2012 IEEE 25th Computer Security Foundations Symposium

Secure Information Flow for Concurrent Programs under Total Store Order

Jeffrey A. Vaughan
University of California, Los Angeles

Todd Millstein
University of California, Los Angeles

***Abstract*—Modern multicore hardware and multithreaded programming languages expose *weak memory models* to programmers, which relax the intuitive sequential consistency (SC) memory model in order to support a variety of hardware and compiler optimizations. However, to our knowledge all prior work on secure information flow in a concurrent setting has assumed SC semantics. This paper investigates the impact of the Total Store Order (TSO) memory model, which is used by Intel x86 and Sun SPARC processors, on secure information flow, focusing on the natural security condition known as possibilistic noninterference. We show that possibilistic noninterference under SC and TSO are incomparable notions; neither property implies the other one. We define a simple type system for possibilistic noninterference under SC and demonstrate that it is unsound under TSO. We then provide two variants of this type system that are sound under TSO: one that requires only a small change to the original type system but is overly restrictive, and another that incorporates a form of flow sensitivity to safely retain additional expressiveness. Finally, we show that the original type system is in fact sound under TSO for programs that are free of data races.**

***Index Terms*—information flow, language-based security, weak memory models**

I. Introduction

A *memory model* [1, 2] forms the foundation of shared-memory multithreaded programming by defining the set of possible orders in which memory operations can execute and become visible to other threads. To our knowledge, all prior work on information flow for multithreaded programs has assumed *sequential consistency* (SC) [9], which requires all memory operations to appear to have executed in a global sequential order consistent with the per-thread program order. SC is the most natural memory model for programmers, since it accords with the intuition of a multithreaded program's behavior as the set of all thread interleavings. However, mainstream hardware architectures (e.g., x86 [13] and POWER [17]) and programming languages (e.g., Java [10] and C++ [4]) instead expose a *weak* memory model to programmers, which can exhibit subtle non-SC behaviors but gain the ability to perform a variety of compiler and hardware optimizations. Therefore, prior results on concurrent information flow are not immediately applicable to today's hardware and software platforms.

This paper explores the implications for secure information flow of the Total Store Order (TSO) memory model. TSO is a natural starting point for understanding how weak memory models interact with secure information flow for several reasons. First, TSO is used by common hardware platforms today, including Intel x86 and Sun SPARC processors, and it has been recently explored as a memory model for concurrent programming languages as well [11, 23]. Second, TSO represents a relatively small weakening of SC and has a natural operational interpretation [13]. Third, understanding the impact of TSO is a first step to understanding the impact of the other memory models used by hardware today, for example those of ARM and IBM POWER processors [17], which are strictly weaker than TSO.

```
X := 0;
Y := 0;
fork (X := 1;
      y := Y);
Y := 1;
x := X
```

Fig. 1. TSO relaxes write-before-read program order dependencies, which allows x and y to both end with value 0.

In the TSO memory model, a store in thread t can become visible to other threads after a later non-dependent load on thread t. For instance, in the canonical example[1] shown in Figure 1, it is possible for both x and y to be assigned the value 0, while at least one of these variables is assigned the value 1 in any SC execution. Under TSO, a thread can also read its own stores before they become visible to other threads. For instance, in the example shown in Figure 2, it is possible to end in a state where X has the value 1, y has the value 0, and x has the value 1. If the x := X assignment could not read X early, then in any final state where X and y map to 1 and 0 variable x would have value 0.

Operationally, TSO's relaxation of SC can be accounted for by the presence of FIFO *write buffers* in hardware, which allow a processor to execute later instructions before pending writes have committed to memory [13]. In our first example above, the writes of 1 to X and Y are buffered, the subsequent loads read the value 0 from main memory, and finally the two buffered writes commit. Our second example is accounted for by TSO's support for *store-to-load forwarding*, whereby a load searches the processor's write buffer for a value before

This research was supported by the National Science Foundation under award CNS-1064844.

[1]Examples use a simple imperative language augmented with the ability to fork new threads. Uppercase variables are shared across threads while lowercase variables are thread-local temporaries. We formalize this language in Section II.

DOI 10.1109/CSF.2012.20

```
X := 0;
Y := 0;
fork (X := 1;
      x := X;
      y := Y);
Y := 2;
X := 2
```

Fig. 2. Under TSO a thread can read its own writes early, which allows this program to end in a state where X has the value 1, y has the value 0, and x has the value 1.

accessing main memory. In the example, the read of X in the forked thread occurs before the prior write to X commits to memory but still sees the value of that write.

In this paper we explore the impact of TSO on *possibilistic noninterference* [19], which is an intuitive generalization of the traditional notion of noninterference to a concurrent setting. We leave exploration of the impact of weak memory models on stronger notions of security, for example those that take into account the probability distribution of outputs due to the thread scheduler [22], to future work.

This paper provides several contributions.

- We define a simple formal language to investigate secure information flow under TSO (Section II). The language includes a standard imperative core along with constructs for dynamic thread creation, lock-based synchronization, and memory barriers.
- We show that relaxing SC to TSO has a nontrivial impact on secure information flow (Section III). There exist programs that satisfy possibilistic noninterference under SC but not under TSO. Perhaps more surprisingly, there also exist programs that satisfy possibilistic noninterference under TSO but not under SC.
- We adapt an existing type system for possibilistic noninterference under SC [19] to our formal language, augmenting it to support both dynamic thread creation and lock-based synchronization (Section IV). The resulting type system, $\vdash^{\mathrm{sc}}$, is of independent interest.
- While sound under SC semantics, it turns out that $\vdash^{\mathrm{sc}}$ does not ensure possibilistic noninterference under TSO. We describe a simple modification to $\vdash^{\mathrm{sc}}$ that is sound for TSO (Section V). However, the resulting type system, $\vdash^{\mathrm{tso}}$ disallows concurrency-related constructs from appearing in high-security contexts.
- We show how to resolve the expressiveness limitation of $\vdash^{\mathrm{tso}}$ by refining it to track a security level for each thread's write buffer (Section VI). The resulting $\vdash^{\mathrm{wb}}$ type system includes a simple form of flow sensitivity for this purpose.
- Finally, we show that the $\vdash^{\mathrm{sc}}$ type system is in fact sound under TSO for programs that are *data-race-free* (Section VII). Since it is considered good programming practice to write race-free programs, this result provides an alternate approach to guaranteeing secure information flow on top of weak memory models like TSO.

