}]\frac{pc, pt \vdash_{lev} c \diamond (pt'') \qquad pt' \sqsubseteq pt''}{pc, pt \vdash_{lev} c \diamond (pt')} \qquad [\text{EM}]\frac{}{pc, pt \vdash_{lev} \epsilon \diamond (pt)}$$

$$[\text{PE}]\frac{}{pt \vdash_{lev} []} \qquad [\text{PF}]\frac{\exists pt' \in \{\mathbf{Low}, \mathbf{High}\}.pt' \vdash_{lev} obs}{pt \vdash_{lev} obs::[\|]} \qquad [\text{PS}]\frac{pt \sqcup lev(r) \sqsubseteq lev(x) \qquad pt \vdash_{lev} obs}{pt \vdash_{lev} obs::[x \twoheadleftarrow v@r]}$$

$$[\text{PL}]\frac{pt \sqcup lev(x) \sqsubseteq lev(r) \qquad pt \vdash_{lev} obs}{pt \vdash_{lev} obs::[x@v \twoheadrightarrow r]} \qquad [\text{PC}]\frac{pt \sqsubseteq lev(r) \qquad pt \vdash_{lev} obs}{pt \vdash_{lev} obs::[v@\text{const} \circlearrowright r]}$$

$$[\text{PV}]\frac{\begin{array}{c}pt \sqcup lev(r_1) \sqcup lev(r_2) \sqsubseteq lev(r) \\ op \in \{\text{eq}, \text{and}\} \qquad pt \vdash_{lev} obs\end{array}}{pt \vdash_{lev} pa::[v@op(r_1, r_2) \circlearrowright r]} \qquad [\text{PT}]\frac{\begin{array}{c}pt \sqsubseteq \mathbf{Low} \qquad pt \vdash_{lev} obs \\ \exists pc, pt', pt'' \in \{\mathbf{Low}, \mathbf{High}\}.pc, pt' \vdash_{lev} c \diamond (pt'')\end{array}}{pt \vdash_{lev} obs::[\nearrow_c]}$$

$$[\text{CS}]\frac{\begin{array}{c}pre(\vec{cs}) = pre(\vec{pc}) = pre(\vec{pt}) \\ \forall i \in pre(\vec{cs}).\exists pt' \in \{\mathbf{Low}, \mathbf{High}\}.\vec{pc}(i), \vec{pt}(i) \vdash_{lev} \vec{cs}(i) \diamond (pt') \wedge \vec{pt}(i) \vdash_{lev} pa\restriction_i\end{array}}{\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}, (pa, \vec{tr}), gst \rangle}$$

Figure 5. Type-check System

Proof: The gist of this proof is that the transforming type system inserts a **fence**, if $lev(r) = \mathbf{High}$ and $pt = \mathbf{Low}$ holds for $\mathbf{if}_\iota\ r$. When performing the type check $pt = \mathbf{High}$ holds when the same $\mathbf{if}_\iota\ r$ is checked, because the rule for **fence** raises pt to **High** and the rule for sequential composition propagates this into the type check for the subsequent $\mathbf{if}_\iota\ r$. The detailed argument works as follows.

Let lev, pc, pt, c, c' and pt' be arbitrary such that $pc, pt \vdash_{lev} c \diamond (pt', c')$ is derivable. We show by an induction on the derivation length of $pc, pt \vdash_{lev} c \diamond (pt', c')$ that $pc, pt \vdash_{lev} c' \diamond (pt')$ is derivable.

For the induction base let the derivation length of $pc, pt \vdash_{lev} c \diamond (pt', c')$ be 1. Only with the rules [SK], [FN], [LC], [LX], [OP] and [ST] a derivation length of 1 is possible. We distinguish these cases.

Case ([SK]):
From the assumption of this case we get by the rule [SK] that $c = c' = \mathbf{skip}_\iota$ and $pt' = pt$. From $c' = \mathbf{skip}$ we get by the rule [CSK] that $pc, pt \vdash_{lev} c' \diamond (pt)$ is derivable.

Case ([FN]):
From the assumption of this case we get by the rule [FN] that $c = c' = \mathbf{fence}_\iota$ and $pt' = \mathbf{High}$. From $c' = \mathbf{fence}_\iota$ we get by the rule [CFN] that $pc, pt \vdash_{lev} c' \diamond (\mathbf{High})$ is derivable.

Case ([LC]):
From the assumption of this case we get by the rule [LC] that $c = c' = \mathbf{load}_\iota\ r\ v$, $v \in \mathcal{V}$, $pc \sqsubseteq lev(r)$ and $pt' = pt \sqcap lev(r)$. From $c' = \mathbf{load}_\iota\ r\ v$, $v \in \mathcal{V}$ and $pc \sqsubseteq lev(r)$ we get by the rule [CLC] that $pc, pt \vdash_{lev} c' \diamond (pt \sqcap lev(r))$ is derivable.

Case ([LX]):
From the assumption of this case we get by the rule [LX] that $c = c' = \mathbf{load}_\iota\ r\ x$, $x \in \mathcal{X}$, $lev(x) \sqcup pc \sqsubseteq lev(r)$

and $pt' = pt \sqcap lev(r)$. From $c' = \mathbf{load}_\iota\ r\ x$, $x \in \mathcal{X}$ and $lev(x) \sqcup pc \sqsubseteq lev(r)$ we get by the rule [CLX] that $pc, pt \vdash_{lev} c' \diamond (pt \sqcap lev(r))$ is derivable.

Case ([OP]):

From the assumption of this case we get by the rule [OP] that $c = c' = op_\iota\ r_1\ r_2\ r_3$, $op \in \{\mathbf{and}, \mathbf{eq}\}$, $lev(r_2) \sqcup lev(r_3) \sqcup pc \sqsubseteq lev(r_1)$ and $pt' = pt \sqcap lev(r_1)$. From $c = c' = op_\iota\ r_1\ r_2\ r_3$, $op \in \{\mathbf{and}, \mathbf{eq}\}$ and $lev(r_2) \sqcup lev(r_3) \sqcup pc \sqsubseteq lev(r_1)$ we get by the rule [COP] that $pc, pt \vdash_{lev} c' \diamond (pt \sqcap lev(r_1))$ is derivable.

Case ([ST]):

From the assumption of this case we get by the rule [ST] that $c = c' = \mathbf{store}_\iota\ x\ r$, $lev(r) \sqcup pc \sqsubseteq lev(x)$ and $pt' = pt \sqcap lev(x)$. From $c' = \mathbf{store}_\iota\ x\ r$ and $lev(r) \sqcup pc \sqsubseteq lev(x)$ we get by the rule [CST] that $pc, pt \vdash_{lev} c' \diamond (pt \sqcap lev(x))$ is derivable.

As induction hypothesis we assume that $pc, pt \vdash_{lev} c \diamond (pt', c')$ implies that $pc, pt \vdash_{lev} c \diamond (pt', c')$ is derivable for all $pc, pt \vdash_{lev} c \diamond (pt', c')$ with an arbitrary derivation length $n \geq 1$.

For the induction step let lev, pc, pt, c, c' and pt' be arbitrary such that $pc, pt \vdash_{lev} c \diamond (pt', c')$ is derivable in $n' = n + 1$ steps. From $n \geq 1$ and $n' = n + 1$ we get $n' \geq 2$. Only with the rules [SP], [IL], [IH], [IT], [WH] and [SQ] a derivation length of $n' \geq 2$ is possible. We distinguish these cases.

Case ([SP]):

From the assumption of this case we get by the rule [SP] that $c = \mathbf{spawn}_\iota c''$, $c' = \mathbf{spawn}_\iota c'''$, $pc = \mathbf{Low}$, $pt' = \mathbf{Low}$, and $pc, pt'' \vdash_{lev} c'' \diamond (pt''', c''')$.

From $pc, pt'' \vdash_{lev} c'' \diamond (pt''', c''')$ and the fact that the derivation length of $pc, pt'' \vdash_{lev} c'' \diamond (pt''', c''')$ is $n' - 1 = n$ we get by the induction hypothesis that $pc, pt'' \vdash_{lev} c''' \diamond (pt''')$ is derivable.

From $c' = \mathbf{spawn}_\iota c'''$, $pc = \mathbf{Low}$, $pt' = \mathbf{Low}$ and $pc, pt' \vdash_{lev} c''' \diamond (pt'')$ we get by the rule [CSP] that $\mathbf{Low}, pt \vdash_{lev} c' \diamond (\mathbf{Low})$ is derivable.

Case ([IL]):

From the assumption of this case we get by the rule [IL] that $lev(r) = \mathbf{Low}$, $c = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}$, $c' = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$, $pt' = pt'' \sqcap pt'''$, $pc, pt \vdash_{lev} c_1 \diamond (pt'', c_1')$ and $pc, pt \vdash_{lev} c_2 \diamond (pt''', c_2')$.

From $pc, pt \vdash_{lev} c_1 \diamond (pt'', c_1')$, $pc, pt \vdash_{lev} c_2 \diamond (pt''', c_2')$ and the fact that the derivation length of these two judgments is at most $n' - 1 = n$ we get by the induction hypothesis that $pc, pt \vdash_{lev} c_1' \diamond (pt'')$ and $pc, pt \vdash_{lev} c_2' \diamond (pt''')$ is derivable.

From $c' = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$, $lev(r) = \mathbf{Low}$, $pt' = pt'' \sqcap pt'''$, $pc, pt \vdash_{lev} c_1' \diamond (pt'')$ and $pc, pt \vdash_{lev} c_2' \diamond (pt''')$ we get by the rule [CIL] that $pc, pt \vdash_{lev} c' \diamond (pt'' \sqcap pt''')$ is derivable.

Case ([IH]):

From the assumption of this case we get by the rule [IH] that $lev(r) = pt = \mathbf{High}$, $c = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}$, $c' = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$, $pt' = \mathbf{High}, \mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High}, c_1')$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High}, c_2')$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High}, c_1')$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High}, c_2')$ and the fact that the derivation length of these two judgments is at most $n' - 1 = n$ we get by the induction hypothesis that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1' \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2' \diamond (\mathbf{High})$ is derivable.

From $c' = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$, $lev(r) = pt = \mathbf{High}$, $pt' = \mathbf{High}$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1' \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2' \diamond (\mathbf{High})$ we get by the rule [CIH] that $pc, \mathbf{High} \vdash_{lev} c' \diamond (\mathbf{High})$ is derivable.

Case ([IT]):

From the assumption of this case we get by the rule [IT] that $lev(r) = \mathbf{High}$, $pt = \mathbf{Low}$, $c = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}$, $c' = \mathbf{fence}_{\iota'};\ \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$, ι' is fresh, $pt' = \mathbf{High}, \mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High}, c_1')$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High}, c_2')$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High}, c_1')$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High}, c_2')$ and the fact that the derivation length of these two judgments is at most $n' - 1 = n$ we get by the induction hypothesis that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1' \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2' \diamond (\mathbf{High})$ is derivable.

From $\mathbf{fence}_{\iota'}$ we get by the rule [CFN] that $pc, pt \vdash_{lev} \mathbf{fence}_{\iota'} \diamond (\mathbf{High})$ is derivable.

From $lev(r) = \mathbf{High}, \mathbf{High}, \mathbf{High} \vdash_{lev} c_1' \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2' \diamond (\mathbf{High})$ we get by the rule [CIH] that $pc, \mathbf{High} \vdash_{lev} \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi} \diamond (\mathbf{High})$ is derivable.

From $c' = \mathbf{fence}_{\iota'};\ \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$, $pt' = \mathbf{High}$, $pc, pt \vdash_{lev} \mathbf{fence}_{\iota'} \diamond (\mathbf{High})$ and $pc, \mathbf{High} \vdash_{lev} \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi} \diamond (\mathbf{High})$ we get by the rule [CSQ] that $pc, pt \vdash_{lev} c' \diamond (\mathbf{High})$ is derivable.

Case ([WH]):

From the assumption of this case we get by the rule [WH] that $c = \mathbf{while}_\iota\ r\ \mathbf{do}\ c''\ \mathbf{od}$, $c' = \mathbf{while}_\iota\ r\ \mathbf{do}\ c'''\ \mathbf{od}$, $lev(r) = \mathbf{Low}$, $pt' = pt \sqcap pt''$ and $pc, \mathbf{Low} \vdash_{lev} c'' \diamond (pt'', c''')$.

From $pc, \mathbf{Low} \vdash_{lev} c'' \diamond (pt'', c''')$ and the fact that the derivation length of this judgments is $n' - 1 = n$ we get by the induction hypothesis that $pc, \mathbf{Low} \vdash_{lev} c''' \diamond (pt'')$ is derivable.

From $c' = \mathbf{while}_\iota\ r\ \mathbf{do}\ c'''\ \mathbf{od}$, $lev(r) = \mathbf{Low}$, $pt' = pt \sqcap pt''$ and $pc, \mathbf{Low} \vdash_{lev} c''' \diamond (pt'')$ we get by the rule [CWH] that $pc, pt \vdash_{lev} c' \diamond (pt')$ is derivable.

Case ([SQ]):

From the assumption of this case we get by the rule [SQ] that $c = c_1;\ c_2$, $c' = c_1';\ c_2'$, $pc, pt \vdash_{lev} c_1 \diamond (pt'', c_1')$ and $pc, pt'' \vdash_{lev} c_2 \diamond (pt', c_2')$.

From $pc, pt \vdash_{lev} c_1 \diamond (pt'', c_1')$ and $pc, pt'' \vdash_{lev} c_2 \diamond (pt', c_2')$ and the fact that the derivation length of these two judgments is at most $n'-1 = n$ we get by the induction hypothesis that $pc, pt \vdash_{lev} c_1' \diamond (pt'')$ and $pc, pt'' \vdash_{lev} c_2' \diamond (pt')$ is derivable.
From $c' = c_1';\ c_2'$, $pc, pt \vdash_{lev} c_1' \diamond (pt'')$ and $pc, pt'' \vdash_{lev} c_2' \diamond (pt')$ we get by the rule [CSQ] that $pc, pt \vdash_{lev} c' \diamond (pt')$ is derivable.

■

The following lemma shows admissable subtypings for the programcounter and the path-type.

Lemma 22 (Subtyping)**.** *The following propositions hold:*

1) $pc, pt \vdash_{lev} c \diamond (pt') \implies pc', pt \vdash_{lev} c \diamond (pt')$ *for* $pc, pc', pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$ *with* $pc' \sqsubseteq pc$ *holds for all* $c \in \mathcal{C}$.
2) $pt \vdash_{lev} obs \implies pt' \vdash_{lev} obs$ *for* $pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$ *with* $pt' \sqsubseteq pt$ *holds for all* $obs \in Ob^*$.

Proof:

1) This follows from the fact that all requirements that compare pc and pc' to another security domain require that pc and pc', respectivley, are smaller or equal than the security domain to which it is compared.
2) This follows from the fact that all requirements that compare pt and pt' to another security domain require that pt and pt', respectively, are smaller or equal than the security domain to which it is compared.

■

The following lemma shows properties that hold for decomposing and composing lists of obligations.

Lemma 23. *If* $pt_1 \vdash_{lev} obs_1$ *and* $pt_2 \vdash_{lev} obs_2$ *are derivable, then the following two propositions hold:*

- $pt_1' \vdash_{lev} obs_1 \setminus k$ *for some* $k < |obs_1| - 1$ *and some* $pt_1' \in \{\mathbf{Low}, \mathbf{High}\}$*, and*
- $pt_1 \sqcap pt_2 \vdash_{lev} obs_1 :: obs_2$.

Proof:

- This follows from the fact that the typing of an obligation list iterates over the complete list and for each element in the list the same rule remains applicable after removing another element from the list.
- This follows from the fact that the typing of an obligation list iterates over the complete list and condition 2 in Lemma 22.

■

The following lemma shows that an execution step of a **High**-typed instruction results in a **High**-typeable instruction again.

Lemma 24. *If* $\langle c_1, pa_1, reg\rangle \rightarrow_i \langle c_2, pa_2\rangle$ *and* $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (pt)$*, then* $pt = \mathbf{High}$ *and* $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

Proof: We prove this by an induction on the derivation length of $\langle c_1, pa_1, reg\rangle \rightarrow_i \langle c_2, pa_2\rangle$.

Let $c_1 \in \mathcal{C}$, $pa_1, pa_2 \in Pa$, $reg \in Reg$ and $c_2 \in (\mathcal{C} \cup \{\epsilon\})$ be arbitrary such that $\langle c_1, pa_1, reg\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (pt)$.

By inspection of the calculus we see immediately that all rules that update the pathtype can only reduce the pathtype to **Low** if $pc = \mathbf{Low}$, but from the assumption of this lemma we have $pc = \mathbf{High}$. Hence, $pt = \mathbf{High}$.

The induction base are derivations with a length of 1. Derivations with length 1 are possible for all instructions that are not sequentially composed.

We distinguish two cases based on whether $c_2 = \epsilon$ or $c_2 \in \mathcal{C}$.

Case ($c_2 = \epsilon$):
In this case we get from $c_2 = \epsilon$ by rule [EM] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.
Case ($c_2 \in \mathcal{C}$):
In this case we get from $c_2 \in \mathcal{C}$ that c_2 is either an **if** or a **while**. From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$ we know that c_1 cannot be a **while**. Hence, $c_1 = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_t\ \mathbf{else}\ c_e\ \mathbf{fi}$.
From the rule [CIL] (in combination with typing rules that update the path using the same argument about pc as before) and [CIH] we get $\mathbf{High}, \mathbf{High} \vdash_{lev} c \diamond (\mathbf{High})$ for $c \in \{c_t, c_e\}$.

As induction hypothesis we assume that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (pt)$ and $pt = \mathbf{High}$ holds for all derivations of $\langle c_A, pa_1, reg\rangle \rightarrow_i \langle c_C, pa_2\rangle$ with an arbitrary length n.

For the induction step let the derivation length of $\langle c_1, pa_1, reg\rangle \rightarrow_i \langle c_2, pa_2\rangle$ be $n' = n + 1$. From the semantics we know that derivations with a length greater than 1 are only possible with sequential composition.

From semantics of sequential composition we get that $c_1 = c_A;\ c_B$ and $\langle c_A, pa_1, reg\rangle \rightarrow_i \langle c_C, pa_2\rangle$ is derivable with $n' - 1 = n$ steps. From $c_1 = c_A;\ c_B$ we get by [SQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (pt)$ and $\mathbf{High}, pt \vdash_{lev} c_B \diamond (\mathbf{High})$. From this we get by the induction hypothesis that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (pt)$ and $pt = \mathbf{High}$.

From semantics of sequential composition we also get $c_2 = c_B$ or $c_2 = c_C;\ c_B$. From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_B \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (\mathbf{High})$ we get either directly or by rule [SQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$. ■

The following lemma shows that executions steps of a typeable instruction with a typeable list of obligations result in typable instructions and lists of obligations again.

Lemma 25. *If $pc, pt \vdash_{lev} c \diamond (pt')$ and $pt \vdash_{lev} pa \restriction_i$ and $\langle c, pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$ are derivable for some $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$, then $pc, pt'' \vdash_{lev} c' \diamond (pt''')$ and $pt'' \vdash_{lev} pa' \restriction_i$ are derivable for some $pt'', pt''' \in \{\mathbf{Low}, \mathbf{High}\}$.*

Proof: Let $i \in \mathcal{I}$, $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$, $c \in \mathcal{C}$, $c' \in \mathcal{C} \cup \{\epsilon\}$ and $pa, pa' \in Pa$ be arbitrary with $pc, pt \vdash_{lev} c \diamond (pt')$ and $pt \vdash_{lev} pa \restriction_i$ and $\langle c, pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$ for some $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$.

We prove that there is $pt'', pt''' \in \{\mathbf{Low}, \mathbf{High}\}$ such that $pc, pt'' \vdash_{lev} c' \diamond (pt''')$ and $pt'' \vdash_{lev} pa' \restriction_i$ by an induction over the length of the derivation of $pc, pt \vdash_{lev} c \diamond (pt')$. The induction base are derivations with a length of 1.

We make a case distinction on the rules for which the judgment $pc, pt \vdash_{lev} c \diamond (pt')$. can be derived in one step, i.e. [CSK], [CFN], [CLC], [CLX], [COP], [CST].

Case ([EM]):
In this case $\langle \epsilon, pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$ cannot be derived and therefore this case cannot apply.

Case ([SK]):
In this case we know that $c = \mathbf{skip}_\iota$ and from semantics of **skip** that $c' = \epsilon$ and $pa' = pa$.
From rule [CSK] we get $pt' = pt$. From $pa' = pa$ and $pt' = pt$ and $pt \vdash_{lev} pa \restriction_i$ we get $pt' \vdash_{lev} pa' \restriction_i$.
From rule [EM] we get that $pc, pt' \vdash_{lev} \epsilon \diamond (pt')$ is derivable.

Case ([FN]):
In this case we know that $c = \mathbf{fence}_\iota$ and from semantics of **fence** that $c' = \epsilon$ and $pa' = pa{::}[(i, \|)]$.
From rule [CFN] we get $pt' = \mathbf{High}$. From $pa' = pa{::}[(i, \|)]$ and $pt \vdash_{lev} pa \restriction_i$ we get by rule [PF] that $\mathbf{High} \vdash_{lev} pa{::}[(i, \|)] \restriction_i$ is derivable.
From rule [EM] we get that $pc, pt' \vdash_{lev} \epsilon \diamond (pt')$ is derivable.

Case ([CLC]):
In this case we know that $c = \mathbf{load}_\iota\ r\ v$ and from semantics of **load** that $c' = \epsilon$ and $pa' = pa{::}[(i, v@\mathrm{const} \circlearrowleft r)]$.
From rule [CLC] we get $pt' = pt \sqcap lev(r)$. From $pt' = pt \sqcap lev(r)$ we get $pt' \sqsubseteq lev(r)$. From $pt' \sqsubseteq lev(r)$ and $pa' = pa{::}[(i, v@\mathrm{const} \circlearrowleft r)]$ and $pt \vdash_{lev} pa \restriction_i$ we get by rule [PC] and Lemma 22 that $pt' \vdash_{lev} pa{::}[(i, v@\mathrm{const} \circlearrowleft r)] \restriction_i$. Hence, $pt' \vdash_{lev} pa' \restriction_i$.
From rule [EM] we get that $pc, pt' \vdash_{lev} \epsilon \diamond (pt')$ is derivable.