Figure 3 summarizes the results described above. It also shows that our three type systems have a natural ordering: typability of a command c in $\vdash^{\mathrm{tso}}$ implies typability in $\vdash^{\mathrm{wb}}$, which in turn implies typability in $\vdash^{\mathrm{sc}}$. All three type systems therefore ensure possibilistic noninterference under SC, and $\vdash^{\mathrm{tso}}$ and $\vdash^{\mathrm{wb}}$ additionally ensure possibilistic noninterference under TSO. Full formal development and proofs of the theorems described in the paper can be found in a companion technical report [21].

II. A Formal Model of TSO Programs

A. Syntax

Figure 4 presents the syntax of the formal language that we use throughout the paper. A program is a command c, which consists of the usual imperative constructs including assignments, sequencing, conditionals, and while loops. It is convenient to distinguish between thread-local temporaries x and possibly-shared variables X. The three kinds of assignments respectively load a value from shared memory, perform local computation, and store a value into shared memory. Expression metavariables a and b respectively include the usual side-effect-free arithmetic and boolean operations, denoted $\oplus$ and $\ominus$.

We augment this imperative language with constructs for shared-memory concurrency. The command $(\mathbf{fork}\ c)$ forks a new thread that asynchronously executes command c, and $(\mathbf{sync}\ \ell\ \mathbf{do}\ c)$ provides a simple form of lock-based synchronization among threads. The **fence** command is a *memory barrier*, which can be used to enforce stronger semantics in the context of a weak memory model. Specifically for TSO, this command stalls execution of the current thread until all pending writes have been committed from the thread's write buffer. For example, inserting **fence**s after the commands $X := 1$ and $Y := 1$ in Figure 1 ensures that x and y cannot both end with value 0. The $(\mathbf{holding}\ \ell\ \mathbf{do}\ c)$ syntax is used only to properly implement the semantics of reentrant locks and may not appear in source programs.

B. Operational Semantics

Our operational semantics is defined as a binary relation on *execution states*, which is a pair of a global state G and a thread pool P (Figure 5). A global state G is a pair of a global store and the set of available locks. A thread pool P is a sequence of threads, with o denoting the empty thread pool. Each thread contains a local state L and a command to be executed. A local state includes a local store, the locks currently held by this thread, and the thread's write buffer. A write buffer is a sequence of pending writes to the global store, with nil denoting the empty write buffer.

Notation Suppose local state $L = (M, \lambda, W)$. We write $L.mem$, $L.locks$, and $L.wb$ for M, λ, and W respectively. If $x \in \mathbf{LocalVar}$ we write $L[x \mapsto i]$ for $(M[x \mapsto i], \lambda, W)$ and $L(x)$ for $M(x)$. We use $L \cup \lambda'$ and $L \setminus \lambda'$ and $\ell \in L$ for $(M, \lambda \cup \lambda', W)$ and $(M, \lambda \setminus \lambda', W)$ and $\ell \in \lambda$. We use the analogous notations to manipulate the global store and lockset within a global state G. We additionally use $L{+\!\!+}(X := i)$ for

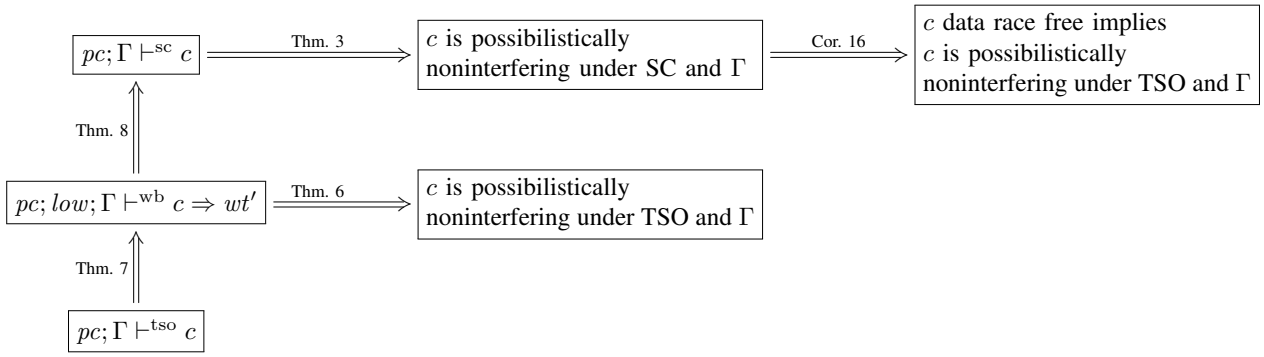

Fig. 3. A summary of the paper's key results. A program is represented as a command c, and Γ is a security policy mapping variables to security labels. The typing judgments are described in detail later in the paper.

$$
\begin{array}{rcl}
c & ::= & x := X \mid x := a \mid X := x \\
& \mid & \textbf{skip} \mid c_1; c_2 \\
& \mid & \textbf{if } b \textbf{ do } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c \\
& \mid & \textbf{fork } c \mid \textbf{sync } \ell \textbf{ do } c \\
& \mid & \textbf{fence} \mid \textbf{holding } \ell \textbf{ do } c \\
a & ::= & x \mid i \mid a \oplus a \\
b & ::= & \textbf{true} \mid \textbf{false} \mid \textbf{isZero } a \mid b \otimes b
\end{array}
$$

Fig. 4. The syntax of our formal language. Metavariable x ranges over a set $\mathbf{LocalVar}$ of thread-local variables, X over a set $\mathbf{HeapVar}$ of global variables, ℓ over a set $\mathbf{Lock}$ of locks, and i over integer literals.

$$
\begin{array}{lrcl}
\text{Global state} & G & ::= & S \times \lambda \\
\text{Global store} & S & \in & \mathbf{HeapVar} \to \mathbb{Z} \\
\text{Lockset} & \lambda & \in & \mathcal{P}(\mathbf{Lock}) \\
\text{Thread pool} & P, Q & ::= & \mathsf{o} \mid t \parallel P \\
\text{Thread} & t & ::= & \langle L, c \rangle \\
\text{Local state} & L & ::= & M \times \lambda \times W \\
\text{Local store} & M & \in & \mathbf{LocalVar} \to \mathbb{Z} \\
\text{Write buffer} & W & ::= & nil \mid (X := i) :: W
\end{array}
$$

Fig. 5. Domains for the operational semantics.

$(M, \lambda, W +\!\!+ (X := i) :: nil)$, where $+\!\!+$ denotes list append, and $(X := i) :: L$ for $(M, \lambda, (X := i) :: W)$. Finally, when there is no possibility of confusion we use a single thread t to denote the corresponding singleton thread pool, $t \parallel \mathsf{o}$.

Figure 6 provides the rules for taking one step of execution on a thread, producing a new global state and zero or more residual threads. We distinguish between two kinds of step operations and annotate each step accordingly. A *commit* operation commits the pending write at the head of the thread's write buffer to the global store. An *eval* operation performs one computation step on the thread's current command.

Most of the *eval* steps are standard. We highlight the most interesting ones. Rule EC-STORE simply adds the write to the end of the thread's write buffer. Rule EC-LOAD uses the auxiliary judgment shown in Figure 7 to lookup the value of a shared variable X. That judgment implements the semantics of store-to-load forwarding: if there is a pending write to X in the thread's write buffer, then the value of the most recent one is returned; otherwise, the value of X is fetched from the global store.

Rule EC-FENCE acts as a no-op but its premise has the effect of forcing computation on this thread to stall until all pending writes have committed. Forking a thread (EC-FORK) as well as acquiring (EC-SYNCACQUIRE) and releasing (EC-HOLDRELEASE) a lock also require an empty write buffer, which accords with the typical semantics of these constructs. As shown in rule EC-SYNCREENTER, our locks are reentrant; the **fence** semantics of lock acquire and release are still enforced in that case. Symbol $L_{\oslash}$ in EC-FORK represents the "empty" local state $((\lambda x.0), \emptyset, nil)$. Finally, the thread may be terminated by rule EC-REAP once its command is **skip**, it has committed all pending writes, and it has released all locks.

Figure 8 shows the rules for stepping an execution state, under both the TSO and SC memory models. Under TSO, a thread is chosen nondeterministically for execution, and that thread can perform either a *commit* or *eval* operation (ranged over by metavariable *op*). The SC memory model is a special case of TSO whereby a *commit* operation is always scheduled for execution if one is enabled. This semantics has the effect of forcing a write to be committed to main memory as soon as it is added to the write buffer.

C. Possibilistic Noninterference

Now we can define the standard notion of possibilistic noninterference [19] for programs in our formalism. Let a *security level* be either *low* or *high* and a *security context* Γ be a function from shared variables, local variables, and locks to security levels. We define security levels as a lattice with partial order $\sqsubseteq$, least upper bound $\sqcup$, and greatest lower bound $\sqcap$. These operators respect the ordering $low \sqsubseteq high$. It is straightforward to allow an arbitrary lattice of security levels rather than just two [6].

$(G, t) \longrightarrow^{commit} (G', P)$

$$\frac{}{(G, \langle (X := i) :: L, c \rangle) \longrightarrow^{commit} (G[X \mapsto i], \langle L, c \rangle)}$$

$(G, t) \longrightarrow^{eval} (G', P)$

$$\frac{}{(G, \langle L, X := x \rangle) \longrightarrow^{eval} (G, \langle L +\!\!+ (X := L(x)), \mathbf{skip} \rangle)}\ \textsc{ec-Store}$$

$$\frac{(G.mem; L.wb)[X] \Downarrow i}{(G, \langle L, x := X \rangle) \longrightarrow^{eval} (G, \langle L[x \mapsto i], \mathbf{skip} \rangle)}\ \textsc{ec-Load} \qquad \frac{L[a] \Downarrow i}{(G, \langle L, x := a \rangle) \longrightarrow^{eval} (G, \langle L[x \mapsto i], \mathbf{skip} \rangle)}\ \textsc{ec-EvalExp}$$

$$\frac{(G, \langle L, c_1 \rangle) \longrightarrow^{eval} (G', \langle L', c_1' \rangle \parallel P)}{(G, \langle L, c_1; c_2 \rangle) \longrightarrow^{eval} (G', \langle L', c_1'; c_2 \rangle \parallel P)}\ \textsc{ec-SeqStruct} \qquad \frac{}{(G, \langle L, \mathbf{skip}; c \rangle) \longrightarrow^{eval} (G, \langle L, c \rangle)}\ \textsc{ec-SeqSkip}$$

$$\frac{L[b] \Downarrow \mathbf{true}}{(G, \langle L, \mathbf{if}\ b\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2 \rangle) \longrightarrow^{eval} (G, \langle L, c_1 \rangle)}\ \textsc{ec-IfTrue} \qquad \frac{L[b] \Downarrow \mathbf{false}}{(G, \langle L, \mathbf{if}\ b\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2 \rangle) \longrightarrow^{eval} (G, \langle L, c_2 \rangle)}\ \textsc{ec-IfFalse}$$

$$\frac{L[b] \Downarrow \mathbf{true}}{(G, \langle L, \mathbf{while}\ b\ \mathbf{do}\ c \rangle) \longrightarrow^{eval} (G, \langle L, c; \mathbf{while}\ b\ \mathbf{do}\ c \rangle)}\ \textsc{ec-WhileTrue}$$

$$\frac{L[b] \Downarrow \mathbf{false}}{(G, \langle L, \mathbf{while}\ b\ \mathbf{do}\ c \rangle) \longrightarrow^{eval} (G, \langle L, \mathbf{skip} \rangle)}\ \textsc{ec-WhileFalse}$$

$$\frac{L.wb = nil}{(G, \langle L, \mathbf{fork}\ c \rangle) \longrightarrow^{eval} (G, \langle L, \mathbf{skip} \rangle \parallel \langle L_\oslash, c \rangle)}\ \textsc{ec-Fork} \qquad \frac{L.wb = nil}{(G, \langle L, \mathbf{fence} \rangle) \longrightarrow^{eval} (G, \langle L, \mathbf{skip} \rangle)}\ \textsc{ec-Fence}$$

$$\frac{\ell \in G \qquad L.wb = nil}{(G, \langle L, \mathbf{sync}\ \ell\ \mathbf{do}\ c \rangle) \longrightarrow^{eval} (G \setminus \{\ell\}, \langle L \cup \{\ell\}, \mathbf{holding}\ \ell\ \mathbf{do}\ c \rangle)}\ \textsc{ec-SyncAcquire}$$

$$\frac{\ell \in L}{(G, \langle L, \mathbf{sync}\ \ell\ \mathbf{do}\ c \rangle) \longrightarrow^{eval} (G, \langle L, \mathbf{fence}; c; \mathbf{fence} \rangle)}\ \textsc{ec-SyncReenter}$$

$$\frac{\ell \in L \qquad (G, \langle L, c \rangle) \longrightarrow^{eval} (G', \langle L', c' \rangle \parallel P)}{(G, \langle L, \mathbf{holding}\ \ell\ \mathbf{do}\ c \rangle) \longrightarrow^{eval} (G', \langle L', \mathbf{holding}\ \ell\ \mathbf{do}\ c' \rangle \parallel P)}\ \textsc{ec-HoldStep}$$

$$\frac{\ell \in L \qquad L.wb = nil}{(G, \langle L, \mathbf{holding}\ \ell\ \mathbf{do\ skip} \rangle) \longrightarrow^{eval} (G \cup \{\ell\}, \langle L \setminus \{\ell\}, \mathbf{skip} \rangle)}\ \textsc{ec-HoldRelease}$$

$$\frac{L.wb = nil \qquad L.locks = \emptyset}{(G, \langle L, \mathbf{skip} \rangle) \longrightarrow^{eval} (G, \mathfrak{o})}\ \textsc{ec-Reap}$$