Case ([CLX]):
In this case we know that $c = \mathbf{load}_\iota\ v\ x$ and from semantics of **load** that $c' = \epsilon$ and $pa' = pa{::}[(i, ?@x \twoheadrightarrow r)]$.
From rule [CLX] we get $pt' = pt \sqcap lev(r)$ and $lev(x) \sqsubseteq lev(r)$. From $pt' = pt \sqcap lev(r)$ and $lev(x) \sqsubseteq lev(r)$ we get $pt' \sqcup lev(x) \sqsubseteq lev(r)$. From $pt' = pt \sqcap lev(r)$ and $pt' \sqcup lev(x) \sqsubseteq lev(r)$ and $pa' = pa{::}[(i, ?@x \twoheadrightarrow r)]$ and $pt \vdash_{lev} pa \restriction_i$ we get by rule [PL] and Lemma 22 that $pt' \vdash_{lev} pa{::}[(i, ?@x \twoheadrightarrow r)]$. Hence, $pt' \vdash_{lev} pa'$.
From rule [EM] we get that $pc, pt' \vdash_{lev} \epsilon \diamond (pt')$ is derivable.

Case ([CST]):
In this case we know that $c = \mathbf{store}_\iota\ x\ r$ and from semantics of **store** that $c' = \epsilon$ and $pa' = pa{::}[(i, x \twoheadleftarrow v@r)]$.
From rule [CST] we get $pt' = pt \sqcap lev(x)$ and $lev(r) \sqsubseteq lev(x)$. From $pt' = pt \sqcap lev(x)$ and $lev(r) \sqsubseteq lev(x)$ we get $pt' \sqcup lev(r) \sqsubseteq lev(x)$. From $pt' = pt \sqcap lev(x)$ and $pt' \sqcup lev(r) \sqsubseteq lev(x)$ and $pa' = pa{::}[(i, x \twoheadleftarrow v@r)]$ and $pt \vdash_{lev} pa$ we get by rule [PS] and Lemma 22 that $pt' \vdash_{lev} pa{::}[(i, x \twoheadleftarrow v@r)]$. Hence, $pt' \vdash_{lev} pa'$.
From rule [EM] we get that $pc, pt' \vdash_{lev} \epsilon \diamond (pt')$ is derivable.

Case ([OP]):
In this case we know that $c = opc_\iota\ r_1\ r_2\ r_3$ for some $opc \in \{\mathbf{and}, \mathbf{eq}\}$, $c' = \epsilon$ and $pa' = pa{::}[(i, v@opo(r_2, r_3) \circlearrowleft r_1)]$ for some $opo \in \{\mathrm{eq}, \mathrm{and}\}$.
From rule [COP] we get $pt' = pt \sqcap lev(r_1)$ and $lev(r_2) \sqcup lev(r_3) \sqsubseteq lev(r_1)$. From $pt' = pt \sqcap lev(r)$ we get $pt' \sqsubseteq lev(r)$. From $pt' = pt \sqcap lev(r)$, $pt' \sqsubseteq lev(r)$, $lev(r_2) \sqcup lev(r_3) \sqsubseteq lev(r_1)$, $pa' = pa{::}[(i, v@opo(r_2, r_3) \circlearrowleft r_1)]$ and $pt \vdash_{lev} pa \restriction_i$ we get by rule [PV] and Lemma 22 that $pt' \vdash_{lev} pa{::}[(i, v@opo(r_2, r_3) \circlearrowleft r_1)] \restriction_i$. Hence, $pt' \vdash_{lev} pa' \restriction_i$.
From rule [EM] we get that $pc, pt' \vdash_{lev} \epsilon \diamond (pt')$ is derivable.

As induction hypothesis we assume that the proposition from the Lemma holds for derivations of the judgment with an arbitrary length $n' \geq 1$. For the induction step let the induction length be $n = n' + 1$. From $n' \geq 1$ and $n = n' + 1$ we get $n \geq 1$. We make a case distinction over the rules for which the judgment $pc, pt \vdash_{lev} c \diamond (pt')$ can be derived in $n \geq 2$ steps, i.e. [CSP], [CSQ], [CIL], [CIH], [CWH].

Case ([CSP]):

In this case we know that $c = \mathbf{spawn}_\iota c_S$ for some $c_S \in \mathcal{C}$ and from semantics of **spawn** that $c' = \epsilon$ and $pa' = pa{::}[(i, \nearrow_{c_S})]$.

From rule [CSP] we know that $pc', pt'' \vdash_{lev} c_S \diamond (pt''')$ is derivable for some $pc', pt'', pt''' \in \{\mathbf{Low}, \mathbf{High}\}$ and $pt' = \mathbf{Low}$. From $pt' = \mathbf{Low}$ we get $pt' \sqsubseteq pt$. From $pt \vdash_{lev} pa\restriction_i$ we get by Lemma 22 that $pt' \vdash_{lev} pa\restriction_i$. From $pc', pt'' \vdash_{lev} c_S \diamond (pt''')$, $pt' \vdash_{lev} pa\restriction_i$ and $pt' = \mathbf{Low}$ we get by rule [PT] that $pt' \vdash_{lev} pa{::}[(i, \nearrow_{c_S})]\restriction_i$. Thus, from $pa' = pa{::}[(i, \nearrow_{c_S})]$ we get $pt' \vdash_{lev} pa'\restriction_i$.

From rule [EM] we get that $pc, pt' \vdash_{lev} \epsilon \diamond (pt')$.

Case ([CSQ]):

In this case we know that $c = c_A;\ c_B$.

From rule [CSQ] we get that $pc, pt \vdash_{lev} c_A \diamond (pt_A)$ and $pc, pt_A \vdash_{lev} c_B \diamond (pt')$.

We now distinguish two cases based on the semantics rule to derive $\langle c, pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$. The two possible rule differ in shape of c'. In the first case $c' = c_B$. In the second case, $c' = c'_A;\ c_B$.

Case ($c' = c_B$):

In this case we get from semantics of sequential composition that $\langle c_A, pa, reg\rangle \rightarrow_i \langle \epsilon, pa_A\rangle$ is derivable.

From the induction hypothesis we get that $pt_A \vdash_{lev} pa'$ is derivable.

Since $pc, pt_A \vdash_{lev} c_B \diamond (pt')$ we are done in this case.

Case ($c' = c'_A;\ c_B$):

In this case we get from semantics of sequential composition that $\langle c_A, pa, reg\rangle \rightarrow_i \langle c'_A, pa'\rangle$ is derivable.

From the induction hypothesis we get that $pt'_A \vdash_{lev} pa'$ and $pc, pt'_A \vdash_{lev} c'_A \diamond (pt''_A)$ are derivable. From the typing rules we know that either $pt''_A = pt_A$ or $pt_A \sqsubseteq pt''_A$, because on the resulting path type is always a lower bound of the possible resulting path types for the instruction resulting after one step. If $pt_A \sqsubseteq pt''_A$ we get from rule [SB] that $pc, pt'_A \vdash_{lev} c'_A \diamond (pt_A)$.

From $pc, pt'_A \vdash_{lev} c_A \diamond (pt_A)$ and $pc, pt_A \vdash_{lev} c_B \diamond (pt')$ we get by rule [CSQ] that $pc, pt'_A \vdash_{lev} c'_A;\ c_B \diamond (pt')$.

Case ([CIL]):

In this case we know that $c = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}$ and from the semantics of **if** that $c' = c_a$ for some $a \in \{1, 2\}$ and $pa' = pa$.

From rules [CIL] we get that $pc, pt \vdash_{lev} c_a \diamond (pt'')$ with $pt' \sqsubseteq pt''$ for $a \in \{1, 2\}$. From $pt' \sqsubseteq pt''$ we get from rule [SB] that $pc, pt'_A \vdash_{lev} c_a \diamond (pt'_A)$.

Since $pa = pa'$ and $pt \vdash_{lev} pa$ we are done in this case.

Case ([CIH]):

In this case we know that $c = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}$ and from the semantics of **if** that $c' = c_a$ for some $a \in \{1, 2\}$, $pa' = pa$ and $pt' = \mathbf{High}$.

From rules [CIH] we get that $pt = \mathbf{High}$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_a \diamond (\mathbf{High})$ with $pt' = \mathbf{High}$ for $a \in \{1, 2\}$.

Since $pa = pa'$ and $pt = \mathbf{High}$, $pt \vdash_{lev} pa$ we are done in this case.

Case ([CWH]):

In this case we know that $c = \mathbf{while}_\iota\ r\ \mathbf{do}\ c_A\ \mathbf{od}$ and $pa = pa'$.

We now distinguish two cases based on the semantics rule to derive $\langle c, pa, reg\rangle \rightarrow_i \langle c', pa'\rangle$. The two possible rules differ in the shape of c'. In the first case $c' = \epsilon$. In the second case, $c' = c_A;\ c$.

Case ($c' = \epsilon$):

From [EM] we get $pc, pt \vdash_{lev} \epsilon \diamond (pt)$.

Since $pa' = pa$ and $pt \vdash_{lev} pa\restriction_i$ we are done in this case.

Case ($c' = c_A;\ c$):

From rule [CWH] we get that $lev(r) = \mathbf{Low}$ and $pc, \mathbf{Low} \vdash_{lev} c_A \diamond (pt'')$ is derivable for some pt''.

Since $c' = c_A;\ c$ and $pc, \mathbf{Low} \vdash_{lev} c_A \diamond (pt'')$ we must show that $pc, pt'' \vdash_{lev} \mathbf{while}_\iota\ r\ \mathbf{do}\ c_A\ \mathbf{od} \diamond (pt''')$ is derivable to satisfy the requirements of rule [CSQ].

From $lev(r) = \mathbf{Low}$ and $pc, \mathbf{Low} \vdash_{lev} c_A \diamond (pt'')$ we get that $pc, pt'' \vdash_{lev} \mathbf{while}_\iota\ r\ \mathbf{do}\ c_A\ \mathbf{od} \diamond (pt''')$ is derivable for $pt''' = pt''$.

Since $pa' = pa$ and $pt \vdash_{lev} pa\restriction_i$ we are done in this case.

■

The following lemma shows that execution steps from typeable global configurations result in typeable configurations again.

Lemma 26. *If $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}, (pa, \vec{tr}), gst\rangle$ and $\langle \vec{cs}, (pa, \vec{tr}), gst\rangle \Longrightarrow_{MM} \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle$ are derivable, then $\vec{pc}', \vec{pt}' \vdash_{lev} \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle$ is derivable for some $\vec{pc}', \vec{pt}' : \mathcal{I} \rightharpoonup$ with $pre(\vec{pc}') = pre(\vec{pt}') = pre(\vec{cs}')$.*

Proof: Let $\vec{pc}, \vec{pt} : \mathcal{I} \rightharpoonup \{\mathbf{Low}, \mathbf{High}\}$, $\vec{cs}, \vec{cs}' : \mathcal{I} \rightharpoonup (\mathcal{C} \cup \{\epsilon\})$, $pa, pa' \in Pa$, $\vec{tr}, \vec{tr}' \in \vec{Tr}$ and $gst, gst' \in Gst$ be arbitrary such that $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}, (pa, \vec{tr}), gst\rangle$ and $\langle \vec{cs}, (pa, \vec{tr}), gst\rangle \Longrightarrow_{MM} \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle$ are derivable.

From $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}, (pa, \vec{tr}), gst\rangle$ we get by the rule [CS] that for all $i \in pre(\vec{cs})$ the judgments $\vec{pc}(i), \vec{pt}(i) \vdash_{lev} \vec{cs}(i) \diamond (pt')$

and $\vec{pt}(i) \vdash_{lev} pa\restriction_i$ are derivable for some $pt' \in \{\mathbf{Low}, \mathbf{High}\}$. We fix the thread identifier $i \in pre(\vec{cs})$ that causes the execution step $\langle \vec{cs}, (pa, \vec{tr}), gst\rangle \Longrightarrow_{MM} \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle$.

We make a case distinction over the 4 semantics rules in Figure 5 of the article.

Case (thread progress):
In this case $\vec{cs}'(j) = \vec{cs}(j)$ for all $j \in pre(\vec{cs})$ with $j \neq i$ and $pa'\restriction_j = pa\restriction_j$ for all $j \in pre(\vec{cs})$ with $j \neq i$.
From semantics of thread progress we get that $\langle \vec{cs}(i), pa, \vec{reg}(i)\rangle \rightarrow_i \langle \vec{cs}'(i), pa'\rangle$ where $gst = (\vec{reg}, mem)$ is derivable. From $\vec{pc}(i), \vec{pt}(i) \vdash_{lev} \vec{cs}(i) \diamond (pt')$ and $\vec{pt}(i) \vdash_{lev} pa\restriction_i$ and $\langle \vec{cs}(i), pa, \vec{reg}(i)\rangle \rightarrow_i \langle \vec{cs}'(i), pa'\rangle$ we get by Lemma 25 that $\vec{pc}(i), pt'' \vdash_{lev} \vec{cs}'(i) \diamond (pt''')$ and $pt'' \vdash_{lev} pa'\restriction_i$ are derivable for some $pt'', pt''' \in \{\mathbf{Low}, \mathbf{High}\}$.
From $\vec{cs}'(j) = \vec{cs}(j)$ and $pa'\restriction_j = pa\restriction_j$ and $\vec{pc}(j), \vec{pt}(j) \vdash_{lev} \vec{cs}(j) \diamond (pt')$ and $\vec{pt}(j) \vdash_{lev} pa\restriction_j$ for some $\vec{pt}$ for all $j \in pre(\vec{cs})$ with $j \neq i$ and $\vec{pc}(i), pt'' \vdash_{lev} \vec{cs}'(i) \diamond (pt''')$ and $pt'' \vdash_{lev} pa'\restriction_i$ for some pt'', pt''' we get $\vec{pc}', \vec{pt}' \vdash_{lev} \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle$ for some $\vec{pc}'$ and $\vec{pt}'$.

Case (fulfill memory update):
In this case $\vec{cs}' = \vec{cs}$ and $pa' = pa \setminus k$, where $pa[k] = (i, ob)$, for some $k < |pa| - 1$.
From $pa' = pa \setminus k$ and $pa[k] = (i, ob)$ we get $pa'\restriction_i = pa\restriction_i \setminus k'$ for some $k' < |pa\restriction_i| - 1$ and $pa'\restriction_j = pa\restriction_j$ for all $j \in pre(\vec{cs})$ and $j \neq i$. Thus we get from $\vec{pt}(i) \vdash_{lev} pa\restriction_i$ by Lemma 23 that $pt'' \vdash_{lev} pa'\restriction_i$ for some pt'' is derivable.
From $\vec{cs} = \vec{cs}'$, $\vec{pc}(j), \vec{pt}(j) \vdash_{lev} \vec{cs}(j) \diamond (pt')$ for all $j \in pre(\vec{cs})$, $\vec{pt}(j) \vdash_{lev} pa\restriction_j$ for all $j \in pre(\vec{cs})$ and $pt'' \vdash_{lev} pa'\restriction_i$ for some pt'' we get $\vec{pc}', \vec{pt}' \vdash_{lev} \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle$ for some $\vec{pc}'$ and $\vec{pt}'$.

Case (fulfill fence):
This case is analogous to the case "fulfill memory update".

Case (fulfill thread creation):
In this case the following propositions hold:
$\vec{cs}'(j) = \vec{cs}(j)$ for all $j \in pre(\vec{cs})$, $pre(\vec{cs}') = pre(\vec{cs}) \cup \{j \mid j = max(pre(\vec{cs})) + 1\}$, $\vec{cs}'(max(pre(\vec{cs})) + 1) = c$ and $pa' = pa \setminus k$, where $pa[k] = (i, \nearrow_c)$, for some $k < |pa| - 1$.
From the typing rule of **spawn** we get $pc, pt'' \vdash_{lev} c \diamond (pt''')$ for some $pt'', pt''' \in \{\mathbf{Low}, \mathbf{High}\}$. From semantics we get that $pa'\restriction_{(max(pre(\vec{cs}))+1)} = []$. From $pa'\restriction_{(max(pre(\vec{cs}))+1)} = []$ we get by rule [PE] that $pt \vdash_{lev} pa'\restriction_{(max(pre(\vec{cs}))+1)}$ for some pt is derivable.
From $pa' = pa \setminus k$ and $pa[k] = (i, ob)$ we get $pa'\restriction_i = pa\restriction_i \setminus k'$ for some $k' < |pa\restriction_i| - 1$ and $pa'\restriction_j = pa\restriction_j$ for all $j \in pre(\vec{cs})$ with $j \neq i$.
From $\vec{cs}(j) = \vec{cs}'(j)$ for all $j \in pre(\vec{cs})$, $\vec{pc}(j), \vec{pt}(j) \vdash_{lev} \vec{cs}(j) \diamond (pt')$ for some pt' for all $j \in pre(\vec{cs})$, $pc, pt'' \vdash_{lev} c \diamond (pt''')$ for some $pt'', pt''' \in \{\mathbf{Low}, \mathbf{High}\}$, $\vec{pt}(j) \vdash_{lev} pa\restriction_j$ for all $j \in pre(\vec{cs})$ with $j \neq i$, $pt'' \vdash_{lev} pa'\restriction_i$ for some pt'', $pt \vdash_{lev} pa'\restriction_{(max(pre(\vec{cs}))+1)}$ for some pt we get $\vec{pc}', \vec{pt}' \vdash_{lev} \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle$ for some $\vec{pc}'$ and $\vec{pt}'$.

■

D. Definitions and Proofs for Equivalences based on Typing

In this section, we introduce some auxiliary definitions for establishing noninterference that relate global states, lists of obligations, paths and commands. We use these definitions to reason about individual steps of related programs. We use superscripts in the symbols to indicate the domains of a relation, i.e. the universe on which binary relations are defined, to distinguish between different relations with the same symbol.

The next two definitions relate local memories and global states that agree on their **Low** parts.

Definition 1. *Two register states $reg, reg' \in Reg$ are* **Low***-equal, denoted by $reg =_L^{\mathcal{R}} reg'$, if $reg(r) = reg'(r)$ holds for all $r \in \mathcal{R}$ with $lev(r)$.*

Definition 2. *Two global states $(\vec{reg}, mem), (\vec{reg}', mem') \in Gst$ are* **Low***-equal, denoted by $(\vec{reg}, mem) =_L^{Gst} (\vec{reg}', mem')$ if they satisfy the following three conditions:*

1) $pre(\vec{reg}) = pre(\vec{reg}')$, *and*
2) $\vec{reg}(i) =_L^{\mathcal{R}} \vec{reg}'(i)$ *holds for all* $i \in pre(\vec{reg})$, *and*
3) $mem =_L mem'$.

Lemma 27. *The relations $=_L$, $=_L^{\mathcal{R}}$ and $=_L^{Gst}$ are equivalence relations.*

Proof: That the relations $=_L$, $=_L^{\mathcal{R}}$ and $=_L^{Gst}$ are equivalence relations follows from the fact that each of the definitions is defined with the identity relation $=$ on subsets of the respective domains and that $pre(\vec{reg}) = pre(\vec{reg}')$ is required for $=_L^{Gst}$. Hence, reflexivity, symmetry and transitivity follow from reflexivity, symmetry and transitivity of $=$. ■

We consider two obligations as **Low**-equal, if they agree on all sources and sinks and in addition on the value, if the sink of the obligation is **Low**.

Definition 3. *Two obligations $ob, ob' \in Ob$ are* **Low***-equal, denoted by $ob =_L^{obs} ob'$, if they satisfy the following conditions:*

- $(ob \in Fe \vee ob' \in Fe) \implies ob = ob'$, *and*
- $\exists xr \in (sinks(ob) \cup sinks(ob')).lev(xr) = \mathbf{Low} \implies ob = ob'$, *and*
- $\forall xr \in (sinks(ob) \cup sinks(ob')).lev(xr) = \mathbf{High} \implies sources(ob) = sources(ob') \wedge sinks(ob) = sinks(ob')$.

Lemma 28. *The relation $=_L^{obs}$ is an equivalence relation.*

Proof: That the relation $=_L^{obs}$ is an equivalence relation follows from the fact that each condition is defined only using identity. Hence, each condition is reflexive, transitive and symmetric due to reflexivity, transitivity and symmetry of identity. ■

Definition 4. *The predicate $fencefree$ on Ob^* is defined by $fencefree(obs) \equiv \forall k < (|obs| - 1).obs \notin Fe$.*

The predicate evaluates to true if the list of obligations does not contain any obligation that works as a fence.

We consider two typeable lists of obligations as **Low** similar, if they have a (possibly empty) prefix that causes only updates **High** variables and **High** registers and agree on the suffix with respect to what register and variables are read, and which values are written to **Low** variables and **Low** registers.

Definition 5. *Two lists of obligations $obs, obs' \in Ob^*$ are* **Low***-similar, denoted by $obs \approx_L^{Ob^*} obs'$, if there are $obs_A, obs'_A, obs_B, obs'_B \in Ob^*$ such that the following conditions are satisfied:*

1) $obs = obs_A :: obs_B \wedge obs' = obs'_A :: obs'_B$, *and*
2) $fencefree(obs_A) \wedge fencefree(obs'_A)$, *and*
3) $\mathbf{High} \vdash_{lev} obs_A \wedge \mathbf{High} \vdash_{lev} obs'_A$, *and*
4) $pt \vdash_{lev} obs_B \wedge pt \vdash_{lev} obs'_B$ *for some* $pt \in \{\mathbf{Low}, \mathbf{High}\}$, *and*
5) $(obs_B = [||] \wedge obs'_B = []) \vee (obs_B = [] \wedge obs'_B = [||]) \vee (|obs_B| = |obs_B| \wedge \forall k < (|obs_B|).obs_B[k] =_L^{obs} obs'_B[k])$.

Lemma 29. *The relation $\approx_L^{Ob^*}$ is symmetric and transitive.*

Proof: The definition of $\approx_L^{Ob^*}$ is a conjunction of 5 conditions. The first four conditions are conditions on how to split the obligation lists for the last conditions. These conditions are conditions on obligation lists that can be related by this relation (i.e. restricting the domain of the relation to a subset of the set of all obligations). Each of the conditions is symmetric and transitive since it puts identical requirements on the obligation list on each side of the relation.

The last condition is a disjunction of three conditions. The first two are symmetric counterparts. The third condition requires identical length of both sublists of obligations (which is symmetric) and point-wise **Low**-equality of the elements at each position in the sublists (which is symmetric, because $=_L^{obs}$ is an equivalence relation). Hence, the condition is symmetric. The condition is also transitive, because of two reasons. First, the third condition is transitive, due to $=_L^{obs}$ being an equivalence relation. Second, if the pair (obs_B, obs'_B) satisfy one of the first two conditions, then the pair (obs'_B, obs''_B) can only satisfy the other of the two first conditions or the third condition. If the pair (obs'_B, obs''_B) satisfies the other of the first two conditions, then the pair (obs_B, obs''_B) satisfies the third condition. If the pair (obs'_B, obs''_B) satisfies the third condition, then the pair (ob_B, ob''_B) satisfies the same condition as the pair (ob_B, ob'_B). This chain of arguments can be repeated arbitrarily often to show that the condition is transitive. ■

Definition 6. *Two pairs of an instruction and a list of obligations $(c, obs), (c', obs') \in (\mathcal{C} \cup \{\epsilon\}) \times (Ob^*)$ are* **Low***-similar, denoted by $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c', obs')$, if for some $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$ it holds that $pc, pt \vdash_{lev} c \diamond (pt')$ and $pc, pt \vdash_{lev} c' \diamond (pt')$ and $pt \vdash_{lev} obs$ and $pt \vdash_{lev} obs'$ and $obs \approx_L^{Ob^*} obs'$ and one of the following conditions is satisfied:*

- $c = c'$, *or*
- $pc = pt = pt' = \mathbf{High}$,
- $c = c_A;\ c_B \wedge c' = c'_A;\ c'_B \wedge c_B = c'_B \wedge (c_A, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_A, obs')$ *for some* $c_A, c'_A, c_B, c'_B \in \mathcal{C}$.

Lemma 30. *The relation $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ is transitive and symmetric.*

Proof:

Symmetry:

We show that $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ is symmetric by an induction on the shape of c.
Let $c, c', c'' \in \mathcal{C}$ and $obs, obs', obs'' \in Ob^*$ be arbitrary with $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c', obs')$ and $(c', obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$.
For the induction base let c be an instruction that is not a sequential composition, i.e. c is not of the shape $c_A;\ c_B$.

From $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c', obs')$ we get $pc, pt \vdash_{lev} c \diamond (pt')$, $pc, pt \vdash_{lev} c' \diamond (pt')$, $pt \vdash_{lev} obs$, $pt \vdash_{lev} obs'$, $obs \approx_L^{Ob^*} obs'$ and $c = c'$ or $pc = pt = pt' = \mathbf{High}$. From this combined with symmetry of $\approx_L^{Ob^*}$ and $=$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c', obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c, obs)$.
For the induction step let $c = c_A;\ c_B$. We distinguish three cases based on whether $c = c'$ or $pc = pt = pt' = \mathbf{High}$ or $c = c_A;\ c_B \wedge c' = c'_A;\ c'_B \wedge c_B = c'_B \wedge (c_A, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_A, obs')$ for some $c_A, c'_A, c_B, c'_B \in \mathcal{C}$. The first two cases are analogous to the induction base. The argument for the third case follows.
From $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c', obs')$ we get $pc, pt \vdash_{lev} c \diamond (pt')$, $pc, pt \vdash_{lev} c' \diamond (pt')$, $pt \vdash_{lev} obs$, $pt \vdash_{lev} obs'$ and $obs \approx_L^{Ob^*} obs'$. From this combined with symmetry of $\approx_L^{Ob^*}$ and $=$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $obs' \approx_L^{Ob^*} obs$.
From $c_B = c'_B$ we get by symmetry of $=$ that $c'_B = c_B$.
From $(c_A, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_A, obs')$ we get by the induction hypothesis that $(c'_A, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_A, obs)$.
Combining all these facts we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c', obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c, obs)$.

Transitivity:

Let $c, c', c'' \in \mathcal{C}$ and $obs, obs', obs'' \in Ob^*$ be arbitrary with $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c', obs')$ and $(c', obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$.
From $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c', obs')$ we get $pc, pt \vdash_{lev} c \diamond (pt')$, $pc, pt \vdash_{lev} c' \diamond (pt')$, $pt \vdash_{lev} obs$, $pt \vdash_{lev} obs'$, $obs \approx_L^{Ob^*} obs'$ and $c = c'$ or $pc = pt = pt' = \mathbf{High}$ or
From $(c', obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$ we get $pc', pt'' \vdash_{lev} c' \diamond (pt''')$, $pc', pt'' \vdash_{lev} c'' \diamond (pt''')$, $pt'' \vdash_{lev} obs'$, $pt'' \vdash_{lev} obs''$, $obs' \approx_L^{Ob^*} obs''$ and $c' = c''$ or $pc' = pt'' = pt''' = \mathbf{High}$ or $c' = c'_A;\ c'_B \wedge c'' = c''_A;\ c''_B \wedge c'_B = c''_B \wedge (c'_A, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_A, obs'')$ for some $c'_A, c''_A, c'_B, c''_B \in \mathcal{C}$.
From $obs \approx_L^{Ob^*} obs'$ and $obs' \approx_L^{Ob^*} obs''$ we get by transitivity of $\approx_L^{Ob^*}$ that $obs \approx_L^{Ob^*} obs''$.
We distinguish three cases based on whether $c = c'$ or $pc = pt = pt' = \mathbf{High}$.

Case ($c = c'$):

In this case we get from $c = c'$ and $obs \approx_L^{Ob^*} obs'$ that the pair $((c, obs), (c'', obs''))$ satisfies the same condition as $((c', obs'), (c'', obs''))$ from $c' = c''$ or $pc' = pt'' = pt''' = \mathbf{High}$ or $c' = c'_A;\ c'_B \wedge c'' = c''_A;\ c''_B \wedge c'_B = c''_B \wedge (c'_A, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_A, obs'')$ for some $c'_A, c''_A, c'_B, c''_B \in \mathcal{C}$.

Case ($pc = pt = pt' = \mathbf{High}$):

In this case we distinguish three cases based on the condition from $c' = c''$ or $pc' = pt'' = pt''' = \mathbf{High}$ or $c' = c'_A;\ c'_B \wedge c'' = c''_A;\ c''_B \wedge c'_B = c''_B \wedge (c'_A, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_A, obs'')$ for some $c'_A, c''_A, c'_B, c''_B \in \mathcal{C}$ that is satisfied.

Case ($c' = c''$):

In this case $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$ follows from the transitivity of $=$ and $\approx_L^{Ob^*}$ and the fact that identical commands and $\mathbf{Low}$-similar obligation list can be typed with identical types for the programcounter and obligations list.

Case ($pc' = pt'' = pt''' = \mathbf{High}$):

In this case we have $pc = pc' = pt = pt' = pt'' = pt''' = \mathbf{High}$. Hence, $pc, pt \vdash_{lev} c \diamond (pt')$, $pc, pt \vdash_{lev} c'' \diamond (pt')$, $pt \vdash_{lev} obs$, $pt \vdash_{lev} obs''$ and $pc = pt = pt' = \mathbf{High}$.
From this combined with $ob \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} obs''$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$.

Case ($c' = c'_A;\ c'_B \wedge c'' = c''_A;\ c''_B \wedge c'_B = c''_B \wedge (c'_A, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_A, obs'')$ for some $c'_A, c''_A, c'_B, c''_B \in \mathcal{C}$):

In this case we know from the definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that either $c'_A = c''_A$ or there is a shortest prefix c'_C of c'_A and c''_C of c''_A such that $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_C \diamond (High)$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_C \diamond (High)$, and either there are no corresponding suffixes or the corresponding suffixes c'_D and c''_D satisfy $c'_D = c''_D$. If the suffix is not empty we have $c'_D;\ c'_B = c''_D = c''_B$. Without loss of generality we assume that the suffixes are empty and hence $c'_A = c'_C$ and $c''_A = c''_C$. Hence, $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (High)$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_A \diamond (High)$.
From $pc, pt \vdash_{lev} c' \diamond (pt'')$, $pc = pt = pt' = \mathbf{High}$ and $c' = c_A;\ c_B$ we get by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (pt'_A)$ and $\mathbf{High}, pt'_A \vdash_{lev} c'_B \diamond (\mathbf{High})$. From $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (\mathbf{High})$ we get $pt'_A = \mathbf{High}$ and, hence, $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_B \diamond (\mathbf{High})$. From this combined with $c'_B = c''_B$ we get $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_B \diamond (\mathbf{High})$.
From $c'' = c''_A;\ c''_B$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_A \diamond (High)$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_B \diamond (\mathbf{High})$ we get by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c'' \diamond (\mathbf{High})$.
From $\mathbf{High}, \mathbf{High} \vdash_{lev} c'' \diamond (\mathbf{High})$ and $pc = pt = pt' = \mathbf{High}$ we get $pc, pt \vdash_{lev} c'' \diamond (pt')$.
From $pc = pt = pt' = \mathbf{High}$, $pc, pt \vdash_{lev} c \diamond (pt')$, $pc, pt \vdash_{lev} c'' \diamond (pt')$, $pt \vdash_{lev} obs$ and $obs \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} obs'$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$.

Case ($c = c_A;\ c_B \wedge c' = c'_A;\ c'_B \wedge c_B = c'_B \wedge (c_A, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_A, obs')$ for some $c_A, c'_A, c_B, c'_B \in \mathcal{C}$):
From the definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that either $c_A = c'_A$ or there is a shortest prefix c_C of c_A and c'_C of c'_A such that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (High)$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_C \diamond (High)$, and either there are no corresponding suffixes or the corresponding suffixes c_D and c'_D satisfy $c_D = c'_D$. If the suffix is not empty we have $c_D;\ c_B = c'_D = c'_B$. Without loss of generality we assume that the suffixes are empty and hence $c_A = c_C$ and $c'_A = c'_C$ (because otherwise we can deconstruct the instruction until it has this form using the recursive definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$). Hence, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (High)$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (High)$.
From $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c', obs')$ we get $pc, pt \vdash_{lev} c \diamond (pt')$, $pc, pt \vdash_{lev} c' \diamond (pt')$, $pt \vdash_{lev} obs$, $pt \vdash_{lev} obs'$ and $obs \approx_L^{Ob^*} obs'$.
From $(c', obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$ we get $pc', pt'' \vdash_{lev} c' \diamond (pt''')$, $pc', pt'' \vdash_{lev} c'' \diamond (pt''')$, $pt'' \vdash_{lev} obs'$, $pt'' \vdash_{lev} obs''$, $obs' \approx_L^{Ob^*} obs''$ and $c' = c''$ or $pc' = pt'' = pt''' = \mathbf{High}$ or $c' = c'_A;\ c'_B \wedge c'' = c''_A;\ c''_B \wedge c'_B = c''_B \wedge (c'_A, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_A, obs'')$ for some $c'_A, c''_A, c'_B, c''_B \in \mathcal{C}$.
From $obs \approx_L^{Ob^*} obs'$ and $obs' \approx_L^{Ob^*} obs''$ we get by transitivity of $\approx_L^{Ob^*}$ that $obs \approx_L^{Ob^*} obs''$.
We distinguish three cases based on the condition from $c' = c''$ or $pc' = pt'' = pt''' = \mathbf{High}$ or $c' = c'_C;\ c'_D \wedge c'' = c''_C;\ c''_D \wedge c'_D = c''_D \wedge (c'_C, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_C, obs'')$ for some $c'_A, c''_A, c'_B, c''_B \in \mathcal{C}$ that is satisfied.

Case ($c' = c''$):

In this case $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$ follows from the fact that $c'' = c'_A;\ c'_B$ (due to $c' = c''$) and $obs \approx_L^{Ob^*} obs''$.

Case ($pc' = pt'' = pt''' = \mathbf{High}$):

In this case we get from $\mathbf{High}, \mathbf{High} \vdash_{lev} c' \diamond (\mathbf{High})$, $c' = c'_A;\ c'_B$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (\mathbf{High})$ by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_B \diamond (\mathbf{High})$. From this combined with $c_B = c'_B$ we get $\mathbf{High}, \mathbf{High} \vdash_{lev} c_B \diamond (\mathbf{High})$. From this combined with $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$ and $c = c_A;\ c_B$ we get by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c \diamond (\mathbf{High})$. Hence, $pc', pt'' \vdash_{lev} c \diamond (pt''')$.
From $pc', pt'' \vdash_{lev} c \diamond (pt''')$, $pc', pt'' \vdash_{lev} c'' \diamond (pt''')$, $pc' = pt'' = pt''' = \mathbf{High}$, $pt'' \vdash_{lev} obs''$ and $obs \approx_L^{Ob^*} obs''$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$.

Case ($c' = c'_C;\ c'_D \wedge c'' = c''_C;\ c''_D \wedge c'_D = c''_D \wedge (c'_C, obs') \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_C, obs'')$ for some $c'_C, c''_C, c'_D, c''_D \in \mathcal{C}$):

In this case we know from the definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that either $c'_C = c''_C$ or there is a shortest prefix c'_E of c'_C and c''_E of c''_C such that $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_E \diamond (High)$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_E \diamond (High)$, and either there are no corresponding suffixes or the corresponding suffixes c'_F and c''_F satisfy $c'_F = c''_F$. Without loss of generality we assume that the suffixes are empty and hence $c'_C = c''_E$ and $c''_C = c''_E$. Hence, $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_C \diamond (High)$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_C \diamond (High)$.
From $c' = c'_A;\ c'_B$ and $c' = c'_C;\ c'_D$ we get that one of the instructions c'_B and c'_D is a suffix of the other. Without loss of generality we assume that c'_B is a suffix of c'_D (the other case is analogous). Hence, $c'_D = c'_E;\ c'_B$ for some c'_E. From this combined $c'_D = c''_D$ we get $c''_D = c'_E;\ c'_B$. From this combined with $c'' = c''_C;\ c''_D$ we get $c'' = c'' = c''_C;\ c'_E;\ c'_B$.
From $c' = c'_A;\ c'_B$, $c' = c'_C;\ c'_D$ and $c'_D = c'_E;\ c'_B$ we get $c'_A = c'_C;\ c'_E$.
From $c'_A = c'_C;\ c'_E$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_C \diamond (\mathbf{High})$ we get by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_E \diamond (\mathbf{High})$.
From $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_C \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_E \diamond (\mathbf{High})$ we get by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_C;\ c'_E \diamond (\mathbf{High})$.
From this combined with $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (High)$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_C;\ c'_E \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} obs$ and $obs \approx_L^{Ob^*} obs''$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_A, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_C;\ c'_E, obs'')$.
From this combined with $c_B = c'_B$ we get $(c_A;\ c_B, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c''_C;\ c'_E;\ c'_B, obs'')$.
From this combined with $c = c_A;\ c_B$ and $c'' = c''_C;\ c'_E;\ c'_B$ we get $(c, obs) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'', obs'')$.

■

We capture the definition of well formed global configurations with the following predicate:

Definition 7. *The predicate wellformed on* $(\mathcal{I} \rightharpoonup (\mathcal{C} \cup \{\epsilon\})) \times De \times Gst$ *is defined by* $wellformed(\langle \vec{cs}, (pa, \vec{tr}), (r\vec{eg}, mem)\rangle) \equiv ((pre(\vec{cs}) = pre(\vec{tr}) = pre(r\vec{eg})) \wedge \{i \mid pa\ \restriction_i \neq [] \} \subseteq pre(\vec{cs}))$.

Definition 8. *Two well formed global configurations* $cnf = \langle \vec{cs}, (pa, \vec{tr}), gst\rangle$, $cnf' = \langle \vec{cs}', (pa', \vec{tr}'), gst'\rangle \in (\mathcal{I} \rightharpoonup (\mathcal{C} \cup \{\epsilon\})) \times De \times Gst$ *are* **Low**-*similar, denoted by* $cnf \approx_L^{Conf} cnf'$, *if the following conditions are satisfied:*

1) $pre(\vec{cs}) = pre(\vec{cs}')$, *and*
2) $(\vec{cs}(i), pa \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (\vec{cs}'(i), pa' \restriction_i)$ *for all* $i \in pre(\vec{cs})$, *and*
3) $gst =_L^{Gst} gst'$.

Lemma 31. *The relation* $\approx_L^{Conf}$ *is transitive and symmetric.*

Proof: That the relation $\approx_L^{Conf}$ is transitive and symmetric follows from the fact that it is defined with a conjunction of three transitive and symmetric conditions. That the first condition is transitive and symmetric follows form transitivity and symmetry of $=$. That the second condition is transitive and symmetric follows from the point-wise comparison with the relation $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ which is transitive and symmetric. That the third condition is transitive and symmetric follows from the fact that $=_L^{Gst}$ is transitive and symmetric. ∎

E. Proofs for Soundness

When we argue about $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ or $\approx_L^{Conf}$ after an execution step in this section we omit the argument that the resulting instruction and list of obligations is typeable with identical path type, because this always follows from Lemma 26.

The following lemma shows that identical instructions that perform execution steps starting in two configurations with **Low**-similar lists of obligations and **Low**-equal local memories result in **Low**-similar pairs of instructions and lists of obligations.

Lemma 32 (Same Instruction in **Low**-similar Configurations)**.** *If* $c_1 = c_1'$, $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1' \restriction_i)$, $reg_1 =_L^{\mathcal{R}} reg_1'$, $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$, $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$, $fencefree(pa_1 \restriction_i)$ *and* $fencefree(pa_1' \restriction_i)$, *then* $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2' \restriction_i)$, $pa_1 \restriction_j = pa_2 \restriction_j$ *and* $pa_1' \restriction_j = pa_2' \restriction_j$ *for all* $j \neq i$.

Proof: We prove this lemma by an induction on the derivation length of the judgment $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$. The induction base are derivations with a length of 1. Derivations with length 1 are possible for all instructions that are not sequentially composed.

Let $reg_1, reg_1' \in Reg$, $pa_1, pa_1', pa_2, pa_2' \in Pa$, $c_1, c_1', c_2 \in \mathcal{C}$ be arbitrary with $reg_1 =_L^{\mathcal{R}} reg_1'$, $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1' \restriction_i)$, $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$, $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$, $fencefree(pa_1 \restriction_i)$ and $fencefree(pa_1' \restriction_i)$.

We make a case distinction on the shapes of c_1 for which a derivation in one step is possible, i.e. all instructions except sequential composition.

Case ($c_1 = \mathbf{skip}_\iota$):
In this case $c_1' = \mathbf{skip}_\iota$ follows from $c_1 = c_1'$.
From $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$ we get by the semantics of **skip** that $c_2 = \epsilon$, $pa_2 = pa_1$, $c_2' = \epsilon$ and $pa_2' = pa_1'$. From $c_2 = \epsilon$, $pa_2 = pa_1$, $c_2' = \epsilon$, $pa_2' = pa_1'$ and $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1' \restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2' \restriction_i)$.
From $pa_2 = pa_1$ and $pa_2' = pa_1'$ we get $pa_2 \restriction_j = pa_1 \restriction_j$ and $pa_2' \restriction_j = pa_1' \restriction_j$ for all $j \neq i$.

Case ($c_1 = \mathbf{while}_\iota\ r\ \mathbf{do}\ c'\ \mathbf{od}$):
In this case, $c_1' = \mathbf{while}_\iota\ r\ \mathbf{do}\ c'\ \mathbf{od}$ follows from $c_1 = c_1'$.
From $pc, pt \vdash_{lev} c_1 \diamond (pt_1')$ we get by the typing rule for **while** that $lev(r) = \mathbf{Low}$. From $reg_1 =_L^{\mathcal{R}} reg_1'$ and $lev(r) = \mathbf{Low}$ we get $reg_1(r) = reg_1'(r)$.
We distinguish two cases based on whether $reg_1(r) = 0$ or $reg_1(r) \neq 0$.

Case ($reg_1(r) = 0$):
Since $reg_1(r) = reg_1'(r)$ we know that only the rule for **while** where the register is 0 can be applied to derive $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$. From this rule we obtain $c_2 = \epsilon$, $c_2' = \epsilon$, $pa_2 = pa_1$ and $pa_2' = pa_1'$.
From $c_2 = \epsilon$, $pa_2 = pa_1$, $c_2' = \epsilon$, $pa_2' = pa_1'$ and $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1' \restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2' \restriction_i)$.
From $pa_2 = pa_1$ and $pa_2' = pa_1'$ we get $pa_2 \restriction_j = pa_1 \restriction_j$ and $pa_2' \restriction_j = pa_1' \restriction_j$ for all $j \neq i$.

Case ($reg_1(r) \neq 0$):
Since $reg_1(r) = reg_1'(r)$ we know that only the rule for **while** where the register is not 0 can be applied to derive $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$. From this rule we obtain $c_2 = c'$; c_1, $c_2' = c'$; c_1', $pa_2 = pa_1$ and $pa_2' = pa_1'$.
From $c_1 = c_1'$, $c_2 = c'$; c_1, $pa_2 = pa_1$, $c_2' = c'$; c_1', $pa_2' = pa_1'$ and $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1' \restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2' \restriction_i)$.
From $pa_2 = pa_1$ and $pa_2' = pa_1'$ we get $pa_2 \restriction_j = pa_1 \restriction_j$ and $pa_2' \restriction_j = pa_1' \restriction_j$ for all $j \neq i$.

Case ($c_1 = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_t\ \mathbf{else}\ c_e\ \mathbf{fi}$):

In this case, $c_1' = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_t\ \mathbf{else}\ c_e\ \mathbf{fi}$ follows from $c_1 = c_1'$.

We distinguish two cases based on the security domain of register r.

Case ($lev(r) = \mathbf{Low}$):

From $reg_1 =_L^{\mathcal{R}} reg_1'$ and $lev(r) = \mathbf{Low}$ we get $reg_1(r) = reg'(r)_1$.

We distinguish two cases based on whether $reg_1(r) \neq 0$ or $reg_1(r) = 0$.

Case ($reg_1(r) \neq 0$):

Since $reg_1(r) = reg_1'(r)$ we know that only the rule for **if** where the register is not 0 can be applied to derive $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$. From this rule we obtain $c_2 = c_t$, $c_2' = c_t$, $pa_2 = pa_1$ and $pa_2' = pa_1'$.

From $c_2 = c_t$, $pa_2 = pa_1$, $c_2' = c_t$, $pa_2' = pa_1'$ and $(c_1, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1'\restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2'\restriction_i)$.

From $pa_2 = pa_1$ and $pa_2' = pa_1'$ we get $pa_2\restriction_j = pa_1\restriction_j$ and $pa_2'\restriction_j = pa_1'\restriction_j$ for all $j \neq i$.

Case ($reg_1(r) = 0$):

Since $reg_1(r) = reg_1'(r)$ we know that only the rule for **if** where the register is 0 can be applied to derive $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$. From this rule we obtain $c_2 = c_e$, $c_2' = c_e$, $pa_2 = pa_1$ and $pa_2' = pa_1'$.

From $c_2 = c_e$, $pa_2 = pa_1$, $c_2' = c_e$, $pa_2' = pa_1'$ and $(c_1, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1'\restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2'\restriction_i)$.

From $pa_2 = pa_1$ and $pa_2' = pa_1'$ we get $pa_2\restriction_j = pa_1\restriction_j$ and $pa_2'\restriction_j = pa_1'\restriction_j$ for all $j \neq i$.

Case ($lev(r) = \mathbf{High}$):

From the two rules for **if** in Figure 7 of the article we get $c_2 = c_A$ and $c_2' = c_B$ for some $A, B \in \{t, e\}$, $pa_2 = pa_1$ and $pa_2' = pa_1'$.