Fig. 6. Rules for taking one step of execution on a thread. We omit the rules for standard judgments $L[a] \Downarrow i$ and $L[b] \Downarrow v$, which evaluate arithmetic and boolean expressions.

$$\boxed{(S;W)[X] \Downarrow i}$$

$$\frac{}{(S;W{+\!\!+}(X := i))[X] \Downarrow i} \qquad \frac{X \neq Y \qquad (S;W)[X] \Downarrow i}{(S;W{+\!\!+}(Y := i_0))[X] \Downarrow i}$$

$$\frac{}{(S;nil)[X] \Downarrow S(X)}$$

Fig. 7. Rules to lookup the value of a shared variable.

$$\boxed{(G,P) \Longrightarrow^{\mathsf{tso}} (G',P')}$$

$$\frac{P = t_1 \ldots t_{i-1} \parallel t_i \parallel t_{i+1} \ldots t_n \qquad (G,t_i) \longrightarrow^{op} (G',Q)}{(G,P) \Longrightarrow^{\mathsf{tso}} (G', t_1 \ldots t_{i-1} \parallel Q \parallel t_{i+1} \ldots t_n)}$$

$$\boxed{(G,P) \Longrightarrow^{\mathsf{sc}} (G',P')}$$

$$\frac{P = t_1 \ldots t_{i-1} \parallel t_i \parallel t_{i+1} \ldots t_n \quad (G,t_i) \longrightarrow^{commit} (G',Q)}{(G,P) \Longrightarrow^{\mathsf{sc}} (G', t_1 \ldots t_{i-1} \parallel Q \parallel t_{i+1} \ldots t_n)}$$

$$\frac{P = t_1 \ldots t_{i-1} \parallel t_i \parallel t_{i+1} \ldots t_n \qquad (G,t_i) \longrightarrow^{eval} (G',Q) \quad \text{there is no } t_j \text{ such that } (G,t_j) \longrightarrow^{commit} (G_j,Q_j)}{(G,P) \Longrightarrow^{\mathsf{sc}} (G', t_1 \ldots t_{i-1} \parallel Q \parallel t_{i+1} \ldots t_n)}$$

Fig. 8. Program evaluation under both TSO and SC memory models.

Definition 1 (Low equivalence)**.** *Given a security context* Γ*, we say that global store* S *is* low-equivalent *to global store* S'*, denoted* $S \sim_\Gamma S'$*, if for all shared variables* X*, it is the case that* $\Gamma(X) = low$ *implies* $S(X) = S'(X)$*.*

Let $\Longrightarrow^{\mathsf{mm}*}$ be the reflexive, transitive closure of the $\Longrightarrow^{\mathsf{mm}}$ relation, where mm is either tso or sc.

Definition 2 (Possibilistic noninterference)**.** *We say that command* c *is* possibilistically noninterfering *(or* possibilistically secure*) under memory model* mm *and policy* Γ *if for all* S_1, S_2 *such that* $S_1 \sim_\Gamma S_2$*, if* $((S_1, \mathbf{Lock}), \langle L_\oslash, c\rangle) \Longrightarrow^{\mathsf{mm}*} (G_1', \mathsf{o})$ *then there exists* G_2' *such that* $((S_2, \mathbf{Lock}), \langle L_\oslash, c\rangle) \Longrightarrow^{\mathsf{mm}*} (G_2', \mathsf{o})$ *and* $G_1'.mem \sim_\Gamma G_2'.mem$*.*

III. TSO and Possibilistic Noninterference

In this section we show that possibilistic security under SC is incomparable to possibilistic security under TSO. That is, there exists a command that is possibilistically secure under SC but not under TSO, and there also exists a command that is possibilistically secure under TSO but not under SC.

Figure 9 shows a program that is possibilistically secure under SC but not under TSO. The portion of this program before the conditional is identical to the program in Figure 1, except that a shared variable Y′ is used to transfer the value of y from the forked thread to the main thread. Suppose that

```
X := 0;
Y := 0;
Y′ := 1;
fork (X := 1;
      y := Y; Y′ := y)
Y := 1;
x := X;
y′ := Y′;
if (isZero x ∧ isZero y′)
  do (h := H; L := h);
else skip
```

Fig. 9. A program that is possibilistically secure under SC but not under TSO.

```
X := 0;
Y := 0;
Y′ := 1;
fork (X := 1;
      y := Y; Y′ := y)
Y := 1;
x := X;
y′ := Y′;
(if (isZero x ∧ isZero y′)
  do
    if isZero H do L := 1 else L := 0
 else
    if isZero H do L := 0 else L := 1);
```

Fig. 10. A program that is possibilistically secure under TSO but not under SC.

Γ maps all shared variables to security level $high$ except for L, which is mapped to low.

Under SC, at least one of x and y′ will be nonzero. Therefore the conditional guard will always fail, so L is never updated by the program. Since L is the only low variable, it is easy to see that the program is possibilistically secure.

On the other hand, as we've seen, under TSO it is possible for both x and y′ to have the value 0. Therefore, on some executions the conditional block will execute, copying the value of H to L. Consider the following global stores:

$$S_1 = \{(X,0),(Y,0),(Y',0),(L,0),(H,1)\}$$

$$S_2 = \{(X,0),(Y,0),(Y',0),(L,0),(H,0)\}$$

Clearly S_1 and S_2 are low-equivalent under Γ. However, there is an execution starting from S_1 that ends with L = 1, while all executions starting from S_2 end with L = 0.

Figure 10 shows a program that is possibilistically secure under TSO but not under SC.[2] The portion before the conditional is identical to the portion before the conditional in Figure 9. Again suppose that Γ maps all shared variables to security level $high$ except for L, which is mapped to low.

Under SC, at least one of x and y′ will be nonzero. Therefore the conditional guard will always fail, and the result is that L

[2]For clarity our examples sometimes directly reference a shared variable in a loop or conditional guard, rather than first loading the variable into a temporary.

always ends with value 0 if H is 0 and ends with value 1 if H is nonzero. Hence the program is not possibilistically secure, and the same stores S_1 and S_2 above serve as a counterexample.

On the other hand, under TSO it is possible for the conditional guard to evaluate to **true** or **false** regardless of the initial global store, depending on how threads are scheduled. Therefore it is always possible for L to end with either the value 0 or 1, so no information is leaked from H to L and the program is possibilistically secure.