From $(c_1, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1'\restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $pc, pt \vdash_{lev} c_1 \diamond (pt')$, $pc, pt \vdash_{lev} c_1' \diamond (pt'')$, $pt \vdash_{lev} pa_1\restriction_i$ and $pt \vdash_{lev} pa_1'\restriction_i$. From $lev(r) = \mathbf{High}$ we get that $pc, pt \vdash_{lev} c_1 \diamond (pt')$ and $pc, pt \vdash_{lev} c_1 \diamond (pt'')$ are derived with rule [CIH]. Hence, from rule [CIH] we get $pt = \mathbf{High}$.

From [CIH] we also get $\mathbf{High}, \mathbf{High} \vdash_{lev} c_t \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_f \diamond (\mathbf{High})$.

From $c_2 = c_A$ and $c_2' = c_B$ for some $A, B \in \{t, e\}$, $pa_2 = pa_1$ and $pa_2' = pa_1'$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_t \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_f \diamond (\mathbf{High})$, $pt \vdash_{lev} pa_1\restriction_i$ and $pt \vdash_{lev} pa_1'\restriction_i$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2'\restriction_i)$.

From $pa_2 = pa_1$ and $pa_2' = pa_1'$ we get $pa_2\restriction_j = pa_1\restriction_j$ and $pa_2'\restriction_j = pa_1'\restriction_j$ for all $j \neq i$.

Case ($c_1 = \mathbf{load}_\iota\ r\ v$):

In this case, $c_1' = \mathbf{load}_\iota\ r\ v$ follows from $c_1 = c_1'$.

From $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1, pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$ we get by the semantics rules of **load** with a constant from Figure 7 in the article that $c_2 = \epsilon$, $c_2' = \epsilon$, $pa_2 = pa_1{::}[(i, v@\text{const}\ \circlearrowleft\ r)]$ and $pa_2' = pa_1'{::}[(i, v@\text{const}\ \circlearrowleft\ r)]$. Hence, $pa_2\restriction_j = pa_1\restriction_j$ and $pa_2'\restriction_j = pa_1'\restriction_j$ for all $j \neq i$.

From $(c_1, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1'\restriction_i)$ we get $pa_1\restriction_i \approx_L^{Ob^*} pa_1'\restriction_i$. From this combined with $fencefree(pa_1\restriction_i)$ and $fencefree(pa_1'\restriction_i)$ we get by definition of $\approx_L^{Ob^*}$ that $pa_1{::}[(i, v@\text{const}\ \circlearrowleft\ r)]\restriction_i \approx_L^{Ob^*} pa_1'{::}[(i, v@\text{const}\ \circlearrowleft\ r)]\restriction_i$. Hence, $pa_2\restriction_i \approx_L^{Ob^*} pa_2'\restriction_i$.

From $pa_2\restriction_i \approx_L^{Ob^*} pa_2'\restriction_i$, $c_2 = \epsilon$ and $c_2' = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2'\restriction_i)$.

Case ($c_1 = \mathbf{load}_\iota\ r\ x$):

In this case, $c_1' = \mathbf{load}_\iota\ r\ x$ follows from $c_1 = c_1'$.

From $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$ we get by the semantics rules of **load** with a constant from Figure 7 in the article that $c_2 = \epsilon$, $c_2' = \epsilon$, $pa_2 = pa_1{::}[(i, ?@x \rightarrowtriangle r)]$ and $pa_2' = pa_1'{::}[(i, ?@x \rightarrowtriangle r)]$. Hence, $pa_2\restriction_j = pa_1\restriction_j$ and $pa_2'\restriction_j = pa_1'\restriction_j$ for all $j \neq i$.

From $(c_1, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_1', pa_1'\restriction_i)$ we get $pa_1\restriction_i \approx_L^{Ob^*} pa_1'\restriction_i$. From this combined with $fencefree(pa_1\restriction_i)$ and $fencefree(pa_1'\restriction_i)$ we get by definition of $\approx_L^{Ob^*}$ that $pa_1{::}[(i, ?@x \rightarrowtriangle r)]\restriction_i \approx_L^{Ob^*} pa_1'{::}[(i, ?@x \rightarrowtriangle r)]\restriction_i$. Hence, $pa_2\restriction_i \approx_L^{Ob^*} pa_2'\restriction_i$.

From $pa_2\restriction_i \approx_L^{Ob^*} pa_2'\restriction_i$, $c_2 = \epsilon$ and $c_2' = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2', pa_2'\restriction_i)$.

Case ($c_1 = \mathbf{store}_\iota\ x\ r$):

In this case, $c_1' = \mathbf{store}_\iota\ x\ r$ follows from $c_1 = c_1'$.

From $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c_1', pa_1', reg_1'\rangle \rightarrow_i \langle c_2', pa_2'\rangle$ we get by the semantics rules of **store** from Figure 7 in the article that $c_2 = \epsilon$, $c_2' = \epsilon$, $pa_2 = pa_1{::}[(i, x \leftarrowtriangle reg_1(r)@r)]$ and $pa_2' = pa_1'{::}[(i, x \leftarrowtriangle reg_1'(r)@r)]$. Hence, $pa_2\restriction_j = pa_1\restriction_j$ and $pa_2'\restriction_j = pa_1'\restriction_j$ for all $j \neq i$.

We distinguish two cases based on the security domain of the variable.

Case ($lev(x) = \mathbf{High}$):

From $lev(x) = \mathbf{High}$ we get by definition of $=_L^{obs}$ that $x \leftarrow reg_1(r)@r =_L^{obs} x \leftarrow reg_1'(r)@r$. From this combined with $pa_1 \restriction_i \approx_L^{Ob^*} pa_1' \restriction_i$, $fencefree(pa_1 \restriction_i)$, $fencefree(pa_1' \restriction_i)$, $pa_2 = pa_1::[(i, x \leftarrow reg_1(r)@r)]$ and $pa_2' = pa_1'::[(i, x \leftarrow reg_1'(r)@r)]$ we get $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$.
From $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$, $c_2 = \epsilon$ and $c_2' = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2', pa_2' \restriction_i)$.

Case ($lev(x) = \mathbf{Low}$):

From $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_1', pa_1' \restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $pc, pt \vdash_{lev} c_1 \diamond (pt')$, $pc, pt \vdash_{lev} c_1' \diamond (pt'')$, $pt \vdash_{lev} pa_1 \restriction_i$ and $pt \vdash_{lev} pa_1' \restriction_i$. From $lev(x) = \mathbf{Low}$ we get by rule [CST] that $lev(r) = \mathbf{Low}$. From this combined with $reg_1 =_L^{\mathcal{R}} reg_1'$ we get $reg_1(r) = reg_1'(r)$. Hence, $x \leftarrow reg_1(r)@r = x \leftarrow reg_1'(r)@r$ and by definition of $=_L^{obs}$ we get $x \leftarrow reg_1(r)@r =_L^{obs} x \leftarrow reg_1'(r)@r$. From this combined with $pa_1 \restriction_i \approx_L^{Ob^*} pa_1' \restriction_i$, $fencefree(pa_1 \restriction_i)$, $fencefree(pa_1' \restriction_i)$, $pa_2 = pa_1::[(i, x \leftarrow reg_1(r)@r)]$ and $pa_2' = pa_1'::[(i, x \leftarrow reg_1'(r)@r)]$ we get $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$.
From $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$, $c_2 = \epsilon$ and $c_2' = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2', pa_2' \restriction_i)$.

Case ($c_1 = \mathbf{eq}_\iota\ r_1\ r_2\ r_3$):

In this case, $c_1' = \mathbf{eq}_\iota\ r_1\ r_2\ r_3$ follows from $c_1 = c_1'$.
From $\langle c_1, pa_1, reg_1 \rangle \rightarrow_i \langle c_2, pa_2 \rangle$ and $\langle c_1', pa_1', reg_1' \rangle \rightarrow_i \langle c_2', pa_2' \rangle$ we get by the semantics rules of **eq** from Figure 7 in the article that $c_2 = \epsilon$, $c_2' = \epsilon$, $pa_2 = pa_1::[(i, v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$ and $pa_2' = pa_1'::[(i, v'@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$. Hence, $pa_2 \restriction_j = pa_1 \restriction_j$ and $pa_2' \restriction_j = pa_1' \restriction_j$ for all $j \neq i$.
We distinguish two cases based on the security domain of the register r_1.

Case ($lev(r_1) = \mathbf{High}$):

From $lev(r_1) = \mathbf{High}$ we get by definition of $=_L^{obs}$ that $v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1 =_L^{obs} v'@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1$. From this combined with $pa_1 \restriction_i \approx_L^{Ob^*} pa_1' \restriction_i$, $fencefree(pa_1 \restriction_i)$, $fencefree(pa_1' \restriction_i)$, $pa_2 = pa_1::[(i, v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$ and $pa_2' = pa_1'::[(i, v'@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$ we get $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$.
From $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$, $c_2 = \epsilon$ and $c_2' = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2', pa_2' \restriction_i)$.

Case ($lev(r_1) = \mathbf{Low}$):

From $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_1', pa_1' \restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $pc, pt \vdash_{lev} c_1 \diamond (pt')$, $pc, pt \vdash_{lev} c_1' \diamond (pt'')$, $pt \vdash_{lev} pa_1 \restriction_i$ and $pt \vdash_{lev} pa_1' \restriction_i$. From $lev(r_1) = \mathbf{Low}$ we get by rule [COP] that $lev(r_2) = lev(r_3) = \mathbf{Low}$. From this combined with $reg_1 =_L^{\mathcal{R}} reg_1'$ we get $reg_1(r_2) = reg_1'(r_2)$ and $reg_1(r_3) = reg_1'(r_3)$ and, thus, $\langle c_1, pa_1, reg_1 \rangle \rightarrow_i \langle c_2, pa_2 \rangle$ and $\langle c_1', pa_1', reg_1' \rangle \rightarrow_i \langle c_2', pa_2' \rangle$ are derived with the same rule from Figure 7 in the article. Hence, $v = v'$. From this we get $v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1 = v'@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1$ and by definition of $=_L^{obs}$ we get $v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1 =_L^{obs} v'@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1$. From this combined with $pa_1 \restriction_i \approx_L^{Ob^*} pa_1' \restriction_i$, $fencefree(pa_1 \restriction_i)$, $fencefree(pa_1' \restriction_i)$, $pa_2 = pa_1::[(i, v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$ and $pa_2' = pa_1'::[(i, v'@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$ we get $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$.
From $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$, $c_2 = \epsilon$ and $c_2' = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2', pa_2' \restriction_i)$.

Case ($c_1 = \mathbf{and}_\iota\ r_1\ r_2\ r_3$):

In this case, $c_1' = \mathbf{and}_\iota\ r_1\ r_2\ r_3$ follows from $c_1 = c_1'$.
From $\langle c_1, pa_1, reg_1 \rangle \rightarrow_i \langle c_2, pa_2 \rangle$ and $\langle c_1', pa_1', reg_1' \rangle \rightarrow_i \langle c_2', pa_2' \rangle$ we get by the semantics rules of **and** from Figure 7 in the article that $c_2 = \epsilon$, $c_2' = \epsilon$, $pa_2 = pa_1::[(i, v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$ and $pa_2' = pa_1'::[(i, v'@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$. Hence, $pa_2 \restriction_j = pa_1 \restriction_j$ and $pa_2' \restriction_j = pa_1' \restriction_j$ for all $j \neq i$.
We distinguish two cases based on the security domain of the register r_1.

Case ($lev(r_1) = \mathbf{High}$):

From $lev(r_1) = \mathbf{High}$ we get by definition of $=_L^{obs}$ that $v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1 =_L^{obs} v'@\mathrm{and}(r_2, r_3) \circlearrowleft r_1$. From this combined with $pa_1 \restriction_i \approx_L^{Ob^*} pa_1' \restriction_i$, $fencefree(pa_1 \restriction_i)$, $fencefree(pa_1' \restriction_i)$, $pa_2 = pa_1::[(i, v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$ and $pa_2' = pa_1'::[(i, v'@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$ we get $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$.
From $pa_2 \restriction_i \approx_L^{Ob^*} pa_2' \restriction_i$, $c_2 = \epsilon$ and $c_2' = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2', pa_2' \restriction_i)$.

Case ($lev(r_1) = \mathbf{Low}$):

From $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_1', pa_1' \restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $pc, pt \vdash_{lev} c_1 \diamond (pt')$, $pc, pt \vdash_{lev} c_1' \diamond (pt'')$, $pt \vdash_{lev} pa_1 \restriction_i$ and $pt \vdash_{lev} pa_1' \restriction_i$. From $lev(r_1) = \mathbf{Low}$ we get by rule [COP] that $lev(r_2) = lev(r_3) = \mathbf{Low}$. From this combined with $reg_1 =_L^{\mathcal{R}} reg_1'$ we get $reg_1(r_2) = reg_1'(r_2)$ and $reg_1(r_3) = reg_1'(r_3)$ and, thus, $\langle c_1, pa_1, reg_1 \rangle \rightarrow_i \langle c_2, pa_2 \rangle$ and $\langle c_1', pa_1', reg_1' \rangle \rightarrow_i \langle c_2', pa_2' \rangle$ are derived with the same rule from Figure 7 in the article. Hence, $v = v'$. From this we get $v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1 = v'@\mathrm{and}(r_2, r_3) \circlearrowleft r_1$ and by definition of $=_L^{obs}$ we get $v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1 =_L^{obs} v'@\mathrm{and}(r_2, r_3) \circlearrowleft r_1$. From this combined with $pa_1 \restriction_i \approx_L^{Ob^*} pa_1' \restriction_i$, $fencefree(pa_1 \restriction_i)$, $fencefree(pa_1' \restriction_i)$,

$pa_2 = pa_1::[(i, v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$ and $pa'_2 = pa'_1::[(i, v'@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$ we get $pa_2 \restriction_i \approx_L^{Ob^*} pa'_2 \restriction_i$.

From $pa_2 \restriction_i \approx_L^{Ob^*} pa'_2 \restriction_i$, $c_2 = \epsilon$ and $c'_2 = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_2, pa'_2 \restriction_i)$.

Case ($c_1 = \mathbf{spawn}_\iota\, c$):

In this case, $c'_1 = \mathbf{spawn}_\iota\, c$ follows from $c_1 = c'_1$.

From $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ and $\langle c'_1, pa'_1, reg'_1\rangle \rightarrow_i \langle c'_2, pa'_2\rangle$ we get by the semantics rules of **spawn** that $c_2 = \epsilon$, $c'_2 = \epsilon$, $pa_2 = pa_1::[(i, \nearrow_c)]$ and $pa'_2 = pa'_1::[(i, \nearrow_c)]$. Hence, $pa_2 \restriction_j = pa_1 \restriction_j$ and $pa'_2 \restriction_j = pa'_1 \restriction_j$ for all $j \neq i$.

From [CSP] we get $pc, pt'' \vdash_{lev} c \diamond (pt''')$ for some $pt'', pt''' \in \{\mathbf{Low}, \mathbf{High}\}$. From the definition of $=_L^{obs}$ we get $\nearrow_c =_L^{obs} \nearrow_c$. From the two previous facts combined with $pa_1 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$, $fencefree(pa_1 \restriction_i)$, $fencefree(pa'_1 \restriction_i)$, $pa_2 = pa_1::[(i, \nearrow_c)]$ and $pa'_2 = pa'_1::[(i, \nearrow_c)]$ we get $pa_2 \restriction_i \approx_L^{Ob^*} pa'_2 \restriction_i$.

From $pa_2 \restriction_i \approx_L^{Ob^*} pa'_2 \restriction_i$, $c_2 = \epsilon$ and $c'_2 = \epsilon$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_2, pa'_2 \restriction_i)$.

As induction hypothesis we assume that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_2, pa'_2 \restriction_i)$, $pa_2 \restriction_i \approx_L^{Ob^*} pa'_2 \restriction_i$, $pa_2 \restriction_j = pa_1 \restriction_j$ and $pa'_2 \restriction_j = pa'_1 \restriction_j$ for all $j \neq i$ holds for derivations with arbitrary length n.

For the induction step let the derivation length be $n' = n + 1$. From the calculus for deriving $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$ we know that only the rules for sequential composition can have a derivation length larger than 1. We distinguish two cases based on the semantics rule for sequential composition used for deriving $\langle c_1, pa_1, reg_1\rangle \rightarrow_i \langle c_2, pa_2\rangle$.

Case (Sequential Composition with $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle \epsilon, pa_3\rangle$):

In this case, $c_1 = c_A;\ c_B$ and $c_1 = c_A;\ c_B$ due to $c_1 = c'_1$. From the assumption of this case we get $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle \epsilon, pa_3\rangle$. From this we get that c_A is an instruction that terminates in one step. The only instruction that may terminate in one or more steps is **while**. However, since **while** is only typeable with a **Low**-register and $reg_1 =_L^{\mathcal{R}} reg'_1$ we know that **while** terminates in one step starting in reg_1 if and only if **while** terminates in one step starting in reg'_1. Hence, from $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle \epsilon, pa_3\rangle$ we obtain $\langle c_A, pa'_1, reg'_1\rangle \rightarrow_i \langle \epsilon, pa'_3\rangle$.

Since $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle \epsilon, pa_3\rangle$ is derived in $n' - 1 = n$ steps we can apply the induction hypothesis to obtain $pa_3 \restriction_i \approx_L^{Ob^*} pa'_3 \restriction_i$, $pa_3 \restriction_j = pa_1 \restriction_j$ and $pa'_3 \restriction_j = pa'_1 \restriction_j$ for all $j \neq i$.

From semantics rule of sequential composition we get $c_2 = c_B$, $c'_2 = c_B$, $pa_2 = pa_3$ and $pa'_2 = pa'_3$. From this combined with $pa_3 \restriction_i \approx_L^{Ob^*} pa'_3 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_2, pa'_2 \restriction_i)$.

Case (Sequential Composition with $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle c_C, pa_3\rangle$):

In this case, $c_1 = c_A;\ c_B$ and $c_1 = c_A;\ c_B$ due to $c_1 = c'_1$. From the assumption of this case we get $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle c_C, pa_3\rangle$. From this we get that c_A is an instruction that does not terminate in one step. The only instruction that may terminate in one or more steps is **while**. However, since **while** is only typeable with a **Low**-register and $reg_1 =_L^{\mathcal{R}} reg'_1$ we know that **while** terminates in more thatn one step starting in reg_1 if and only if **while** terminates in one step starting in reg'_1. Hence, from $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle c_C, pa_3\rangle$ we obtain $\langle c_A, pa'_1, reg'_1\rangle \rightarrow_i \langle c'_C, pa'_3\rangle$.

Since $\langle c_A, pa_1, reg_1\rangle \rightarrow_i \langle \epsilon, pa_3\rangle$ is derived in $n' - 1 = n$ steps we can apply the induction hypothesis to obtain $(c_C, pa_3) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_C, pa_3)$, $pa_3 \restriction_i \approx_L^{Ob^*} pa'_3 \restriction_i$, $pa_3 \restriction_j = pa_1 \restriction_j$ and $pa'_3 \restriction_j = pa'_1 \restriction_j$ for all $j \neq i$.

From $(c_C, pa_3) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_C, pa_3)$ we get by the third condition of the definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_C;\ c_B, pa_3) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_C;\ c_B, pa_3)$.

From semantics rule of sequential composition we get $c_2 = c_C;\ c_B$, $c'_2 = c'_C;\ c_B$, $pa_2 = pa_3$ and $pa'_2 = pa'_3$. From this combined with $(c_C;\ c_B, pa_3) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_C;\ c_B, pa_3)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(c_2, pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_2, pa'_2 \restriction_i)$.

■

The following lemma shows that an execution step of a **High**-typable instruction results in a **High**-typed obligation list, i.e. such an execution step does not add an obligation that causes the obligation list to become **Low**.

Lemma 33. *If* $\langle c_A;\ c_B, pa_1, reg\rangle \rightarrow_i \langle c_C, pa_2\rangle$, $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$, $fencefree(pa_1 \restriction_i)$ *and* $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ *with* $c_C \in \mathcal{C}$, *then* $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$.

Proof: From $\langle c_A;\ c_B, pa_1, reg\rangle \rightarrow_i \langle c_C, pa_2\rangle$ we know by the rules for sequential composition that $\langle c_A, pa_1, reg\rangle \rightarrow_i \langle c'_A, pa_2\rangle$ is derivable for some $c'_A \in (\mathcal{C} \cup \{\epsilon\})$.

We prove this lemma by an induction on the derivation length of the judgment $\langle c_A, pa_1, reg\rangle \rightarrow_i \langle c'_A, pa_2\rangle$. The induction base are derivations with a length of 1. Derivations with length 1 are possible for all instructions that are not sequentially composed.

Let $reg_1 \in Reg$, $pa_1, pa_2 \in Pa$, $c_A, c_B, c_C \in \mathcal{C}$ be arbitrary with $\langle c_A;\ c_B, pa_1, reg\rangle \rightarrow_i \langle c_C, pa_2\rangle$ with $c_C \in \mathcal{C}$, $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$, $fencefree(pa_1\restriction_i)$ and $\mathbf{High} \vdash_{lev} pa_1\restriction_i$.

We make a case distinction on the shapes of c_A for which a derivation in one step is possible, i.e. all instructions except sequential composition.

Case ($c_A = \mathbf{skip}_\iota$):
In this case we get from rule of **skip** that $pa_2 = pa_1$. Hence, $\mathbf{High} \vdash_{lev} pa_2\restriction_i$ follows directly from $\mathbf{High} \vdash_{lev} pa_1\restriction_i$.

Case ($c_A = \mathbf{while}_\iota\ r\ \mathbf{do}\ c'\ \mathbf{od}$):
This case is not applicable, because rule [CWH] requires that $pt = \mathbf{Low}$, i.e. $\mathbf{Low}, pt \vdash_{lev} c_A \diamond (\mathbf{Low})$.

Case ($c_A = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_t\ \mathbf{else}\ c_e\ \mathbf{fi}$):
In this case we get from the rules of **if** that $pa_2 = pa_1$. Hence, $\mathbf{High} \vdash_{lev} pa_2\restriction_i$ follows directly from $\mathbf{High} \vdash_{lev} pa_1\restriction_i$.

Case ($c_A = \mathbf{load}_\iota\ r\ v$):
In this case we get from the rule of **load** with a constant that $pa_2 = pa_1{::}[(i, v@\mathrm{const} \circlearrowleft r)]$.
From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [CLC] that $lev(r) = \mathbf{High}$. Hence, from rule [PC] we get $\mathbf{High} \vdash_{lev} pa_2\restriction_i$.

Case ($c_1 = \mathbf{load}_\iota\ r\ x$):
In this case we get from the rule of **load** with a variable that $pa_2 = pa_1{::}[(i, ?@x \twoheadrightarrow r)]$.
From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [CLX] that $lev(r) = \mathbf{High}$. Hence, from rule [PL] we get $\mathbf{High} \vdash_{lev} pa_2\restriction_i$.