The example programs in Figures 9 and 10 use simple mechanisms to leak high information and would be rejected by standard information-flow type systems. The following sections describe more subtle information flows due to concurrency in general as well as TSO specifically.

IV. Possibilistic Noninterference for SC

In this section we adapt an existing type system for possibilistic noninterference of SC programs [19] to our formal language. This involves extending that type system to handle dynamic thread creation via **fork** as well as lock-based synchronization. The type system is shown in Figure 11.

The rules for the sequential fragment of the language are standard except for the extra restrictions on loops in rule SC-WHILE. First, the loop guard cannot depend on $high$ data [19]. This restriction prevents $high$ data from affecting a program's termination, which can violate possibilistic noninterference as illustrated in the following program, where Γ maps L to low and H to $high$:

```
L := 1;
while (isZero H) do skip
```

More subtly, the type system also must prevent loops from occurring in $high$ contexts [19]. This is illustrated in the program in Figure 12, where Γ maps X and L to low and H to $high$. In this program, the final value of L records whether or not H has the value 0. It is possible to allow more permissive typing for $high$ loops [18] by tracking additional information, but such extensions are orthogonal to our goal of investigating relaxed memory models.

Our language's concurrency constructs have no analogue in the language of Smith and Volpano [19], which supports neither dynamic thread creation nor any form of synchronization. The main novelty is the treatment of synchronization. The security policy Γ provides a security level for each lock. Similar to the treatment of conditionals and loops, if a $high$ lock is acquired then the body of the critical section must type as $high$. Furthermore, the rules prevent a low lock from being acquired in a $high$ context. These restrictions on locks rule out programs where synchronization allows $high$ data to influence whether or not a program terminates. Figure 13 shows a program which terminates when H is 1 but runs forever when H is 0. This program does not satisfy possibilistic noninterference assuming Γ maps H to $high$, and it properly fails to typecheck in $\vdash^{\text{sc}}$ in that case: $\Gamma(\ell)$ must be low in order to type the nested while loop, but $\Gamma(\ell)$ must be $high$ in order to acquire the lock from a $high$ context in the forked thread. These rules for structured locks are less restrictive than those proposed by Sabelfeld [15] for semaphores, a lower-level concurrency construct that he requires be low-typed.

We have proven that well-typed programs are secure under SC:

Theorem 3. *If $pc; \Gamma \vdash^{\text{sc}} c$, then c is possibilistically noninterfering under SC and Γ.*

V. Possibilistic Noninterference for TSO

Unfortunately, the $\vdash^{\text{sc}}$ type system does not ensure possibilistic noninterference under TSO. The key problem is that the concurrency constructs all have the effect of flushing a thread's write buffer. Therefore, employing concurrency within a $high$ context can leak information to low variables by forcing low writes to be committed.

Figure 14 illustrates a simple example of the problem. The program is identical to the one in Figure 1 except that it copies the values of x and y to shared variables X′ and Y′ and it conditionally includes two **fence** instructions. The program typechecks under $\vdash^{\text{sc}}$ assuming all variables have security level low except for H. However, the program is not possibilistically secure under TSO. As we've seen, under TSO it is possible for both X′ and Y′ to end with value 0. However, this is not possible when H has the value 0, since in that case the program executes sufficient **fence** instructions to ensure that the result is sequentially consistent. As a result, if an execution does end with both X′ and Y′ having the value 0, we have leaked the fact that H is nonzero.

Figure 15 shows a type system that resolves this problem. The rules for the sequential fragment are identical to those in Figure 11. The rules for the concurrency constructs (**fence**, **sync**, and **fork**) specialize those in Figure 11 by forbidding such constructs from appearing in $high$ contexts. This additional restriction makes the program in Figure 14 ill-typed. This restriction also makes it unnecessary to track security levels for locks. Indeed, we could soundly replace the **sync** rule with the following revised version that ignores the lock's security level:

$$\frac{pc; \Gamma \vdash^{\text{tso}} c}{low; \Gamma \vdash^{\text{tso}} \mathbf{sync}\ \ell\ \mathbf{do}\ c}$$

We use the slightly more complex rule in order to maintain uniformity with the paper's other type systems; there is no loss of expressiveness.

We have proven that well-typed programs in $\vdash^{\text{tso}}$ are secure under TSO:

Theorem 4. *If $pc; \Gamma \vdash^{\text{tso}} c$ then c is possibilistically noninterfering under TSO and Γ.*

Furthermore, Theorems 7 and 8 described in Section VI below imply the following property relating $\vdash^{\text{tso}}$ to $\vdash^{\text{sc}}$:

Corollary 5. *If $pc; \Gamma \vdash^{\text{tso}} c$ then $pc; \Gamma \vdash^{\text{sc}} c$.*

Together with Theorem 3 this means that the $\vdash^{\text{tso}}$ type system also ensures possibilistic noninterference under SC.

$\boxed{\Gamma \vdash a : \tau}$

$$\frac{\Gamma(x) \sqsubseteq \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash i : \tau} \qquad \frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1 \oplus a_2 : \tau}$$

$\boxed{\Gamma \vdash b : \tau}$

$$\frac{}{\Gamma \vdash \mathbf{true} : \tau} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \tau} \qquad \frac{\Gamma \vdash a : \tau}{\Gamma \vdash \mathbf{isZero}\ a : \tau} \qquad \frac{\Gamma \vdash b_1 : \tau \qquad \Gamma \vdash b_2 : \tau}{\Gamma \vdash b_1 \otimes b_2 : \tau}$$

$\boxed{pc; \Gamma \vdash^{\text{sc}} c}$

$$\frac{pc \sqcup \Gamma(Y) \sqsubseteq \Gamma(x)}{pc; \Gamma \vdash^{\text{sc}} x := Y}\ \text{SC-LOAD} \qquad \frac{pc \sqcup \Gamma(y) \sqsubseteq \Gamma(X)}{pc; \Gamma \vdash^{\text{sc}} X := y}\ \text{SC-STORE} \qquad \frac{\Gamma \vdash a : \tau \qquad pc \sqcup \tau \sqsubseteq \Gamma(x)}{pc; \Gamma \vdash^{\text{sc}} x := a}\ \text{SC-EVAL}$$

$$\frac{pc; \Gamma \vdash^{\text{sc}} c_1 \qquad pc; \Gamma \vdash^{\text{sc}} c_2}{pc; \Gamma \vdash^{\text{sc}} c_1;\ c_2}\ \text{SC-SEQ} \qquad \frac{\Gamma \vdash b : \tau \qquad pc \sqcup \tau; \Gamma \vdash^{\text{sc}} c_1 \qquad pc \sqcup \tau; \Gamma \vdash^{\text{sc}} c_2}{pc; \Gamma \vdash^{\text{sc}} \mathbf{if}\ b\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2}\ \text{SC-IF}$$