Case ($c_1 = \mathbf{store}_\iota\ x\ r$):
In this case we get from the rule of **store** that $pa_2 = pa_1{::}[(i, x \twoheadleftarrow reg_1(r)@r)]$.
From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [CST] that $lev(x) = \mathbf{High}$. Hence, from rule [PS] we get $\mathbf{High} \vdash_{lev} pa_2\restriction_i$.

Case ($c_1 = \mathbf{eq}_\iota\ r_1\ r_2\ r_3$):
In this case we get from the rules of **eq** that $pa_2 = pa_1{::}[(i, v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$.
From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [COP] that $lev(r_1) = \mathbf{High}$. Hence, from rule [PV] we get $\mathbf{High} \vdash_{lev} pa_2\restriction_i$.

Case ($c_1 = \mathbf{and}_\iota\ r_1\ r_2\ r_3$):
In this case we get from the rule of **and** that $pa_2 = pa_1{::}[(i, v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$.
From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [COP] that $lev(r_1) = \mathbf{High}$. Hence, from rule [PV] we get $\mathbf{High} \vdash_{lev} pa_2\restriction_i$.

Case ($c_1 = \mathbf{spawn}_\iota c$):
This case is not applicable, because rule [CSP] requires that $pt = \mathbf{Low}$, i.e. $\mathbf{Low}, pt \vdash_{lev} c_A \diamond (\mathbf{Low})$.

As induction hypothesis we assume that $\mathbf{High} \vdash_{lev} pa_2\restriction_i$ holds for derivations with arbitrary length n.

For the induction step let the derivation length be $n' = n + 1$. From the calculus for deriving $\langle c_A, pa_1, \rangle \rightarrow_i \langle c'_A, pa_2\rangle$ we know that only the rules for sequential composition can have a derivation length larger than 1.

From the semantics rules of sequential composition we know that $\langle c_D, pa_1, reg\rangle \rightarrow_i \langle c'_D, pa_2\rangle$ is derivable for some $c_D \in \mathcal{C}$ and $c'_D \in (\mathcal{C} \cup \{\epsilon\})$ with $c_A = c_D;\ c_E$ for some c_E. From this combined with $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [CSQ] that $\mathbf{High}, pt \vdash_{lev} c_D \diamond (pt'')$. Hence, we can apply the induction hypothesis to obtain $\mathbf{High} \vdash_{lev} pa_2\restriction_i$, because the derivation length of $\langle c_D, pa_1, reg\rangle \rightarrow_i \langle c'_D, pa_2\rangle$ is $n' - 1 = n$. ■

The following lemma shows that a step of a **High**-typeable instruction with a **High**-typeable list of obligations results in an instruction and obligation list that is **Low**-similar to the original instruction and obligation list.

Lemma 34 (Confinement of Thread Step)**.** *If* $\langle c_1, pa_1, reg\rangle \rightarrow_i \langle c_2, pa_2\rangle$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$ *and* $\mathbf{High} \vdash_{lev} pa_1\restriction_i$, *then* $(c_1, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c_2, pa_2\restriction_i)$.

Proof: Let $c_1 \in \mathcal{C}$, $pa_1, pa_2 \in Pa$, $reg \in Reg$ and $c_2 \in (\mathcal{C} \cup \{\epsilon\})$ be arbitrary such that $\langle c_1, pa_1, reg\rangle \rightarrow_i \langle c_2, pa_2\rangle$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$ and $\mathbf{High} \vdash_{lev} pa_1\restriction_i$.

We prove this lemma by an induction on the derivation length of $\langle c_1, pa_1, reg\rangle \rightarrow_i \langle c_2, pa_2\rangle$. The induction base are derivations with a length of 1. Derivations with a length of 1 are possible with all instructions that are not sequentially composed. We distinguish cases based on the shape of c_1 for which the derivation length is 1.

Case ($c_1 = \mathbf{skip}_\iota$):
In this case we get from rule of **skip** that $pa_2 = pa_1$ and $c_2 = \epsilon$. Hence, $\mathbf{High} \vdash_{lev} pa_2\restriction_i$ follows directly from $\mathbf{High} \vdash_{lev} pa_1\restriction_i$.
From $c_2 = \epsilon$ we get by rule [EM] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$.

Case ($c_A = \mathbf{while}_\iota\ r\ \mathbf{do}\ c'\ \mathbf{od}$):

This case is not applicable, because rule [CWH] requires that $pt = \mathbf{Low}$, i.e. $\mathbf{Low}, pt \vdash_{lev} c_A \diamond (\mathbf{Low})$.

Case ($c_A = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_t\ \mathbf{else}\ c_e\ \mathbf{fi}$):

In this case we get from the rules of **if** that $pa_2 = pa_1$ and $c_2 = c$ for some $c \in \{c_t, c_e\}$. Hence, $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ follows directly from $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$.

From the rule [CIL] and [CIH] we get $\mathbf{High}, \mathbf{High} \vdash_{lev} c \diamond (\mathbf{High})$ for all $c \in \{c_t, c_e\}$. From $c_2 = c$ for some $c \in \{c_t, c_e\}$ we get $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$.

Case ($c_A = \mathbf{load}_\iota\ r\ v$):

In this case we get from the rule of **load** with a constant that $pa_2 = pa_1 :: [(i, v@\mathrm{const} \circlearrowleft r)]$ and $c_2 = \epsilon$.

From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [CLC] that $lev(r) = \mathbf{High}$. Hence, from rule [PC] we get $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$.

From $c_2 = \epsilon$ we get by rule [EM] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$.

Case ($c_1 = \mathbf{load}_\iota\ r\ x$):

In this case we get from the rule of **load** with a variable that $pa_2 = pa_1 :: [(i, ?@x \rightarrow r)]$ and $c_2 = \epsilon$.

From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [CLX] that $lev(r) = \mathbf{High}$. Hence, from rule [PL] we get $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$.

From $c_2 = \epsilon$ we get by rule [EM] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$.

Case ($c_1 = \mathbf{store}_\iota\ x\ r$):

In this case we get from the rule of **store** that $pa_2 = pa_1 :: [(i, x \leftarrow reg_1(r)@r)]$ and $c_2 = \epsilon$.

From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [CST] that $lev(x) = \mathbf{High}$. Hence, from rule [PS] we get $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$.

From $c_2 = \epsilon$ we get by rule [EM] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$.

Case ($c_1 = \mathbf{eq}_\iota\ r_1\ r_2\ r_3$):

In this case we get from the rules of **eq** that $pa_2 = pa_1 :: [(i, v@\mathrm{eq}(r_2, r_3) \circlearrowleft r_1)]$ and $c_2 = \epsilon$.

From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [COP] that $lev(r_1) = \mathbf{High}$. Hence, from rule [PV] we get $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$.

From $c_2 = \epsilon$ we get by rule [EM] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$.

Case ($c_1 = \mathbf{and}_\iota\ r_1\ r_2\ r_3$):

In this case we get from the rule of **and** that $pa_2 = pa_1 :: [(i, v@\mathrm{and}(r_2, r_3) \circlearrowleft r_1)]$ and $c_2 = \epsilon$.

From $\mathbf{High}, pt \vdash_{lev} c_A \diamond (pt')$ we get by rule [COP] that $lev(r_1) = \mathbf{High}$. Hence, from rule [PV] we get $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$.

From $c_2 = \epsilon$ we get by rule [EM] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$.

Case ($c_1 = \mathbf{spawn}_\iota c$):

This case is not applicable, because rule [CSP] requires that $pt = \mathbf{Low}$, i.e. $\mathbf{Low}, pt \vdash_{lev} c_A \diamond (\mathbf{Low})$.

As induction hypothesis we assume that $(c_A, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_C, pa_2 \restriction_i)$ holds for derivations with arbitrary length n.

For the induction step let the derivation length be $n' = n + 1$. From the calculus for deriving $\langle c_1, pa_1, \rangle \rightarrow_i \langle c_2, pa_2 \rangle$ we know that only the rules for sequential composition can have a derivation length larger than 1.

From the semantics rules we get $pa_2 = pa_1 :: pa_3$ for some pa_3 with $|pa_3| \leq 1$ and $|pa_3| = |pa_3 \restriction_i|$.

Hence, $c_1 = c_A;\ c_B$. From this combined with $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$ we get by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (pt)$ and $\mathbf{High}, pt \vdash_{lev} c_B \diamond (\mathbf{High})$. From the calculus for typing we obtain that $pt = \mathbf{High}$.

From $c_1 = c_A;\ c_B$ and $\langle c_1, pa_1, \rangle \rightarrow_i \langle c_2, pa_2 \rangle$ we get by semantics of sequential composistion we that $(c_A, pa_1 \restriction_i) \rightarrow_i (c_C, pa_2 \restriction_i)$ is derivable for some $c_C \in (\mathcal{C} \cup \{\epsilon\})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$ we get by Lemma 24 that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (\mathbf{High})$. If $c_c \neq \epsilon$ we get from

$\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_B \diamond (\mathbf{High})$ by rule [CSQ] that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C;\ c_B \diamond (\mathbf{High})$.

From semantics of sequential composition we get $c_2 = c_B$ or $c_2 = c_C;\ c_B$. Hence, we get either from $\mathbf{High}, \mathbf{High} \vdash_{lev} c_B \diamond (\mathbf{High})$ or $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C;\ c_B \diamond (\mathbf{High})$ that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$, and $\mathit{fencefree}(pa_1 \restriction_i)$ we get by Lemma 33 that $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$. From $pa_2 = pa_1 :: pa_3$ for some pa_3 with $|pa_3| \leq 1$ and $|pa_3| = |pa_3 \restriction_i|$, $\mathit{fencefree}(pa_1)$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{Ob^*}$ that $pa_1 \restriction_i \approx_L^{Ob^*} pa_2 \restriction_i$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(c_1, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_2, pa_2 \restriction_i)$. ■

The following lemma shows that a step in a **High**-typeable prefix of an instruction with a **High**-typeable obligation list results in a **Low**-similar configuration.

Lemma 35 (Confinement of Next Step of a Thread)**.** *If* $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$, $\vec{cs}_1(i) = c_A;\ c_B$, $\vec{cs}_2(i) = c_C;\ c_B$ *with* $c_C \in \mathcal{C}$, $\langle c_A;\ c_B, pa_1, \vec{reg}(i) \rangle \rightarrow_i \langle c_C;\ c_B, pa_2 \rangle$, $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ *and* $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$*, then* $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$ *and* $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ *hold.*

Proof: Let $\vec{cs}_1, \vec{cs}_2 : \mathcal{I} \rightharpoonup (\mathcal{C} \cup \{\epsilon\})$, $pa_1, pa_2 \in Pa$, $\vec{tr}_1, \vec{tr}_2 \in \vec{Tr}$, $gst_1, gst_2 \in Gst$, $c_A, c_B, c_C \in \mathcal{C}$ and $i \in \mathcal{I}$ be arbitrary with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1) \rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$, $\vec{cs}_1(i) = c_A;\ c_B$, $\vec{cs}_2(i) = c_C;\ c_B$, $\langle c_A;\ c_B, pa_1, \vec{reg}(i) \rangle \rightarrow_i \langle c_C;\ c_B, pa_2 \rangle$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$.

From the rule for thread progress we get $gst_2 = gst_1$, $pre(\vec{cs}_2) = pre(\vec{cs}_1)$, $\vec{cs}_2(j) = \vec{cs}_1(j)$, $pa_2 \restriction_j = pa_1 \restriction_j$ for all $j \neq i$ and $\mathit{fencefree}(pa_1 \restriction_i)$.

From $\langle c_A;\ c_B, pa_1, \vec{reg}(i) \rangle \rightarrow_i \langle c_C;\ c_B, pa_2 \rangle$ we know by the rules for sequential composition that $\langle c_A, pa_1, reg \rangle \rightarrow_i \langle c_C, pa_2 \rangle$.

From $\langle c_A, pa_1, reg \rangle \rightarrow_i \langle c_C, pa_2 \rangle$, $\mathit{fencefree}(pa_1 \restriction_i)$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$ we get by Lemma 24 that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (\mathbf{High})$ and by Lemma 33 that $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ and by Lemma 34 that $(c_A, pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (c_C, pa_2 \restriction_i)$ and, hence, $pa_1 \restriction_i \approx_L^{Ob^*} pa_2 \restriction_i$.

From $\vec{cs}_1(i) = c_A;\ c_B$, $\vec{cs}_2(i) = c_C;\ c_B$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_C \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$, $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$ and $pa_1 \restriction_i \approx_L^{Ob^*} pa_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*}$ that $(\vec{cs}_1(i), pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (\vec{cs}_2(i), pa_2 \restriction_i)$. From this combined with $gst_2 = gst_1$, $pre(\vec{cs}_2) = pre(\vec{cs}_1)$, $\vec{cs}_2(j) = \vec{cs}_1(j)$, $pa_2 \restriction_j = pa_1 \restriction_j$ for all $j \neq i$ we get by definition of $\approx_L^{Conf}$ that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$. ■

The following lemma shows that an execution step of an instruction that is typeable as **High** results in an instruction and list of obligations that are **Low**-similar to the original instruction and list of obligations.

Lemma 36 (Confinement of Threads)**.** *If* $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1) \rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$ *and* $\langle \vec{cs}_1(i), pa_1, \vec{reg}(i) \rangle \rightarrow_i \langle \vec{cs}_2(i), pa_2 \rangle$ *and* $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ *and* $\mathbf{High}, \mathbf{High} \vdash_{lev} \vec{cs}_1(i) \diamond (\mathbf{High})$*,*
then $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1) \rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$.

Proof: Let $\vec{cs}_1, \vec{cs}_2 : \mathcal{I} \rightharpoonup (\mathcal{C} \cup \{\epsilon\})$, $pa_1, pa_2 \in Pa$, $\vec{tr}_1, \vec{tr}_2 \in \vec{Tr}$, $gst_1, gst_2 \in Gst$ and $i \in \mathcal{I}$ be arbitrary with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1) \rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$, $\langle \vec{cs}_1(i), pa_1, \vec{reg}(i) \rangle \rightarrow_i \langle \vec{cs}_2(i), pa_2 \rangle$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} \vec{cs}_1(i) \diamond (\mathbf{High})$.

From the rule for thread progress we get $gst_2 = gst_1$, $pre(\vec{cs}_2) = pre(\vec{cs}_1)$, $\vec{cs}_2(j) = \vec{cs}_1(j)$, $pa_2 \restriction_j = pa_1 \restriction_j$ for all $j \neq i$ and $\mathit{fencefree}(pa_1 \restriction_i)$.

From $\mathbf{High}, \mathbf{High} \vdash_{lev} \vec{cs}_1(i) \diamond (\mathbf{High})$, $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$, $\mathit{fencefree}(pa_1 \restriction_i)$ and $\langle \vec{cs}_1(i), pa_1, \vec{reg}(i) \rangle \rightarrow_i \langle \vec{cs}_2(i), pa_2 \rangle$ we get by Lemma 34 that $(\vec{cs}_1(i), pa_1 \restriction_i) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (\vec{cs}_2(i), pa_2 \restriction_i)$.

From this combined with $gst_2 = gst_1$, $pre(\vec{cs}_2) = pre(\vec{cs}_1)$, $\vec{cs}_2(j) = \vec{cs}_1(j)$ and $pa_2 \restriction_j = pa_1 \restriction_j$ for all $j \neq i$ we get by definition of $\approx_L^{Conf}$ that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1) \rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$. ■

The following lemma shows that fulfilling an obligation of a thread with a list of obligations that is typeable as **High** results in configuration that is **Low**-similar to the original configuration.

Lemma 37 (Confinement of Obligation Fulfilling)**.** *If* $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$ *and* $pa_2 = pa_1 \setminus m$ *and* $pa_1[m] = (i, ob)$ *and* $\mathbf{High} \vdash_{lev} pa_1[0 \ldots k] \restriction_i$ *for all* $k < m$*, then* $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$ *and* $\vec{cs}_2 = \vec{cs}_1$.

Proof: Let $\vec{cs}_1, \vec{cs}_2 : \mathcal{I} \rightharpoonup (\mathcal{C} \cup \{\epsilon\})$, $pa_1, pa_2 \in Pa$, $\vec{tr}_1, \vec{tr}_2 \in \vec{Tr}$, $gst_1, gst_2 \in Gst$ and $i \in \mathcal{I}$ be arbitrary with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle$ and $pa_2 = pa_1 \setminus m$ and $pa[m] = (i, ob)$ and $\mathbf{High} \vdash_{lev} pa_1[0 \ldots k] \restriction_i$ for all $k < m$.

From $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$ and $pa_2 = pa_1 \setminus m$ we get that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$ must be derived with the second, third or fourth rule of Figure 5 in the article. Hence, $next_\Phi(pa_1, m)$ must hold. Since the program-order relaxations in Figure 1, 2 and 3 of the article do not allow reordering of an obligation $ob' \in Fe$ we get $fencefree(pa_1[0 \dots m]\restriction_i)$.

From $\mathbf{High} \vdash_{lev} pa_1[0 \dots k]\restriction_i$ for all $k < m$ and $fencefree(pa_1[0 \dots m-1]\restriction_i [])$ we get by definition of $\approx_L^{Ob^*}$ that $pa_1\restriction_i \approx_L^{Ob^*} pa_1 \setminus m\restriction_i$. From this combined with $pa_2 = pa_1 \setminus m$ we get $pa_1\restriction_i \approx_L^{Ob^*} pa_2\restriction_i$.

From $\mathbf{High} \vdash_{lev} pa_1[0 \dots k]\restriction_i$ for all $k < m$ we get that $ob \notin \{\nearrow_c |\ c \in \mathcal{C}\}$, because the only typing rule for these obligations is [PT] and this rule is only applicable if $pt = \mathbf{Low}$. Hence, only the second and third rule of Figure 5 in the article are applicable.

From the second and third rule in Figure 5 of the article we get $\vec{cs}_1 = \vec{cs}_2$ and either $gst_1 = gst_2$ or $gst_2 = gst_1[i \mapsto (effect(ob', gst_1[i]))]$. From the definition of *effect* we get that the only cases where $gst_1 \neq gst_2$ holds are the cases where ob is of one of the following shapes $v'@x \twoheadrightarrow r$ or $x \twoheadleftarrow v@r$ or $e \circlearrowright r$. In these cases we get from $\mathbf{High} \vdash_{lev} pa_1[0 \dots k]\restriction_i$ for all $k < m$ by the rules [PS], [PL], [PC] and [PV] that the domain assignment assigns $\mathbf{High}$ to the target variable or register. Hence, $gst_1 =_L^{Gst} gst_2$.

From $\vec{cs}_1 = \vec{cs}_2$, $pa_1\restriction_i \approx_L^{Ob^*} pa_2\restriction_i$, $gst_1 =_L^{Gst} gst_2$ we get by definition of $\approx_L^{Conf}$ that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$. ■

The following lemma shows that an execution step from a configuration can be matched by a (possibly empty) sequence of execution steps from a $\mathbf{Low}$-similar configuration such that the resulting configurations are $\mathbf{Low}$-similar again.

Lemma 38 (One Step Security)**.** *If* $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ *holds and* $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$ *is derivable, then there are* $\vec{cs}'_2 : \mathcal{I} \rightharpoonup (\mathcal{C} \cup \{\epsilon\})$, $pa'_2 \in Pa$, $\vec{tr}'_2 \in \vec{Tr}$, $gst'_2 \in Gst$ *such that* $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \Longrightarrow^*_{MM} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$ *and* $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Proof: Let $MM \in \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$ $\vec{cs}_1, \vec{cs}'_1, \vec{cs}_2 : \mathcal{I} \rightharpoonup (\mathcal{I} \cup \{\epsilon\})$, $pa_1, pa'_1, pa_2 \in Pa$, $\vec{tr}_1, \vec{tr}'_1, \vec{tr}_2 \in \vec{Tr}$, $gst_1, gst'_1, gst_2 \in Gst$ be arbitrary such that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ holds and the judgment $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$ is derivable.

The execution step $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$ is either caused by progress of a thread i or by fulfilling an obligation of a thread i with $i \in pre(\vec{cs}_1)$ for some $i \in pre(\vec{cs}_1)$. We fix $i \in pre(\vec{cs}_1)$ and distinguish two cases based on whether the thread i progresses or an obligation of the thread i is fulfilled.

Case (obligation fulfilled):

In this case we know that the judgment was derived with either the second or the third or the fourth rule from Figure 5 depending on the obligation that was fulfilled.

From $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get by definition of $\approx_L^{Conf}$ and of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $pa_1\restriction_i \approx_L^{Ob^*} pa'_1\restriction_i$ holds.

From $pa_1\restriction_i \approx_L^{Ob^*} pa'_1\restriction_i$ we get by definition of $\approx_L^{Ob^*}$ that there is $obs_A, obs'_A, obs_B, obs'_B \in Ob^*$ with $pa_1\restriction_i = obs_A :: obs_B$, $pa'_1\restriction_i = obs'_A :: obs'_B$, $\mathbf{High} \vdash_{lev} obs_A$, $fencefree(obs_A)$, $\mathbf{High} \vdash_{lev} obs'_A$, $fencefree(obs'_A)$, $pt \vdash_{lev} obs_B$, $pt \vdash_{lev} obs'_B$ for some $pt \in \{\mathbf{Low}, \mathbf{High}\}$.

We distinguish two cases based on whether the obligation that gets fulfilled is an obligation from obs_A or obs_B.

Case (obligation from obs_A gets fulfilled):

In this case there is a $m < |pa_1|$ such that $pa_2 = pa_1 \setminus m$ and a $m' < |obs_A|$ such that $obs_A[m']$ is the obligation of $pa_1[m]$.

From $\mathbf{High} \vdash_{lev} obs_A$ and $fencefree(obs_A)$ we get that $\mathbf{High} \vdash_{lev} obs_A[0 \dots k]$ for all $k < |obs_A|$. Since $m' < |obs_A|$ we also get $\mathbf{High} \vdash_{lev} obs_A[0 \dots k]$ for all $k < m'$.

From $\mathbf{High} \vdash_{lev} obs_A[0 \dots k]$ for all $k < m'$ we get by Lemma 37 that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$. Hence, by symmetry and transitivity of $\approx_L^{Conf}$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$.

Case (obligation from obs_B gets fulfilled):

In this case there is a $m < |pa_1|$ such that $pa_2 = pa_1 \setminus m$ and a $m' < |obs_B|$ such that $obs_B[m']$ is the obligation of $pa_1[m]$.

We distinguish three cases based on the disjunct of the fifth condition in Definition 5 that is satisfied.

Case ($obs_B = [] \wedge obs'_B = [|]$):

This case is not applicable, because according to the assumption of this case an obligation from obs_B gets fulfilled.

Case ($obs_B = [|] \wedge obs'_B = []$):

In this case, the third rule of Figure 5 from the article is the only applicable rule to derive the judgment $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$. From this rule we get $\vec{cs}_2 = \vec{cs}_1$ and $gst_2 = gst_1$ and $pa_2 = pa_1 \setminus m$ with $pa_2 \restriction_i = obs_A::[]$.
From $pa_1 \restriction_i = obs_A::obs_B$, $pa'_1 \restriction_i = obs'_A::obs'_B$, $pa_2 \restriction_i = obs_A::[]$, $obs'_B = []$ and $pa_1 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$ we get by definition of $\approx_L^{Ob^*}$ that $pa_2 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$.
Since $pa_2 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$, $pa_2 = pa_1 \setminus m$ with $pa_1[m] = (i, ob)$, $\vec{cs}_1 = \vec{cs}_2$, $gst_1 = gst_2$ and $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$.

Case ($|obs_B| = |obs'_B| \wedge \forall k < (|obs_B|).obs_B[k] =_L^{obs} obs'_B[k]$):
In this case, each of the last three rules in Figure 5 of the article is applicable to derive the judgment $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$.
In a first step, we reduce the second configuration to a configuration where the **High**-typed prefix ob'_A of the obligation list of thread i is empty.
From $\mathbf{High} \vdash_{lev} obs'_A$ and $fencefree(obs'_A)$ we get that $\mathbf{High} \vdash_{lev} obs'_A[0 \ldots k]$ for all $k < |obs'_A|$. Thus we can apply Lemma 37 and transitivity of $\approx_L^{Conf}$ $|obs'_A|$ times to reach a configuration $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ with $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \Longrightarrow_{MM}^* \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ and $path'_3 \restriction_i = obs'_B$.
In the second step, we show that $next_\Phi(pa'_3, m'')$ holds where m' is the index of $obs'_B[m']$ in path pa'_3.
From the three rules for fulfilling obligations from Figure 5 in the article we know that $next_\Phi(pa_1, m)$ holds. Hence, we get by definition of $next$ and $\bar{\Phi}$ that there is $\phi \in \Phi$ such that $\phi(obs_A::obs_B[0 \ldots (|obs_A| - 1 + m')], k)$ holds for each $k < |obs_A::obs_B[0 \ldots (|obs_A| - 1 + m')]|$, because $pa_1 \restriction_i = obs_A::obs_B$. Thus, there is $\phi \in \Phi$ such that $\phi(obs_B[0 \ldots (m')], k)$.
From the assumption of this case we get by the definition of $\approx_L^{Ob^*}$ that $sources(obs_B[k]) = sources(obs'_B[k])$, $sinks(ob_B[k]) = sinks(obs'_B[k])$ and the obligations at index k agree on their type for all $k < |obs_B|$. Hence, from the existence of $\phi \in \Phi$ such that $\phi(obs_B[0 \ldots (m')], k)$ we get that $\phi(obs'_B[0 \ldots (m')], k)$.
Since $pa'_3 \restriction_i = obs'_B$ we get by definition of $\bar{\Phi}$ and $next$ that $next_\Phi(pa'_3, m'')$ holds where m'' is the index of the obligation $obs'_B[m']$ in path pa'_3 and the obligations in the pairs $pa_1[m]$ and $pa'_3[m'']$ are **Low**-equal due to the assumption of this case. Since $next_\Phi(pa'_3, m'')$ holds we also get that $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle \Longrightarrow_{MM} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$ is derivable for some $\langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$ with $pa'_2 = pa'_3 \setminus m''$.
In a third step we show that after fulfilling the obligations $pa_1[m]$ and $pa'_3[m'']$ the resulting configurations are **Low**-similar again, i.e. $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.
Since $obs_B[k] =_L^{obs} obs'_B[k]$ for all $k < |obs_B|$ we get $obs_B \setminus m'[k] =_L^{obs} obs'_B \setminus m'[k]$ for all $k < |obs_B \setminus m'|$. Hence, $obs_A::(obs_B \setminus m') \approx_L^{Ob^*} ::(obs'_B \setminus m')$ where m' is the index of the obligations $pa_1[m]$ and $pa'_3[m'']$ in the paths. Thus, from $pa_2 = pa_1 \setminus m$, $pa'_2 = pa'_3 \setminus m''$ and $pa_1 \restriction_j \approx_L^{Ob^*} pa'_3 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ we get that $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$.
Let ob_1 and ob'_3 be the obligations with $pa_1[m] = (i, ob_1)$ and $pa'_3[m''] = (i, ob'_3)$. We distinguish seven cases based on the shape of the obligation ob_1.

Case ($ob_1 = \|$):
In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob'_3 = \|$. Hence, the only applicable rule for fulfilling ob_1 and ob'_3 is the third rule in Figure 5 of the article.
From this rule we get $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $gst_2 = gst_1$, $gst'_2 = gst'_3$, $pa_2 = pa_1 \setminus m$ and $pa'_2 = pa'_3 \setminus m''$.
Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ and $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case ($ob_1 = v@x \rightarrowtail r$ for some $v \in \mathcal{V}$):
In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob'_3 = v'@x \rightarrowtail r$ for some $v' \in \mathcal{V} \cup \{?\}$. Hence, the only applicable rule for fulfilling ob_1 and ob'_3 is the second rule in Figure 5 of the article.
From definition of $specialize_\Psi$ we get $specialize_\Psi(pa_1[0 \ldots (m-1)], i, ob_1, gst_1) = ob_1$ and $specialize_\Psi(pa'_3[0 \ldots (m''-1)], i, ob'_3, gst'_3) = ob''_3$.
From the second rule in Figure 5 in the article we get $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $gst_2 = gst_1[i \mapsto (effect(ob_1, gst_1[i]))]$ and $gst'_2 = gst'_3[i \mapsto (effect(ob''_3, gst'_3[i]))]$. Hence, from definition of $effect$ we get $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_2 = gst'_3[i \mapsto (reg'_3[r \mapsto v''], mem'_3)]$ for some $v'' \in \mathcal{V}$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i) = reg'_3$.
We distinguish two cases based on whether $lev(r) = \mathbf{High}$ or $lev(r) = \mathbf{Low}$.

Case ($lev(r) = \mathbf{High}$):

In this case $gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst_1$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_3[i \mapsto (reg'_3[r \mapsto v''], mem'_3)] =_L^{Gst} gst'_3$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i) = reg'_3$ holds, according to the definition of $=_L^{Gst}$. Hence, $gst_2 =_L^{Gst} gst_1$ and $gst'_2 =_L^{Gst} gst'_3$. From this we get by transitivity and symmetry of $=_L^{Gst}$ that $gst_2 =_L^{Gst} gst'_2$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst'_2$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case ($lev(r) = \mathbf{Low}$):

In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob_1 = ob'_3$ and, hence, $ob''_3 = ob'_3$. Thus we get $v = v''$ and in consequence $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_2 = gst'_3[i \mapsto (reg'_3[r \mapsto v], mem'_3)]$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i)$. From this combined with $gst_1 =_L^{Gst} gst'_3$ we get by transitivity and symmmetry of $=_L^{Gst}$ $gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst'_3[i \mapsto (reg'_3[r \mapsto v], mem'_3)]$. Hence, $gst_2 =_L^{Gst} gst'_2$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst'_2$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case ($ob_1 = ?@x \rightarrowtriangle r$):

In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob'_3 = v'@x \rightarrowtriangle r$ for some $v' \in \mathcal{V} \cup \{?\}$. Hence, the only applicable rule for fulfilling ob_1 and ob'_3 is the second rule in Figure 5 of the article.

From definition of $specialize_\Psi$ we get $specialize_\Psi(pa_1[0 \ldots (m-1)], i, ob_1, gst_1) = ob'_1 = v@x \rightarrowtriangle r$ for some $v \in \mathcal{V}$ and $specialize_\Psi(pa'_3[0 \ldots (m''-1)], i, ob'_3, gst'_3) = ob''_3 = v''@x \rightarrowtriangle r$ for some $v'' \in \mathcal{V}$.

From the second rule in Figure 5 in the article we get $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $gst_2 = gst_1[i \mapsto (\mathit{effect}(ob'_1, gst_1[i]))]$ and $gst'_2 = gst'_3[i \mapsto (\mathit{effect}(ob''_3, gst'_3[i]))]$. Hence, from definition of *effect* we get $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_2 = gst'_3[i \mapsto (reg'_3[r \mapsto v''], mem'_3)]$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i) = reg'_3$.

We distinguish two cases based on whether $lev(r) = \mathbf{High}$ or $lev(r) = \mathbf{Low}$.

Case ($lev(r) = \mathbf{High}$):

In this case $gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst_1$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_3[i \mapsto (reg'_3[r \mapsto v''], mem'_3)] =_L^{Gst} gst'_3$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i) = reg'_3$ holds, according to the definition of $=_L^{Gst}$. Hence, $gst_2 =_L^{Gst} gst_1$ and $gst'_2 =_L^{Gst} gst'_3$. From this we get by transitivity and symmetry of $=_L^{Gst}$ that $gst_2 =_L^{Gst} gst'_2$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst'_2$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case ($lev(r) = \mathbf{Low}$):

In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob_1 = ob'_3$ and, hence, $ob'_3 = ?@x \rightarrowtriangle r$.

From $lev(r) = \mathbf{Low}$ we get by rule [PL] that $lev(x) \sqsubseteq lev(r)$ and, hence, $lev(x) = \mathbf{Low}$.

From $lev(x) = \mathbf{Low}$ we get by rule [PS] that any obligation ob with $x \in sinks(ob)$ is not **High**-typeable. Thus any obligation with $x \in sinks(ob)$ must be in obs_B. From this and $obs_B[k] =_L^{obs} obs'_B[k]$ we get that $x \in sinks(obs_B[k])$ if and only if $x \in sinks(obs'_B[k])$ and if this is the case in addition $obs_B[k] = obs'_B[k]$. From this combined with $gst_1 =_L^{Gst} gst'_3$ we get by definition of *specialize* that $ob'_1 = ob''_3$ and, in particular, $v = v''$.

In consequence we have $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_2 = gst'_3[i \mapsto (reg'_3[r \mapsto v], mem'_3)]$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i)$. From this combined with $gst_1 =_L^{Gst} gst'_3$ we get by transitivity and symmmetry of $=_L^{Gst}$ $gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst'_3[i \mapsto (reg'_3[r \mapsto v], mem'_3)]$. Hence, $gst_2 =_L^{Gst} gst'_2$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 =$

$\vec{cs}_3'$, $pa_2 \restriction_j \approx_L^{Ob^*} pa_2' \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst_2'$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle \approx_L^{Conf} \langle \vec{cs}_2', (pa_2', \vec{tr}_2'), gst_2' \rangle$.

Case ($ob_1 = x \leftarrow v@r$):

In this case we get from $ob_1 =_L^{obs} ob_3'$ that $ob_3' = x \leftarrow v'@r$. Hence, the only applicable rule for fulfilling obs_1 and ob_3' is the second rule in Figure 5 of the article. From definition of $specialize_\Psi$ we get $specialize_\Psi(pa_1[0 \ldots (m-1)], i, ob_1, gst_1) = ob_1$ and $specialize_\Psi(pa_3'[0 \ldots (m''-1)], i, ob_3', gst_3') = ob_3'$.

From the second rule in Figure 5 in the article we get $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}_2' = \vec{cs}_3'$, $gst_2 = gst_1[i \mapsto (\mathit{effect}(ob_1, gst_1[i]))]$ and $gst_2' = gst_3'[i \mapsto (\mathit{effect}(ob_3', gst_3'[i]))]$. Hence, from definition of *effect* we get $gst_2 = gst_1[i \mapsto (reg_1, mem_1[x \mapsto v])]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst_2' = gst_3'[i \mapsto (reg_3', mem_3'[x \mapsto v])]$ where $gst_3' = (\vec{reg}_3', mem_3')$ and $\vec{reg}_3'(i) = reg_3'$.

We distinguish two cases based on whether $lev(x) = \mathbf{High}$ or $lev(x) = \mathbf{Low}$.

Case ($lev(x) = \mathbf{High}$):

In this case $gst_1[i \mapsto (reg_1, mem_1[x \mapsto v])] =_L^{Gst} gst_1$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst_3'[i \mapsto (reg_3', mem_3'[x \mapsto v'])] =_L^{Gst} gst_3'$ where $gst_3' = (\vec{reg}_3', mem_3')$ and $\vec{reg}_3'(i) = reg_3'$ holds, according to the definition of $=_L^{Gst}$. Hence, $gst_2 =_L^{Gst} gst_1$ and $gst_2' =_L^{Gst} gst_3'$. From this we get by transitivity and symmetry of $=_L^{Gst}$ that $gst_2 =_L^{Gst} gst_2'$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \approx_L^{Conf} \langle \vec{cs}_3', (pa_3', \vec{tr}_3'), gst_3' \rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}_2' = \vec{cs}_3'$, $pa_2 \restriction_j \approx_L^{Ob^*} pa_2' \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst_2'$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle \approx_L^{Conf} \langle \vec{cs}_2', (pa_2', \vec{tr}_2'), gst_2' \rangle$.

Case ($lev(r) = \mathbf{Low}$):

In this case we get from $ob_1 =_L^{obs} ob_3'$ that $ob_1 = ob_3'$. Thus we get $v = v'$ and in consequence $gst_2 = gst_1[i \mapsto (reg_1, mem_1[x \mapsto v])]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst_2' = gst_3'[i \mapsto (reg_3', mem_3'[x \mapsto v])]$ where $gst_3' = (\vec{reg}_3', mem_3')$ and $\vec{reg}_3'(i)$. From this combined with $gst_1 =_L^{Gst} gst_3'$ we get by transitivity and symmmetry of $=_L^{Gst}$ $gst_1[i \mapsto (reg_1, mem_1[x \mapsto v])] =_L^{Gst} gst_3'[i \mapsto (reg_3', mem_3'[x \mapsto v])]$. Hence, $gst_2 =_L^{Gst} gst_2'$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \approx_L^{Conf} \langle \vec{cs}_3', (pa_3', \vec{tr}_3'), gst_3' \rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}_2' = \vec{cs}_3'$, $pa_2 \restriction_j \approx_L^{Ob^*} pa_2' \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst_2'$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle \approx_L^{Conf} \langle \vec{cs}_2', (pa_2', \vec{tr}_2'), gst_2' \rangle$.

Case ($ob_1 = v@\mathrm{const} \circlearrowright r$):

In this case we get from $ob_1 =_L^{obs} ob_3'$ that $ob_3' = v'@\mathrm{const} \circlearrowright r$. Hence, the only applicable rule for fulfilling obs_1 and ob_3' is the second rule in Figure 5 of the article. From definition of $specialize_\Psi$ we get $specialize_\Psi(pa_1[0 \ldots (m-1)], i, ob_1, gst_1) = ob_1$ and $specialize_\Psi(pa_3'[0 \ldots (m''-1)], i, ob_3', gst_3') = ob_3'$.

From the second rule in Figure 5 in the article we get $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}_2' = \vec{cs}_3'$, $gst_2 = gst_1[i \mapsto (\mathit{effect}(ob_1, gst_1[i]))]$ and $gst_2' = gst_3'[i \mapsto (\mathit{effect}(ob_3', gst_3'[i]))]$. Hence, from definition of *effect* we get $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst_2' = gst_3'[i \mapsto (reg_3'[r \mapsto v'], mem_3')]$ where $gst_3' = (\vec{reg}_3', mem_3')$ and $\vec{reg}_3'(i) = reg_3'$.

We distinguish two cases based on whether $lev(r) = \mathbf{High}$ or $lev(r) = \mathbf{Low}$.

Case ($lev(r) = \mathbf{High}$):

In this case $gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst_1$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst_3'[i \mapsto (reg_3'[r \mapsto v'], mem_3')] =_L^{Gst} gst_3'$ where $gst_3' = (\vec{reg}_3', mem_3')$ and $\vec{reg}_3'(i) = reg_3'$ holds, according to the definition of $=_L^{Gst}$. Hence, $gst_2 =_L^{Gst} gst_1$ and $gst_2' =_L^{Gst} gst_3'$. From this we get by transitivity and symmetry of $=_L^{Gst}$ that $gst_2 =_L^{Gst} gst_2'$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1 \rangle \approx_L^{Conf} \langle \vec{cs}_3', (pa_3', \vec{tr}_3'), gst_3' \rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}_2' = \vec{cs}_3'$, $pa_2 \restriction_j \approx_L^{Ob^*} pa_2' \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst_2'$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2 \rangle \approx_L^{Conf} \langle \vec{cs}_2', (pa_2', \vec{tr}_2'), gst_2' \rangle$.

Case ($lev(r) = \mathbf{Low}$):

In this case we get from $ob_1 =_L^{obs} ob_3'$ that $ob_1 = ob_3'$. Thus we get $v = v'$ and in consequence $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst_2' = gst_3'[i \mapsto (reg_3'[r \mapsto v], mem_3')]$ where $gst_3' = (\vec{reg}_3', mem_3')$ and $\vec{reg}_3'(i)$. From this combined with $gst_1 =_L^{Gst} gst_3'$ we get by transitivity and symmmetry of $=_L^{Gst}$ $gst_1[i \mapsto$

$(reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst'_3[i \mapsto (reg'_3[r \mapsto v], mem'_3)]$. Hence, $gst_2 =_L^{Gst} gst'_2$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst'_2$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case ($ob_1 = v@binop(r_2, r_3) \circlearrowright r_1$):

In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob'_3 = v'@binop(r_2, r_3) \circlearrowright r_1$. Hence, the only applicable rule for fulfilling obs_1 and ob'_3 is the second rule in Figure 5 of the article.

From definition of $specialize_\Psi$ we get $specialize_\Psi(pa_1[0 \dots (m-1)], i, ob_1, gst_1) = ob_1$ and $specialize_\Psi(pa'_3[0 \dots (m''-1)], i, ob'_3, gst'_3) = ob'_3$.

From the second rule in Figure 5 in the article we get $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $gst_2 = gst_1[i \mapsto (effect(ob_1, gst_1[i]))]$ and $gst'_2 = gst'_3[i \mapsto (effect(ob'_3, gst'_3[i]))]$. Hence, from definition of *effect* we get $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_2 = gst'_3[i \mapsto (reg'_3[r \mapsto v'], mem'_3)]$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i) = reg'_3$.

We distinguish two cases based on whether $lev(r) = \mathbf{High}$ or $lev(r) = \mathbf{Low}$.

Case ($lev(r) = \mathbf{High}$):

In this case $gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst_1$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_3[i \mapsto (reg'_3[r \mapsto v'], mem'_3)] =_L^{Gst} gst'_3$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i) = reg'_3$ holds, according to the definition of $=_L^{Gst}$. Hence, $gst_2 =_L^{Gst} gst_1$ and $gst'_2 =_L^{Gst} gst'_3$. From this we get by transitivity and symmetry of $=_L^{Gst}$ that $gst_2 =_L^{Gst} gst'_2$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst'_2$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case ($lev(r) = \mathbf{Low}$):

In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob_1 = ob'_3$. Thus we get $v = v'$ and in consequence $gst_2 = gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)]$ where $gst_1 = (\vec{reg}_1, mem_1)$ and $\vec{reg}_1(i) = reg_1$, and $gst'_2 = gst'_3[i \mapsto (reg'_3[r \mapsto v], mem'_3)]$ where $gst'_3 = (\vec{reg}'_3, mem'_3)$ and $\vec{reg}'_3(i)$. From this combined with $gst_1 =_L^{Gst} gst'_3$ we get by transitivity and symmmetry of $=_L^{Gst}$ $gst_1[i \mapsto (reg_1[r \mapsto v], mem_1)] =_L^{Gst} gst'_3[i \mapsto (reg'_3[r \mapsto v], mem'_3)]$. Hence, $gst_2 =_L^{Gst} gst'_2$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\vec{cs}_2 = \vec{cs}_1$, $\vec{cs}'_2 = \vec{cs}'_3$, $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ and $gst_2 =_L^{Gst} gst'_2$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case ($ob_1 =\nearrow_c$):

In this case we get from $ob_1 =_L^{obs} ob'_3$ that $ob'_3 =\nearrow_c$. Hence, the only applicable rule for fulfilling ob_1 and ob'_3 is the fourth rule in Figure 5 of the article.

From this rule we get $\vec{cs}_2(j) = \vec{cs}_1(j)$, $\vec{cs}'_2(j) = \vec{cs}'_3(j)$ for all $j \in pre(\vec{cs}_1)$, $\vec{cs}_2(max(pre(\vec{cs}_1)) + 1) = c$, $\vec{cs}'_2(max(pre(\vec{cs}_1)) + 1) = c$, $pa_2 = pa_1 \setminus m$, $pa'_2 = pa'_3 \setminus m''$, $mem_2 = mem_1$, $mem'_2 = mem'_3$, $\vec{reg}_2(j) = \vec{reg}_1(j)$, $\vec{reg}'_2(j) = \vec{reg}'_3(j)$ for all $j \in pre(\vec{cs}_1)$, $\vec{reg}_2(max(pre(\vec{cs}_1)) + 1) = reg_0$ and $\vec{reg}'_2(max(pre(\vec{cs}_1)) + 1) = reg_0$ where $gst_1 = (\vec{reg}_1, mem_1)$, $gst_2 = (\vec{reg}_2, mem_2)$, $gst'_3 = (\vec{reg}'_3, mem'_3)$, $gst'_2 = (\vec{reg}'_2, mem'_2)$ and $\forall r \in \mathcal{R}.reg_0(r) = 0$.

Since $\vec{reg}_2(max(pre(\vec{cs}_1)) + 1) = reg_0$ and $\vec{reg}'_2(max(pre(\vec{cs}_1)) + 1) = reg_0$ we get $\vec{reg}_2(max(pre(\vec{cs}_1)) + 1) = \vec{reg}'_2(max(pre(\vec{cs}_1)) + 1)$. From rule [PT] we get that $pc, pt \vdash_{lev} c \diamond (pt')$ is derivable for some $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$. From this combined with $\vec{cs}_2(max(pre(\vec{cs}_1)) + 1) = c$, $\vec{cs}'_2(max(pre(\vec{cs}_1)) + 1) = c$, $pa_2 \restriction_{max(pre(\vec{cs}_1))+1} = []$ and $pa'_2 \restriction_{max(pre(\vec{cs}_1))+1} = []$ we get $(\vec{cs}_2(max(pre(\vec{cs}_1)) + 1), pa_2 \restriction_{max(pre(\vec{cs}_1))+1}) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (\vec{cs}'_2(max(pre(\vec{cs}_1)) + 1), pa'_2 \restriction_{max(pre(\vec{cs}_1))+1})$.