$$\frac{\Gamma \vdash b : low \qquad pc; \Gamma \vdash^{\text{sc}} c}{low; \Gamma \vdash^{\text{sc}} \mathbf{while}\ b\ \mathbf{do}\ c}\ \text{SC-WHILE} \qquad \frac{}{pc; \Gamma \vdash^{\text{sc}} \mathbf{skip}}\ \text{SC-SKIP} \qquad \frac{pc \sqsubseteq \Gamma(\ell) \qquad \Gamma(\ell); \Gamma \vdash^{\text{sc}} c}{pc; \Gamma \vdash^{\text{sc}} \mathbf{sync}\ \ell\ \mathbf{do}\ c}\ \text{SC-SYNC}$$

$$\frac{}{pc; \Gamma \vdash^{\text{sc}} \mathbf{fence}}\ \text{SC-FENCE} \qquad \frac{pc; \Gamma \vdash^{\text{sc}} c}{pc; \Gamma \vdash^{\text{sc}} \mathbf{fork}\ c}\ \text{SC-FORK}$$

Fig. 11. A type system for possibilistic noninterference of SC programs.

```
X := 0;
fork (
  if (isZero H) do
    while(isZero X) do skip
  else skip;
  L := 0; X := 1
);
if !(isZero H) do
  while (isZero X) do skip;
else skip;
L := 1; X := 1
```

Fig. 12. Leaking information through *high* while loops.

```
sync ℓ do (
  S := 0;
  fork ( (if (isZero H) do
            sync ℓ do skip
          else
            skip);
         S := 1);
  while (isZero S) do skip
)
```

Fig. 13. A program which is not possibilistically secure (under TSO or SC) due to unrestricted use of synchronization.

```
X := 0;
Y := 0;
fork (X := 1;
      if (isZero H) do fence else skip;
      y := Y; Y′ := y);
Y := 1;
if (isZero H) do fence else skip;
x := X; X′ := x
```

Fig. 14. An example illustrating why fences and other concurrency constructs cannot occur in *high* contexts.

While our new type system is sound under TSO, it does not allow locks to be acquired nor threads forked from within a *high* context. In the next section we show how to safely relax these restrictions while preserving possibilistic noninterference.

VI. Typing the Write Buffer for More Expressiveness

As we saw in the previous section, employing concurrency in a *high* context can leak information via the write buffer. However, we observe that if the write buffer only contains pending writes to *high* variables at the point where such concurrency occurs, then no leakage is possible. We have designed a new type system based on this observation. In

$\boxed{pc; \Gamma \vdash^{\text{tso}} c}$

$$\frac{pc \sqcup \Gamma(Y) \sqsubseteq \Gamma(x)}{pc; \Gamma \vdash^{\text{tso}} x := Y}\ \textsc{tso-Load} \qquad \frac{pc \sqcup \Gamma(y) \sqsubseteq \Gamma(X)}{pc; \Gamma \vdash^{\text{tso}} X := y}\ \textsc{tso-Store} \qquad \frac{\Gamma \vdash a : \tau \qquad pc \sqcup \tau \sqsubseteq \Gamma(x)}{pc; \Gamma \vdash^{\text{tso}} x := a}\ \textsc{tso-Eval}$$

$$\frac{pc; \Gamma \vdash^{\text{tso}} c_1 \qquad pc; \Gamma \vdash^{\text{tso}} c_2}{pc; \Gamma \vdash^{\text{tso}} c_1; c_2}\ \textsc{tso-Seq} \qquad \frac{\Gamma \vdash b : \tau \qquad pc \sqcup \tau; \Gamma \vdash^{\text{tso}} c_1 \qquad pc \sqcup \tau; \Gamma \vdash^{\text{tso}} c_2}{pc; \Gamma \vdash^{\text{tso}} \textbf{if } b \textbf{ do } c_1 \textbf{ else } c_2}\ \textsc{tso-If}$$

$$\frac{\Gamma \vdash b : low \qquad pc; \Gamma \vdash^{\text{tso}} c}{low; \Gamma \vdash^{\text{tso}} \textbf{while } b \textbf{ do } c}\ \textsc{tso-While} \qquad \frac{}{pc; \Gamma \vdash^{\text{tso}} \textbf{skip}}\ \textsc{tso-Skip} \qquad \frac{\Gamma(\ell); \Gamma \vdash^{\text{tso}} c}{low; \Gamma \vdash^{\text{tso}} \textbf{sync } \ell \textbf{ do } c}\ \textsc{tso-Sync}$$

$$\frac{}{low; \Gamma \vdash^{\text{tso}} \textbf{fence}}\ \textsc{tso-Fence} \qquad \frac{pc; \Gamma \vdash^{\text{tso}} c}{low; \Gamma \vdash^{\text{tso}} \textbf{fork } c}\ \textsc{tso-Fork}$$

Fig. 15. A type system for possibilistic noninterference of TSO programs.

addition to tracking the security level of the program counter as usual, the type system also tracks the security level of each thread's write buffer: *high* indicates that all entries in the write buffer are writes to *high* variables, and *low* indicates that the write buffer may contain writes to *low* variables. Tracking the write buffer's contents in this way requires our type system to incorporate a form of flow sensitivity.

The rules for our new type system are shown in Figure 16. The judgment $pc; wt; \Gamma \vdash^{\text{wb}} c \Rightarrow wt'$ includes a write buffer typing wt as an extra assumption and "produces" a new write buffer typing wt' that takes into account the possible effects of the command c on the write buffer. The write buffer typing is threaded through the typing of a command, as illustrated by the rule WB-SEQ.

The most important rule in the sequential fragment is WB-STORE, which ensures that the produced write buffer typing wt' reflects the security level of the variable X. In particular, if $\Gamma(X) = low$ then $wt' = low$. The rule for conditionals conservatively takes the meet of the write buffer typings resulting from the two branches. The rule for loops is similar, except that—as is common in flow sensitive analyses—the loop body's output wt' must also be incorporated into its input write buffer typing.

Typing for each of the concurrency constructs requires $pc \sqsubseteq wt$, which captures our earlier informal observation. Namely, the type system allows concurrency in a *high* context as long as the write buffer is guaranteed to only contain writes to *high* variables. The concurrency constructs all produce a write buffer typing of *high* to reflect the fact that they empty the thread's write buffer. For a similar reason it is safe to typecheck the body of a **sync** and a **fork** under a *high* write buffer typing.

The $\vdash^{\text{wb}}$ type system properly rejects the program in Figure 14 shown earlier. The two occurrences of **fence** fail to typecheck because the write buffer typing at each of those points is *low*, due to the preceding writes to X and Y.