Since $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ and $(\vec{cs}_2(max(pre(\vec{cs}_1)) + 1), pa_2 \restriction_{max(pre(\vec{cs}_1))+1}) \approx_L^{(\mathcal{C} \cup \{\epsilon\}) \times Ob^*} (\vec{cs}'_2(max(pre(\vec{cs}_1)) + 1), pa'_2 \restriction_{max(pre(\vec{cs}_1))+1})$ and $pa_2 \restriction_j \approx_L^{Ob^*} pa'_2 \restriction_j$ for all $j \in pre(\vec{cs}_1)$ we get $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$.

Case (thread progress):

In this case we know that the judgment was derived with the first rule from Figure 5 and from this rule we know that $gst_2 = gst_1$, $\forall n \in \{0,\dots,(|pa_1|-1)\}.\forall obs \in Fe.pa_1[n] \neq (i,ob)$, $\langle \vec{cs}_1(i), pa_1, \vec{reg}(i)\rangle \rightarrow_i \langle \vec{cs}_2(i), pa_2\rangle$, $pre(\vec{cs}_2) = pre(\vec{cs}_1)$ and $\vec{cs}_2(j) = \vec{cs}_1(j)$ for all $j \in pre(\vec{cs}_1)$ with $j \neq i$.
We distinguish three cases based on the condition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that is satisfied for $(\vec{cs}_1(i), pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (\vec{cs}'_1(i), pa'_1\restriction_i)$.

Case ($\vec{cs}_1(i) = \vec{cs}'_1(i)$):

In a first step, we ensure that the next step of $\vec{cs}'_1(i)$ is not blocked, i.e. there is no $ob \in Fe$ in the path and no update of a source register of the next instruction is pending.
From $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get by definition of $\approx_L^{Conf}$ and of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $pa_1\restriction_i \approx_L^{Ob^*} pa'_1\restriction_i$ holds.
From $pa_1\restriction_i \approx_L^{Ob^*} pa'_1\restriction_i$ we get by definition of $\approx_L^{Ob^*}$ that there is $obs_A, obs'_A, obs_B, obs'_B \in Ob^*$ with $pa_1\restriction_i = obs_A :: obs_B$, $pa'_1\restriction_i = obs'_A :: obs'_B$, $\mathbf{High} \vdash_{lev} obs_A$, $fencefree(obs_A)$, $\mathbf{High} \vdash_{lev} obs'_A$, $fencefree(obs'_A)$, $pt \vdash_{lev} obs_B$, $pt \vdash_{lev} obs'_B$, for some $pt \in \{\mathbf{Low}, \mathbf{High}\}$ and $(obs_B = [\|] \wedge obs'_B = []) \vee (obs_B = [] \wedge obs'_B = [\|]) \vee (|obs_B| = |obs'_B| \wedge \forall k < (|obs_B|).obs_B[k] =_L^{obs} obs'_B[k])$.
Since $\forall n \in \{0,\dots,(|pa_1|-1)\}.\forall obs \in Fe.pa_1[n] \neq (i,ob)$ holds, $obs_B = [\|] \wedge obs'_B = []$ cannot hold. Hence, $(obs_B = [] \wedge obs'_B = [\|]) \vee (|obs_B| = |obs'_B| \wedge \forall k < (|obs_B|).obs_B[k] =_L^{obs} obs'_B[k])$ holds.
From $\mathbf{High} \vdash_{lev} obs'_A$ and $fencefree(obs'_A)$ we get that $\mathbf{High} \vdash_{lev} obs'_A[0\dots k]$ for all $k < |obs'_A|$.
Combining the two facts from the two previous paragraphs, we can apply Lemma 37 and transitivity of $\approx_L^{Conf}$ $|obs'_A|$ times (and one additional time if $obs_B = [] \wedge obs'_B = [\|]$) to reach a configuration $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ with $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \Longrightarrow_{MM}^* \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ and $path'_3\restriction_i = obs''_B$ with $\vec{cs}'_3 = \vec{cs}'_1$ and $\forall k < (|obs_B|).obs_B[k] =_L^{obs} obs''_B[k])$.
Hence, in combination with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get by transitivity of $\approx_L^{Conf}$ that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$.
From $pa'_3\restriction_i = obs''_B$, $\forall k < (|obs_B|).obs_B[k] =_L^{obs} obs''_B[k])$ and $\forall n \in \{0,\dots,(|pa_1|-1)\}.\forall obs \in Fe.pa_1[n] \neq (i,ob)$ we get by definition of $=_L^{obs}$ that $\forall n \in \{0,\dots,(|pa'_3|-1)\}.\forall obs \in Fe.pa'_3[n] \neq (i,ob)$.
It remains to show that $\langle \vec{cs}'_3, pa'_3, \vec{reg}'_3(i)\rangle \rightarrow_i \langle c, pa'_2\rangle$ is derivable and $\langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle \approx_L^{Conf} \langle \vec{cs}'_3[i \mapsto c], (pa'_2, \vec{tr}'_3), gst'_3\rangle$ for some c and pa'_2.
To show that $\langle \vec{cs}'_3, pa'_3, \vec{reg}'_3(i)\rangle \rightarrow_i \langle c, pa'_2\rangle$ is derivable, we observe that the only conditions that could prohibit deriving $\langle \vec{cs}'_3, pa'_3, \vec{reg}'_3(i)\rangle \rightarrow_i \langle c, pa'_2\rangle$ for a not fixed c and pa'_2 are the requirements that check whether some source registers of instructions are updated in the list of obligations of thread i.
From $\vec{cs}_1(i) = \vec{cs}'_3(i)$ we get that $\vec{cs}'_3(i)$ reads from register r if and only if $\vec{cs}_1(i)$ reads from register r. Since $\langle \vec{cs}_1, pa_1, \vec{reg}(i)\rangle \rightarrow_i \langle \vec{cs}_2, pa_2\rangle$ is derivable we know that $r \notin sinks(pa_1\restriction_i)$ for all $r \in \mathcal{R}$ that $\vec{cs}_1(i)$ reads in this step. From this combined with $\vec{cs}'_3(i)$ reads from register r if and only if $\vec{cs}_1(i)$ reads from register r, $pa_1\restriction_i = obs_A :: obs_B$, $pa'_3\restriction_i = obs''_B$ and $\forall k < (|obs_B|).obs_B[k] =_L^{obs} obs''_B[k])$ we obtain by definition of $=_L^{obs}$ that $r \notin sinks(pa'_3\restriction_i)$ for all $r \in \mathcal{R}$ that $\vec{cs}'_3(i)$ reads in this step. Thus $\langle \vec{cs}'_3, pa'_3, \vec{reg}'_3(i)\rangle \rightarrow_i \langle c, pa'_2\rangle$ for some c and pa'_2.
From $\vec{cs}_1(i) = \vec{cs}'_3(i)$, $(\vec{cs}_1(i), pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (\vec{cs}'_3(i), pa'_3\restriction_i)$, $\vec{reg}_1(i) =_L^{\mathcal{R}} \vec{reg}'_3(i)$, $\langle \vec{cs}_1, pa_1, \vec{reg}_1(i)\rangle \rightarrow_i \langle \vec{cs}_2(i), pa_2\rangle$, $\langle \vec{cs}'_3, pa'_3, \vec{reg}'_3(i)\rangle \rightarrow_i \langle c, pa'_2\rangle$, $fencefree(pa_1\restriction_i)$ and $fencefree(pa'_3\restriction_i)$ we get by Lemma 32 that $(\vec{cs}_2(i), pa_2\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c, pa'_2\restriction_i)$, $pa_2\restriction_j = pa_1\restriction_j$, $pa'_2\restriction_j = pa'_2\restriction_j$ for all $j \neq i$.
From this combined with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ we get definition of $\approx_L^{Conf}$ that $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_1[i \mapsto c], (pa'_2, \vec{tr}'_3), gst'_3\rangle$.
From $\langle \vec{cs}'_3, pa'_3, \vec{reg}'_3(i)\rangle \rightarrow_i \langle c, pa'_2\rangle$ and $\forall n \in \{0,\dots,(|pa'_3|-1)\}.\forall obs \in Fe.pa'_3[n] \neq (i,ob)$ we get by the first rule of Figure 5 in the article that $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle \Longrightarrow_{MM} \langle \vec{cs}'_1[i \mapsto c], (pa'_2, \vec{tr}'_3), gst'_3\rangle$ is derivable.

Case ($pc = pt = pt' = \mathbf{High}$):

In this case $\mathbf{High} \vdash_{lev} pa_1\restriction_i$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} \vec{cs}_1(i) \diamond (\mathbf{High})$ holds. Thus we can apply Lemma 36 to obtain $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$. From this combined with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get by transitivity and symmetry of $\approx_L^{Conf}$ that $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$. Thus, we are done in this case.

Case ($\vec{cs}_1(i) = c_A;\ c_B \wedge \vec{cs}'_1(i) = c'_A;\ c'_B \wedge c_B = c'_B \wedge (c_A, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_A, pa'_1\restriction_i)$):

In this case each of the three conditions of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ can be satisfied for $(c_A, pa_1\restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (c'_A, pa'_1\restriction_i)$. If the first condition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ is satisfied, then this case is analogous the case $\vec{cs}_1(i) = \vec{cs}'_1(i)$. If the third condition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ is satisfied, we can again distinguish these three cases. The only distinct case is the case in which the second condition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ is satisfied. We provide the arguments for this case in detail in the following.

In this case $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$, $\mathbf{High} \vdash_{lev} pa'_1 \restriction_i$, $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$ and $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (\mathbf{High})$ are derivable.
From the semantics of sequential composition we get that $\langle c_A, pa_1, r\vec{e}g_1(i)\rangle \rightarrow_i \langle c_C, pa_2\rangle$ is derivable for some $c_C \in \mathcal{C} \cup \{\epsilon\}$. We distinguish two cases based on whether $c_C \in \mathcal{C}$ or $c_C = \epsilon$.

Case ($c_C \in \mathcal{C}$):

In this case we can apply Lemma 35 to obtain $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$.
From this combined with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get by transitivity and symmetry of $\approx_L^{Conf}$ that $\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$. Thus, we are done in this case.

Case ($c_C = \epsilon$):

In this case we first reduce the configuration $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ to a configuration where the $\mathbf{High}$ prefix of $\vec{cs}'_1(i)$ terminates in one step.
From $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get by definition of $\approx_L^{Conf}$ and of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $pa_1 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$ holds.
From $pa_1 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$ we get by definition of $\approx_L^{Ob^*}$ that there is $obs_A, obs'_A, obs_B, obs'_B \in Ob^*$ with $pa_1 \restriction_i = obs_A :: obs_B$, $pa'_1 \restriction_i = obs'_A :: obs'_B$, $\mathbf{High} \vdash_{lev} obs_A$, $fencefree(obs_A)$, $\mathbf{High} \vdash_{lev} obs'_A$, $fencefree(obs'_A)$, $pt \vdash_{lev} obs_B$, $pt \vdash_{lev} obs'_B$, for some $pt \in \{\mathbf{Low}, \mathbf{High}\}$ and $(obs_B = [\|] \wedge obs'_B = []) \vee (obs_B = [] \wedge obs'_B = [\|])$
$\vee (|obs_B| = |obs'_B| \wedge \forall k < (|obs_B|).obs_B[k] =_L^{obs} obs'_B[k])$.
Since the first rule of Figure 5 in the article is applicable to derive $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \Longrightarrow_{MM} \langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle$ for thread i we know that $\forall n \in \{0, \ldots, |pa_1| - 1\}. \forall ob \in Fe. pa_1[n] \neq (i, ob)$.
From $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ we get by definition of $\approx_L^{Conf}$ and of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $pa_1 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$ holds.
From $pa_1 \restriction_i \approx_L^{Ob^*} pa'_1 \restriction_i$ we get by definition of $\approx_L^{Ob^*}$ that there is $obs_A, obs'_A, obs_B, obs'_B \in Ob^*$ with $pa_1 \restriction_i = obs_A :: obs_B$, $pa'_1 \restriction_i = obs'_A :: obs'_B$, $\mathbf{High} \vdash_{lev} obs_A$, $fencefree(obs_A)$, $\mathbf{High} \vdash_{lev} obs'_A$, $fencefree(obs'_A)$, $pt \vdash_{lev} obs_B$, $pt \vdash_{lev} obs'_B$, for some $pt \in \{\mathbf{Low}, \mathbf{High}\}$ and $(obs_B = [\|] \wedge obs'_B = []) \vee (obs_B = [] \wedge obs'_B = [\|])$
$\vee (|obs_B| = |obs'_B| \wedge \forall k < (|obs_B|).obs_B[k] =_L^{obs} obs'_B[k])$. From this combined with $\forall n \in \{0, \ldots, |pa_1| - 1\}. \forall ob \in Fe. pa_1[n] \neq (i, ob)$ we get that $obs_B = [] \wedge obs'_B = [\|]$ is the only case in which an obligation $ob \in Fe$ appears in $pa'_1 \restriction_i$. Thus from $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$, $\mathbf{High} \vdash_{lev} obs'_A$ and $fencefree(obs'_A)$ we get that $\mathbf{High} \vdash_{lev} pa'_1 \restriction_i [0 \ldots n]$ holds for all $n < |pa'_1 \restriction_i|$.
Since $\mathbf{High} \vdash_{lev} pa'_1 \restriction_i [0 \ldots n]$ holds for all $n < |pa'_1 \restriction_i|$, $\vec{cs}'_1(i) = c'_A$; c'_B and $\mathbf{High}, \mathbf{High} \vdash_{lev} c'_A \diamond (\mathbf{High})$ we can apply Lemma 37 and Lemma 35 multiple times to obtain $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \Longrightarrow_{MM}^* \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ with $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$, $pa'_3 \restriction_i = []$, $\vec{cs}'_3(i) = c''_A$; c'_B and $\langle c''_A, pa'_3, reg'_3\rangle \rightarrow_i \langle \epsilon, pa'_2\rangle$ is derivable. From $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ and $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ we get by transitivity of $\approx_L^{Conf}$ that $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$.
In a second step we perform one execution step of each thread such that $\vec{cs}_2(i) = c_B$ and $\vec{cs}'_2(i) = c'_B$ and show that the resulting configurations are $\mathbf{Low}$-similar.
Since $\langle c''_A, pa'_3, reg'_3\rangle \rightarrow_i \langle \epsilon, pa'_2\rangle$ is derivable, $pa'_3 \restriction_i = []$ and $\vec{cs}'_3(i) = \vec{cs}''_A$; $\vec{cs}_B$ we obtain by the first rule of Figure 5 in the article and the rule for sequential composition that $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle \Longrightarrow_{MM} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle$ is derivable with $\vec{cs}'_2 = \vec{cs}'_3[i \mapsto c'_B]$.
Since $\mathbf{High}, \mathbf{High} \vdash_{lev} c''_A \diamond (\mathbf{High})$, $pa'_3 \restriction_i = []$ and $\mathbf{High} \vdash_{lev} []$ we obtain by Lemma 33 that $\mathbf{High} \vdash_{lev} pa'_2 \restriction_i$.
From the assumption of this case we get that $\langle c_A;\ c_B, pa_1, reg_1\rangle \rightarrow_i \langle c_B, pa_2\rangle$ is derivable with $\mathbf{High}, \mathbf{High} \vdash_{lev} c_A \diamond (\mathbf{High})$ and $\mathbf{High} \vdash_{lev} pa_1 \restriction_i$. Hence, we obtaing by Lemma 33 that $\mathbf{High} \vdash_{lev} pa_2 \restriction_i$.
From the semantics we get $pa_2 \restriction_i = pa_1 \restriction_i :: obs$ for some obs with $|obs| \leq 1$ and $pa'_2 \restriction_i = pa'_3 \restriction_i :: obs$ for some obs with $|obs| \leq 1$. From this combined with $fencefree(pa_1 \restriction_)$ and $fencefree(pa'_3 \restriction_)$ we get $\mathbf{High} \vdash_{lev} pa'_2 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa_2 \restriction_i [0 \ldots n]$ for all $n < pa_2 \restriction_i$ and $\mathbf{High} \vdash_{lev} pa'_2 \restriction_i [0 \ldots n]$ for all $n < pa'_2 \restriction_i$. From this combined with $pa_2 \restriction_= pa_1 \restriction_:: obs$ for some obs $|obs| \leq 1$ and $pa'_2 \restriction_= pa'_3 \restriction_:: obs$ for some obs $|obs| \leq 1$. From this combined with $fencefree(pa_1 \restriction_)$ and $fencefree(pa'_3 \restriction_)$ we get by definition of $\approx_L^{Ob^*}$ that $pa_2 \restriction_i \approx_L^{Ob^*} pa'_2 \restriction_i$.
Since $c_B = c'_B$ due to assumption of this case, $\vec{cs}_2(i) = c_B$, $\vec{cs}'_2(i) = c'_B$ and $pa_2 \restriction_i \approx_L^{Ob^*} pa'_2 \restriction_i$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $(\vec{cs}_2(i), pa_2 \restriction_i) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (\vec{cs}_2(i), pa'_2 \restriction_i)$.
From this combined with $\langle \vec{cs}_1, (pa_1, \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), gst'_3\rangle$ we get that

$$\langle \vec{cs}_2, (pa_2, \vec{tr}_2), gst_2\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), gst'_2\rangle.$$

■

The following lemma shows that a configuration that is **Low**-similar to a configuration of a terminated program run can perform a sequence of execution steps such that the resulting configuration is **Low**-similar to the configurations of the terminated run and the resulting configurations is a terminated configuration itself.

Lemma 39 (Termination)**.** *If two well-formed configurations satisfy* $\langle \vec{cs}_1, ([], \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ *and* $\vec{cs}_1(i) = \epsilon$ *for all* $i \in pre(\vec{cs}_1)$*, then there is* $\vec{cs}'_2$*,* $\vec{tr}'_2$*,* gst'_2 *such that* $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \Longrightarrow_{MM}^* \langle \vec{cs}'_2, ([], \vec{tr}'_2), gst'_2\rangle$ *with* $\langle \vec{cs}_1, ([], \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_2, ([], \vec{tr}'_2), gst'_2\rangle$ *and* $\vec{cs}'_2(i) = \epsilon$ *for all* $i \in pre(\vec{cs}'_2)$*.*

Proof: Let $MM \in \{\text{SC}, \text{IBM370}, \text{TSO}, \text{PSO}\}$, $\vec{cs}_1, \vec{cs}'_1 : \mathcal{I} \rightharpoonup \mathcal{C} \cup \{\epsilon\}$, $pa'_1 \in Pa$, $\vec{tr}_1, \vec{tr}'_1 \in \vec{Tr}$, gst_1, gst'_1 be arbitrary such that $\langle \vec{cs}_1, ([], \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ and $\vec{cs}_1(i) = \epsilon$ for all $i \in pre(\vec{cs}_1)$.

From $\langle \vec{cs}_1, ([], \vec{tr}_1), gst_1\rangle \approx_L^{Conf} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ and $\vec{cs}_1(i) = \epsilon$ for all $i \in pre(\vec{cs}_1)$ we get by definition of $\approx_L^{Conf}$ that $pre(\vec{cs}_1) = pre(\vec{cs}'_1)$, $(\vec{cs}_1(i), []) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (\vec{cs}'_1(i), pa'_1 \restriction_i)$ holds for all $i \in pre(\vec{cs}_1)$ and $gst_1 =_L^{Gst} gst'_1$.

From $(\vec{cs}_1(i), []) \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} (\vec{cs}'_1(i), pa'_1 \restriction_i)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $[] \approx_L^{Ob^*} pa'_1 \restriction_i$. From this we get by definition of $\approx_L^{Ob^*}$ that $\mathbf{High} \vdash_{lev} pa'_1[0 \dots m] \restriction_i$ for all $m < |pa'_1|$.

Since $\vec{cs}_1(i) = \epsilon$ for all $i \in pre(\vec{cs}_1)$ we get by definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that the third condition in the definition of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ cannot hold and thus one of the following conditions holds for all $i \in pre(\vec{cs}_1)$:

- $\vec{cs}'_1(i) = \epsilon$ and $pa'_1 \restriction_i = []$, or
- $\mathbf{High}, pt \vdash_{lev} \vec{cs}'_1(i) \diamond (pt')$ and $\mathbf{High} \vdash_{lev} pa'_1[0 \dots m] \restriction_i$ for all $m < |pa'_1|$.

We need to show that for all $i \in pre(\vec{cs}'_1)$ the thread i can proceed into a configuration such that nothing remains to be executed and the thread has no obligations assumed. To this end, we fix an arbitrary $i \in \mathcal{I}$.

Note that the first condition implies the second condition due to the rule [EM]. Hence, we consider only the case that the second rule is satisfied.

From $\mathbf{High}, pt \vdash_{lev} \vec{cs}'_1(i) \diamond (pt')$ and $\mathbf{High} \vdash_{lev} pa'_1[0 \dots m] \restriction_i$ for all $m < |pa'_1|$ we get by applying Lemma 36 and Lemma 37 multiple times that $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \Longrightarrow_{MM}^* \langle \vec{cs}'_2, ([], \vec{tr}'_2), gst'_2\rangle$ is derivable with $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} \langle \vec{cs}'_2, ([], \vec{tr}'_2), gst'_2\rangle$, $\vec{cs}'_2(i) = \epsilon$ and $\vec{cs}'_2(j) = \vec{cs}'_1(j)$ for all $j \in pre(\vec{cs}'_2)$ where $j \neq i$.

From $\langle \vec{cs}_1, ([], \vec{tr}_1), gst_1\rangle \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} \langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle$ and $\langle \vec{cs}'_1, (pa'_1, \vec{tr}'_1), gst'_1\rangle \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} \langle \vec{cs}'_2, ([], \vec{tr}'_2), gst'_2\rangle$ we get by transitivity of $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ that $\langle \vec{cs}_1, ([], \vec{tr}_1), gst_1\rangle \approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*} \langle \vec{cs}'_2, ([], \vec{tr}'_2), gst'_2\rangle$. ■

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

Proof: Let $c \in \mathcal{C}$ be arbitrary such that $pc, \mathbf{High} \vdash_{lev} c \diamond (pt)$.

According to definition of NI_{MM} we must show that for all $MM \in \{\text{SC}, \text{IBM370}, \text{TSO}, \text{PSO}\}$ and $mem_1, mem_2, mem'_1 \in Mem$ with $\langle c, mem_1\rangle \Downarrow_{MM} mem_2$ and $mem_1 =_L^{Gst} mem'_1$ there is mem'_2 such that $\langle c, mem'_1\rangle \Downarrow_{MM} mem'_2$ and $mem_2 =_L^{Gst} mem'_2$.

Let $MM \in \{\text{SC}, \text{IBM370}, \text{TSO}, \text{PSO}\}$ and $mem_1, mem_2, mem'_1 \in Mem$ be arbitrary such that the previous conditions are fulfilled.