At the same time, the type system safely supports several useful idioms that involve concurrency in *high* contexts. For example, in the following well-typed code sketch a group of cooperating threads are forked to perform some *high* computation whenever a password check succeeds. Additionally, the number of password checks is recorded in a *low* global variable.

```
Checks := 0;
password := Password;
fence;
while(true) do (

  guess := ... ; // Read a password guess
  if(guess = password) do (
      fork c1;
      fork c2;
      fork c3
  ) else
      skip;

  Checks := Checks + 1;
  fence
)
```

"Worker threads" c_1, c_2, and c_3 may safely acquire *high* locks, fork new *high* threads, and use the **fence** instruction to run a concurrent protocol. These threads may not write *low* variables or acquire *low* locks because they are spawned in a *high* context, that is, within a conditional that branches on the *high* password variable. The **fork** instructions occur in positions where the type system can verify that the write buffer does not contain *low* writes; the two **fence** instructions are used to establish and maintain this invariant.

We have proven that well-typed programs in this type system are secure under TSO:

Theorem 6. *If $pc; wt; \Gamma \vdash^{\text{wb}} c \Rightarrow wt'$ then c is possibilistically noninterfering under TSO and Γ.*

Our three type systems have a natural ordering in terms of expressiveness:

$pc; wt; \Gamma \vdash^{\mathrm{wb}} c \Rightarrow wt$

$$\frac{pc \sqcup \Gamma(Y) \sqsubseteq \Gamma(x)}{pc; wt; \Gamma \vdash^{\mathrm{wb}} x := Y \Rightarrow wt}\ \textsc{wb-Load} \qquad \frac{pc \sqcup \Gamma(y) \sqsubseteq \Gamma(X)}{pc; wt; \Gamma \vdash^{\mathrm{wb}} X := y \Rightarrow wt \sqcap \Gamma(X)}\ \textsc{wb-Store}$$

$$\frac{\Gamma \vdash a : \tau \qquad pc \sqcup \tau \sqsubseteq \Gamma(x)}{pc; wt; \Gamma \vdash^{\mathrm{wb}} x := a \Rightarrow wt}\ \textsc{wb-Eval} \qquad \frac{pc; wt; \Gamma \vdash^{\mathrm{wb}} c_1 \Rightarrow wt_1 \qquad pc; wt_1; \Gamma \vdash^{\mathrm{wb}} c_2 \Rightarrow wt_2}{pc; wt; \Gamma \vdash^{\mathrm{wb}} c_1; c_2 \Rightarrow wt_2}\ \textsc{wb-Seq}$$

$$\frac{\Gamma \vdash b : \tau \qquad pc \sqcup \tau; wt; \Gamma \vdash^{\mathrm{wb}} c_1 \Rightarrow wt_1 \qquad pc \sqcup \tau; wt; \Gamma \vdash^{\mathrm{wb}} c_2 \Rightarrow wt_2}{pc; wt; \Gamma \vdash^{\mathrm{wb}} \textbf{if } b \textbf{ do } c_1 \textbf{ else } c_2 \Rightarrow wt_1 \sqcap wt_2}\ \textsc{wb-If}$$

$$\frac{\Gamma \vdash b : low \qquad low; wt \sqcap wt'; \Gamma \vdash^{\mathrm{wb}} c \Rightarrow wt'}{low; wt; \Gamma \vdash^{\mathrm{wb}} \textbf{while } b \textbf{ do } c \Rightarrow wt \sqcap wt'}\ \textsc{wb-While} \qquad \frac{}{pc; wt; \Gamma \vdash^{\mathrm{wb}} \textbf{skip} \Rightarrow wt}\ \textsc{wb-Skip}$$

$$\frac{pc \sqsubseteq \Gamma(\ell) \qquad pc \sqsubseteq wt \qquad \Gamma(\ell); high; \Gamma \vdash^{\mathrm{wb}} c \Rightarrow wt'}{pc; wt; \Gamma \vdash^{\mathrm{wb}} \textbf{sync } \ell \textbf{ do } c \Rightarrow high}\ \textsc{wb-Sync} \qquad \frac{pc \sqsubseteq wt}{pc; wt; \Gamma \vdash^{\mathrm{wb}} \textbf{fence} \Rightarrow high}\ \textsc{wb-Fence}$$

$$\frac{pc; high; \Gamma \vdash^{\mathrm{wb}} c \Rightarrow wt' \qquad pc \sqsubseteq wt}{pc; wt; \Gamma \vdash^{\mathrm{wb}} \textbf{fork } c \Rightarrow high}\ \textsc{wb-Fork}$$

Fig. 16. More precise security typing for TSO programs.

Theorem 7. *If $pc; \Gamma \vdash^{\mathrm{tso}} c$ then there exists wt' such that $pc; low; \Gamma \vdash^{\mathrm{wb}} c \Rightarrow wt'$.*

Theorem 8. *If $pc; wt; \Gamma \vdash^{\mathrm{wb}} c \Rightarrow wt'$ then $pc; \Gamma \vdash^{\mathrm{sc}} c$.*

The $\vdash^{\mathrm{sc}}$ type system in Figure 11 is the most expressive of the three (i.e., it accepts the most programs) but is also the only one that does not guarantee possibilistic security under TSO. The last theorem above combined with Theorem 3 implies that the $\vdash^{\mathrm{wb}}$ type system ensures possibilistic noninterference under SC, just as the other two type systems do.

VII. Data Race Freedom and Possibilistic Noninterference

It is considered good programming practice to properly synchronize concurrent accesses to shared data, thereby avoiding *data races*. In this section we consider the impact of data-race freedom on secure information flow for concurrent programs.

Intuitively, a data race occurs when two threads are about to access the same shared variable, where at least one access is a write. The following definitions formalize this intuition for our formal language.