From $\langle c, mem_1\rangle \Downarrow_{MM} mem_2$ we get by the rule for deriving this judgment that there is $\vec{cs}_1, \vec{cs}_2 : \mathcal{I} \to \mathcal{C}$, $\vec{tr}_1, \vec{tr}_2 \in \vec{Tr}$ and $\vec{reg}_1, \vec{reg}_2 : \mathcal{I} \to Reg$ such that $pre(\vec{cs}_1) = pre(\vec{tr}_1) = pre(\vec{reg}_1) = \{0\}$, $\vec{cs}_1(0) = c$, $\vec{tr}_1(0) = []$, $\forall r \in \mathcal{R}.\vec{reg}_1(0)(r) = 0$, $\forall i \in pre(\vec{cs}_2).\vec{cs}_2(i) = \epsilon$ and $\langle \vec{cs}_1, ([], \vec{tr}_1), (\vec{reg}_1, mem_1)\rangle \Longrightarrow_{MM}^* \langle \vec{cs}_2, ([], \vec{tr}_2), (\vec{reg}_2, mem_2)\rangle$ and that we must show that $\langle \vec{cs}_1, ([], \vec{tr}_1), (\vec{reg}_1, mem'_1)\rangle \Longrightarrow_{MM}^* \langle \vec{cs}_2, ([], \vec{tr}_2), (\vec{reg}_2, mem_2)'\rangle$ is derivable.

From $\vec{cs}_1(0) = c$, $pre(\vec{cs}_1) = 0$ and $pc, \mathbf{High} \vdash_{lev} c \diamond (pt)$ we get that $pc, \mathbf{High} \vdash_{lev} \vec{cs}_1(i) \diamond (pt)$ is derivable for all $i \in pre(\vec{cs}_1)$.

By the rule [PE] we get that $pt \vdash_{lev} [] \restriction_i$ is derivable for all $i \in pre(\vec{cs}_1)$.

From $pc, \mathbf{High} \vdash_{lev} \vec{cs}_1(i) \diamond (pt)$ is derivable for all $i \in pre(\vec{cs}_1)$ and $pt \vdash_{lev} [] \restriction_i$ is derivable for all $i \in pre(\vec{cs}_1)$ we get by the rule [CS] that $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}_1, ([], \vec{tr}_1), (\vec{reg}_1, mem_1)\rangle$ and $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}_1, ([], \vec{tr}_1), (\vec{reg}_1, mem'_1)\rangle$ are derivable for some $\vec{pc} : \mathcal{I} \rightharpoonup \{\mathbf{Low}, \mathbf{High}\}$ and $\vec{pt} : \mathcal{I} \rightharpoonup \{\mathbf{Low}, \mathbf{High}\}$.

From $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem_1)\rangle$, $\vec{pc}, \vec{pt} \vdash_{lev} \langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem'_1)\rangle$, $(\vec{cs}_1(i), [] \restriction_i) = (\vec{cs}_1(i), [] \restriction_i)$ for all $i \in pre(\vec{cs}_1)$, $r\vec{eg}_1 = r\vec{eg}_1$ and $mem_1 =_L mem'_1$ we get by definition of $\approx_L^{Conf}$, $\approx_L^{(\mathcal{C}\cup\{\epsilon\})\times Ob^*}$ and $=_L^{Gst}$ that $\langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem_1)\rangle \approx_L^{Conf} \langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem'_1)\rangle$.

From $\langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem_1)\rangle \approx_L^{Conf} \langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem'_1)\rangle$ and $\langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem_1)\rangle \Longrightarrow_{MM}^* \langle \vec{cs}_2, ([], \vec{tr}_2), (r\vec{eg}_2, mem_2)\rangle$ in n steps we get by applying Lemma 38 n times that $\langle \vec{cs}_1, ([], \vec{tr}_1), (r\vec{eg}_1, mem_1)\rangle \Longrightarrow_{MM}^* \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), (r\vec{eg}'_3, mem'_3)\rangle$ is derivable with $\langle \vec{cs}_2, ([], \vec{tr}_2), (r\vec{eg}_2, mem_2)\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), (r\vec{eg}'_3, mem'_3)\rangle$.

From $\langle \vec{cs}_2, ([], \vec{tr}_2), (r\vec{eg}_2, mem_2)\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), (r\vec{eg}'_3, mem'_3)\rangle$ and $\vec{cs}_2(i) = \epsilon$ for all $i \in pre(\vec{cs}_2)$ we get by Lemma 39 that $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), (r\vec{eg}'_3, mem'_3)\rangle \Longrightarrow_{MM}^* \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), (r\vec{eg}'_2, mem'_2)\rangle$ with $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), (r\vec{eg}'_3, mem'_3)\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), (r\vec{eg}'_2, mem'_2)\rangle$ and $\vec{cs}'_2(i) = \epsilon$ for all $i \in pre(\vec{cs}'_2)$.

From $\langle \vec{cs}_2, ([], \vec{tr}_2), (r\vec{eg}_2, mem_2)\rangle \approx_L^{Conf} \langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), (r\vec{eg}'_3, mem'_3)\rangle$ and $\langle \vec{cs}'_3, (pa'_3, \vec{tr}'_3), (r\vec{eg}'_3, mem'_3)\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), (r\vec{eg}'_2, mem'_2)\rangle$ we get by transitivity of $\approx_L^{Conf}$ that $\langle \vec{cs}_2, ([], \vec{tr}_2), (r\vec{eg}_2, mem_2)\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), (r\vec{eg}'_2, mem'_2)\rangle$.

From $\langle \vec{cs}_2, ([], \vec{tr}_2), (r\vec{eg}_2, mem_2)\rangle \approx_L^{Conf} \langle \vec{cs}'_2, (pa'_2, \vec{tr}'_2), (r\vec{eg}'_2, mem'_2)\rangle$ we get by definition of $\approx_L^{Conf}$ on configuraitons and $=_L^{Gst}$ on global states that $mem_2 =_L mem'_2$. ■

Theorem 3 (Soundness of Transformation)**.** *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 all* $MM \in \{SC, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$.

Proof: From $pc, \mathbf{High} \vdash_{lev} c \diamond (pt, c')$ we get by Lemma 21 that $pc, \mathbf{High} \vdash_{lev} c' \diamond (pt)$ and from $pc, \mathbf{High} \vdash_{lev} c' \diamond (pt)$ we get by Theorem 2 that $c' \in NI_{MM}$ for all $MM \in \{\mathrm{SC}, \mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$. ■

F. Proofs for showing that transformation does not enforce sequential consistency

In this section we prove that the transformation does not enforce sequentialy consistent behavior for the transformed programs (Theorem 3 from the article). For this purpose we recall in Figure 6 the example programs from Figure 13 in the article.

$c = c_1;\ \mathbf{if}_{14}\ \mathrm{r}_1\ \mathbf{then}\ \mathbf{fence}_{15}\ \mathbf{else}\ \mathbf{skip}_{16}\ \mathbf{fi};\ c_2$

$c' = c_1;\ \mathbf{fence}_{18};\ \mathbf{if}_{14}\ \mathrm{r}_1\ \mathbf{then}\ \mathbf{fence}_{15}\ \mathbf{else}\ \mathbf{skip}_{16}\ \mathbf{fi};\ c_2$

where

$c_1 = \mathbf{load}_1\ \mathrm{r}_1\ \mathrm{h};\ \mathbf{load}_2\ \mathrm{r}_2\ 0;\ \mathbf{load}_3\ \mathrm{r}_3\ 1;\ \mathbf{spawn}_4(c_S);\ \mathbf{store}_{12}\ \mathrm{x}\ \mathrm{r}_2;\ \mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$

$c_S = \mathbf{load}_5\ \mathrm{r}_4\ \mathrm{z};\ \mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y};\ \mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x};\ \mathbf{and}_8\ \mathrm{r}_7\ \mathrm{r}_4\ \mathrm{r}_6;\ \mathbf{and}_9\ \mathrm{r}_8\ \mathrm{r}_5\ \mathrm{r}_6;\ \mathbf{store}_{10}\ \mathrm{l}_1\ \mathrm{r}_7;\ \mathbf{store}_{11}\ \mathrm{l}_2\ \mathrm{r}_8$

$c_2 = \mathbf{store}_{17}\ \mathrm{z}\ \mathrm{r}_3$

$lev(\mathrm{h}) = lev(\mathrm{r}_1) = \mathbf{High}$,
$lev(x) = \mathbf{Low}$ for all $x \in \mathcal{X} \setminus \{\mathrm{h}\}$
$lev(r) = \mathbf{Low}$ for all $r \in \mathcal{R} \setminus \{\mathrm{r}_1\}$.

Figure 6. Transformed program without sequentially consistent behavior

The following program shows that the program c from Figure 6 is transformed to the program c' from Figure 6 by our transformation, given the domain assignment is lev from Figure 6.

Lemma 40. *The judgment* $\mathbf{Low}, \mathbf{High} \vdash_{lev} c \diamond (\mathbf{Low}, c')$ *is derivable for c, c′ and lev from Figure 6.*

Proof: Using the rule [SQ] multiple times together [LX], [OP] and [ST], the judgment $\mathbf{Low}, \mathbf{High} \vdash_{lev} c_S \diamond (\mathbf{Low}, c_S)$ is derivable, because $pc = \mathbf{Low}$ and $lev(xr) = \mathbf{Low}$ holds for all $xr \in \{\mathrm{r}_4, \mathrm{r}_5, \mathrm{r}_6, \mathrm{r}_7, \mathrm{r}_8, \mathrm{l}_1, \mathrm{l}_2\}$, i.e. all variables that are accessed in c_S, and hence $pc \sqcup lev(xr) \sqsubseteq lev(xr')$ for all $xr, xr' \in \{\mathrm{r}_4, \mathrm{r}_5, \mathrm{r}_6, \mathrm{r}_7, \mathrm{r}_8, \mathrm{l}_1, \mathrm{l}_2\}$.

Using the rule [SQ] multiple times together [LX], [LC], [SP], [ST] and [OP] the judgment $\mathbf{Low}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{Low}, c_1)$ is derivable, because $pc = \mathbf{Low}$ and the following three reasons:

First, $lev(xr) = \mathbf{Low}$ for all $xr \in \{\mathrm{r}_2, \mathrm{r}_3, \mathrm{r}_3, \mathrm{x}, \mathrm{y}\}$, and hence $pc \sqcup lev(xr) \sqsubseteq lev(xr')$, i.e. all variables that are accessed in all commands except $\mathbf{load}_1$, and hence $pc \sqcup lev(xr) \sqsubseteq lev(xr')$ for all $xr, xr' \in \{\mathrm{r}_2, \mathrm{r}_3, \mathrm{r}_3, \mathrm{x}, \mathrm{y}\}$.

Second, $lev(\mathrm{r}_1) = \mathbf{High}$, i.e. that register that is written in command $\mathbf{load}_1$, and hence $pc \sqcup lev(\mathrm{h}) \sqsubseteq lev(\mathrm{r}_1)$.

Third, $pc, pt \vdash_{lev} c_S \diamond (pt', c_S)$ is derivable for $pc = \mathbf{Low}$, $pt = \mathbf{High}$ and $pt' = \mathbf{Low}$ is derivable.

Using the rule [ST], the judgment $\mathbf{Low}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{Low}, c_S)$ is derivable, because $lev(\mathrm{r}_3) \sqsubseteq lev(\mathrm{z})$.

Using the rules [IT], the judgment
$\mathbf{Low}, \mathbf{Low} \vdash_{lev} \mathbf{if}_{14}\ \mathrm{r}_1\ \mathbf{then}\ \mathbf{fence}_{15}\ \mathbf{else}\ \mathbf{skip}_{16}\ \mathbf{fi} \diamond (\mathbf{High}, \mathbf{fence}_{18};\ \mathbf{if}_{14}\ \mathrm{r}_1\ \mathbf{then}\ \mathbf{fence}_{15}\ \mathbf{else}\ \mathbf{skip}_{16}\ \mathbf{fi})$ is derivable, because $lev(\mathrm{r}_1) = \mathbf{High}$, $\mathbf{High}, \mathbf{High} \vdash_{lev} \mathbf{fence}_{15} \diamond (\mathbf{High}, \mathbf{fence}_{15})$ is derivable with rule [FN] and $\mathbf{High}, \mathbf{High} \vdash_{lev} \mathbf{skip}_{16} \diamond (\mathbf{High}, \mathbf{skip}_{16})$ is derivable with rule [SK].

From the derived fact in the last paragraph combined with $\mathbf{Low}, \mathbf{High} \vdash_{lev} c_1 \diamond (\mathbf{Low}, c_1)$, and $\mathbf{Low}, \mathbf{High} \vdash_{lev} c_2 \diamond (\mathbf{Low}, c_S)$ we get by multiple applications of [SQ] that $\mathbf{Low}, \mathbf{High} \vdash_{lev} c \diamond (\mathbf{Low}, c')$ is derivable. ■

Lemma 41. *The judgment $\langle c', mem\rangle \Downarrow_{\mathrm{PSO}} mem'$ with $mem(\mathrm{x}) = 1$ and $mem(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{x}\}$ and $mem'(\mathrm{l}_2) = 1$ is derivable for c' from Figure 6.*

Proof: Let $1, 2, 3, 4, 5, \mathbf{13}, 6, 7, 8, 9, 10, 11, \mathbf{12}, 18, 17$ be a sequence for fulfilling obligations of commands with the corresponding identifiers in a program run of c' from Figure 6.

This sequence is possible, because all obligations of commands except of $\mathbf{store}_{12}\ \mathrm{x}\ \mathrm{r}_2$ and $\mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$ are fulfilled in the order in which they were assumed, the obligations $\mathbf{store}_{12}\ \mathrm{x}\ \mathrm{r}_2$ and $\mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$ may be fulfilled out-of-order due to $\phi_{\mathrm{WW}} \in \Phi$ for PSO, and r_1 is 0 when $\mathbf{if}_{14}\ \mathrm{r}_1$ is fetched (because r_1 is updated to the initial value of h by $\mathbf{load}_1\ \mathrm{r}_1\ \mathrm{h}$).

This sequence results in a final memory mem' with $mem'(\mathrm{l}_2) = 1$, because

1) $\mathbf{store}_{11}\ \mathrm{l}_2\ \mathrm{r}_8$ updates l_2 to the value of r_8 obtained by $\mathbf{and}_9\ \mathrm{r}_8\ \mathrm{r}_5\ \mathrm{r}_6$,
2) $\mathbf{and}_9\ \mathrm{r}_8\ \mathrm{r}_5\ \mathrm{r}_6$ updates r_8 to the conjunction of the values of r_5 and r_6 obtained by $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$ and $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$,
3) $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$ updates r_6 to the initial value of x and this value is 1
4) $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$ updates r_5 to the value of y obtained by $\mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$
5) $\mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$ updates y to the value of r_3 obtained by $\mathbf{load}_7\ \mathrm{r}_3\ 1$ and this value is 1.

■

Lemma 42. *The judgment $\langle c', mem\rangle \Downarrow_{\mathrm{SC}} mem'$ with $mem(\mathrm{x}) = 1$ and $mem(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{x}\}$ and $mem'(\mathrm{l}_2) = 1$ is not derivable for c' from Figure 6.*

Proof: Since the initial value of l_2 is 0, i.e. $mem(\mathrm{l}_2) = 0$, the variable l_2 must be updated to reach a final memory mem' for which $mem'(\mathrm{l}_2) = 1$ holds. The only update of l_2 is $\mathbf{store}_{11}\ \mathrm{l}_2\ \mathrm{r}_8$ and, consequently, $\mathbf{store}_{11}\ \mathrm{l}_2\ \mathrm{r}_8$ must update l_2 to 1 to reach a final memory mem' for which $mem'(\mathrm{l}_2) = 1$ holds.

Since the initial value of r_8 is 0, r_8 must be updated. The only update of r_8 is $\mathbf{and}_9\ \mathrm{r}_8\ \mathrm{r}_5\ \mathrm{r}_6$. According to the semantics of **and**, $\mathbf{and}_9\ \mathrm{r}_8\ \mathrm{r}_5\ \mathrm{r}_6$ updates r_8 only to 1 if the values of r_5 and r_6 are both unequal to 0 when assuming the obligation. Since r_5 is initially 0, r_5 must be updated before assuming the obligation. The only update of r_5 is $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$. Since y is initially 0, y must be updated before fulfilling the obligation.

The only update of y is $\mathbf{store}_{13}\ \mathrm{y}\ \mathrm{r}_3$. Since sequential consistency requires that all obligations are fulfilled in the order in which they were assumed, this means that $\mathbf{store}_{12}\ \mathrm{x}\ \mathrm{r}_2$ must also be fulfilled before the obligation of $\mathbf{load}_6\ \mathrm{r}_5\ \mathrm{y}$ and that $\mathbf{store}_{12}\ \mathrm{x}\ \mathrm{r}_2$ must also be fulfilled before the obligation of $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$. Since r_2 is initially is initially 0 and the only update of r_2, i.e. $\mathbf{load}_2\ \mathrm{r}_2\ 0$, updates r_2 to 0, this means that x is updated to 0 before $\mathbf{load}_7\ \mathrm{r}_6\ \mathrm{x}$ is fulfilled in all program runs in which r_5 is updated to 1. Consequently, it is not possible that both registers r_5 and r_6 are 1 when assuming the obligation of $\mathbf{and}_9\ \mathrm{r}_8\ \mathrm{r}_5\ \mathrm{r}_6$ and, hence, it is not possible that $\mathbf{store}_{11}\ \mathrm{l}_2\ \mathrm{r}_8$ updates l_2 to 1. Hence, no final memory mem' for which $mem'(\mathrm{l}_2) = 1$.

This finally shows that $\langle c', mem\rangle \Downarrow_{\mathrm{SC}} mem'$ with $mem(\mathrm{x}) = 1$ and $mem(x) = 0$ for all $x \in \mathcal{X} \setminus \{\mathrm{x}\}$ and $mem'(\mathrm{l}_2) = 1$ is not derivable for c' from Figure 6. ■

We recall Theorem 3 from the article:

Theorem 4. *The fact that $pc, \mathbf{High} \vdash_{lev} c \diamond (pt, c')$ for some $pc, pt \in \{\mathbf{Low}, \mathbf{High}\}$ is derivable does not imply that $\langle c', mem\rangle \Downarrow_{MM} mem' \iff \langle c', mem\rangle \Downarrow_{\mathrm{SC}} mem'$ for all $MM \in \{\mathrm{IBM370}, \mathrm{TSO}, \mathrm{PSO}\}$ holds.*

Proof: The programs c and c' with the domain assignment lev from Figure 6 are a concrete counter example according to Lemmas 40, 41 and 42. ■

G. Proofs for Idempotency of the Transformation

In this section we prove that the transforming type system is idempotent. We recall Theorem 4 from the article:

Theorem 5. *If $pc, \mathbf{High} \vdash_{lev} c \diamond (pt, c')$ is derivable for some $pc, pt \in \{\mathbf{Low}, \mathbf{High}\}$, then $pc, \mathbf{High} \vdash_{lev} c' \diamond (pt', c')$ is also derivable.*

Proof: We prove the more general proposition:

If $pc, pt \vdash_{lev} c \diamond (pt', c')$ is derivable for some $pc, pt, pt' \in \{\mathbf{Low}, \mathbf{High}\}$, then $pc, pt \vdash_{lev} c' \diamond (pt', c')$ is also derivable.

We prove this by an induction on the length of the derivation for the judgment $pc, pt \vdash_{lev} c \diamond (pt', c')$.

The induction base are derivations with a length of 1. These derivations are only possible with the rules [SK], [FN], [LC], [LX], [OP], [OP], [ST]. These rule do not perform any transformation. Thus the proposition holds for these cases.

As induction step we assume that the theorem holds for derivations with arbitrary length $n' \geq 1$. For the induction step let the derivation length n be $n' + 1$. These derivations are possible with the rules [SP], [SQ], [IL], [IH], [IT] and [WH]. Note that among these rules there is only one rule, namely [IT], that actually performs a transformation directly. In all other cases we can apply the induction hypothesis to obtain that the proposition holds. Thus we focus on the rule [IT].

Only the rule [IT] performs a transformation. From the typing rule [IT] we know that the original instruction is $\mathbf{if}_\iota\ r\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}$ with $\mathbf{High}, \mathbf{High} \vdash_{lev} c_n \diamond (\mathbf{High}, c_n')$ for $n \in \{1, 2\}$ for some $\iota \in \mathbb{N}$, $c_1, c_1', c_2, c_2' \in \mathcal{C}$ and $r \in \mathcal{R}$ with $\mathbf{High}, \mathbf{High} \vdash_{lev} c_n \diamond (\mathbf{High}, c_n')$ for $n \in \{1, 2\}$ and $lev(r) = \mathbf{High}$ and that $pt' = \mathbf{High}$. From the typing rule, we also know that the transformed instruction is $\mathbf{fence}_{\iota'};\ \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$ for some $\iota' \in \mathbb{N}$, $c_1', c_2' \in \mathcal{C}$.

From $\mathbf{fence}_{\iota'};\ \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$ we get by rule [SQ] that in the second application of the transformation $pc, pt \vdash_{lev} \mathbf{fence}_{\iota'} \diamond (pt'', c_F)$ and $pc, pt'' \vdash_{lev} \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi} \diamond (\mathbf{High}, c_I)$ must be derivable and the transformed program is $c_F;\ c_I$.

From the rule [FN] we get that $pc, pt \vdash_{lev} \mathbf{fence}_{\iota'} \diamond (\mathbf{High}, \mathbf{fence}_{\iota'})$ is derivable and thus $c_F = \mathbf{fence}_{\iota'}$ and $pt' = \mathbf{High}$.

From $pt = \mathbf{High}$ and $lev(r) = \mathbf{High}$ we get that the only applicable rule is [IH]. From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_n \diamond (\mathbf{High}, c_n')$ we get by the induction hypothesis that $\mathbf{High}, \mathbf{High} \vdash_{lev} c_n' \diamond (\mathbf{High}, c_n')$ for $n \in \{1, 2\}$ and $lev(r) = \mathbf{High}$. From $\mathbf{High}, \mathbf{High} \vdash_{lev} c_n' \diamond (\mathbf{High}, c_n')$ for $n \in \{1, 2\}$ and $lev(r) = \mathbf{High}$ we get by rule [IH] that $c_I = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$.

From $pc, pt \vdash_{lev} \mathbf{fence}_{\iota'} \diamond (\mathbf{High}, c_F)$, $pc, pt'' \vdash_{lev} \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi} \diamond (\mathbf{High}, c_I)$, $c_F = \mathbf{fence}_{\iota'}$ and $c_I = \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi}$ we get that $pc, pt \vdash_{lev} \mathbf{fence}_{\iota'};\ \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi} \diamond (\mathbf{High}, \mathbf{fence}_{\iota'};\ \mathbf{if}_\iota\ r\ \mathbf{then}\ c_1'\ \mathbf{else}\ c_2'\ \mathbf{fi})$ is derivable. ■