Definition 9 (Reads Next)**.** *Thread $\langle L, c\rangle$* reads *X* next *if one of the following conditions holds:*

- *c has the form $x := X$*
- *c has the form $c_1; c_2$ and $\langle L, c_1\rangle$ reads X next*
- *c has the form $\textbf{holding } \ell \textbf{ do } c'$ and $\ell \in L$ and $\langle L, c'\rangle$ reads X next*

Definition 10 (Writes Next)**.** *Thread $\langle L, c\rangle$* writes *X* next *if one of the following conditions holds:*

- *c has the form $X := x$*
- *c has the form $c_1; c_2$ and $\langle L, c_1\rangle$ writes X next*
- *c has the form $\textbf{holding } \ell \textbf{ do } c'$ and $\ell \in L$ and $\langle L, c'\rangle$ writes X next*

Definition 11 (Accesses Next)**.** *Thread t* accesses *X* next *if either t reads X next or t writes X next.*

Definition 12 (Conflicting Threads)**.** *Threads s and t* conflict *if there exists a variable X such that each thread accesses X next and at least one thread writes X next.*

Definition 13 (Race-Exhibiting Thread Pool)**.** *Thread pool P* exhibits a race *if it contains two distinct threads that conflict.*

A program is considered race-free if it cannot reach a race-exhibiting thread pool on any SC execution [12]:

Definition 14 (Race-Free Command)**.** *Command c is defined to be* race-free *if for all S, G', and P' such that $((S, \mathbf{Lock}), \langle L_\oslash, c\rangle) \Longrightarrow^{\mathsf{sc}*} (G', P')$, it is not the case that P' exhibits a race.*

As a (non-)example, the program in Figure 1 is not race-free, because there exists an SC execution in which the write to X in the forked thread conflicts with the read of X in the main thread.

It is well known that race-free programs do not exhibit any more behaviors under TSO than they do under SC [12], and this property holds in our formal language:

Theorem 15. *If $((S, \mathbf{Lock}), \langle L_\oslash, c\rangle) \Longrightarrow^{\mathsf{tso}*} (G', \mathsf{o})$ and c is race-free, then $((S, \mathbf{Lock}), \langle L_\oslash, c\rangle) \Longrightarrow^{\mathsf{sc}*} (G', \mathsf{o})$.*

This result, combined with the fact that SC executions are a subset of TSO executions, implies that our original type system for possibilistic noninterference under SC is sound under TSO as well, as long as such programs are data-race-free:

Corollary 16. *If c is race-free and c is possibilistically noninterfering under SC and Γ then c is possibilistically noninterfering under TSO and Γ.*

Corollary 17. *If c is race-free and $pc; \Gamma \vdash^{\mathrm{sc}} c$, then c is possibilistically noninterfering under TSO and Γ.*

The above corollary provides an alternate way to ensure secure information flow for programs running on weak memory models like TSO. Rather than designing dedicated type systems for such memory models, we can typecheck the program in a type system for secure information flow under SC, such as our $\vdash^{\mathrm{sc}}$ system, and separately check the program for data races. This approach is appealing because it reduces the problem to two problems that have existing solutions, and it avoids the need to reason about weak memory models. On the other hand, dedicated type systems for weak memory models can be more expressive. For example, our $\vdash^{\mathrm{tso}}$ and $\vdash^{\mathrm{wb}}$ type systems safely allow some programs that contain data races.

VIII. Related Work

To the best of our knowledge this is the first paper to address information-flow security for concurrency in the presence of weak memory models.

Smith and Volpano [19] introduced the use of type systems to ensure possibilistic noninterference of concurrent programs with SC semantics. Their setting includes a fixed set of threads with sequentially-consistent access to shared memory. Our $\vdash^{\mathrm{sc}}$ type system starts from this work and extends it to support synchronization (**sync**), thread creation (**fork**), and memory barriers (**fence**).

Possibilistically noninterfering programs are vulnerable to attacks based on timing, statistical inference, scheduling, and termination. For instance, an attacker who knows how threads are likely be scheduled may have an advantage in guessing a concurrent process's secret inputs based on its public outputs. Broadly these attacks are based on the ability of an attacker to resolve nondeterminism essential to the specification of a program, a language model, a schedule, or a security statement.

Probabilistic noninterference addresses these issues by ensuring that $high$ inputs to a program do not change the probability distribution describing its low outputs. Probabilistic noninterference is a very strong property and difficult to enforce in a practical manner. While some enforcement techniques are purely type-based [18], others also mix static typing with new runtime features such as atomicity instructions [22] or special compilation strategies that ensure $high$-data-dependent program paths are not distinguished by scheduling [16].

Observationally deterministic concurrency [14, 20, 24] ensures that all runs of a concurrent program look identical to low observers. Zdancewic and Myers [24] present a lambda calculus with both message-passing and sequentially-consistent shared-memory concurrency and introduce a type system that ensures observational determinism. Observational determinism is substantially more restrictive than the notion of data-race-freedom that we employ in Section VII, but it in turn provides stronger security guarantees.

As mentioned above, possibilistically noninterfering programs are vulnerable to timing attacks—where an attacker makes inference about confidential values based on the execution time of a run—and termination attacks—where an attacker makes inferences based on a program's termination behavior. Effective enforcement of timing and termination sensitive variants of noninterference is an open area of research. Some techniques include statically padding the branches of $high$-data-dependent conditionals with no-op instructions to mask (internally or externally) observable differences in instruction count [3, 15, 16], statically tracking timing information using information flow labels as an approximation mechanism [18], and partitioning programs so that $high$ components may be treated specially by a security-aware scheduler [8].

This paper analyzes an idealized version of the TSO memory model, with a simple small-step semantics that captures the essence of TSO and that is comparable to language models commonly used to investigate information-flow type systems [15, 16, 18, 19, 22]. Earlier formalisms of TSO have different goals, including comparing multiple relaxed memory models [7], supporting program verification [5], and accurately describing extant hardware platforms [13].

IX. Conclusions and Future Work

This paper has investigated the impact of the Total Store Order (TSO) memory model on secure information flow. We have shown that relaxing SC has a nontrivial impact on the notion of possibilistic noninterference and that it causes a natural security type system to become unsound. We provided two alternative type systems that are sound under TSO, with different tradeoffs between expressiveness and complexity. We also proved that the original type system is sound under TSO for programs that are free of data races.

Given the ubiquity of weak memory models in mainstream multicore hardware and concurrent programming languages, there is much more to be done in future work. We would like to consider common memory models other than TSO, for example hardware memory models like ARM and POWER [17] and language-level memory models like those of Java [10] and C++ [4]. We would also like to address more detailed architectural models such as those supporting full instruction sets [13] or with realistic bounds on hardware resources. Finally, we would like to consider stronger notions of security in the concurrent setting, such as probabilistic noninterference [16, 18, 22] and notions that take into account timing channels [3, 8, 15, 16, 18].

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53:90–101, August 2010.

[3] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[4] H. J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of PLDI*, pages 68–78. ACM, 2008.

[5] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 107–120. Springer Berlin / Heidelberg, 2008.

[6] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5): 236–243, May 1976.

[7] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *In Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, 1997.

[8] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 413–428, Washington, DC, USA, 2011. IEEE Computer Society.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 100(28):690–691, 1979.

[10] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proceedings of POPL*, pages 378–391. ACM, 2005.

[11] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an sc-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 199–210. ACM, 2011.

[12] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP 2010 - Object-Oriented Programming*, pages 478–503. 2010.

[13] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs '09: Conference on Theorem Proving in Higher Order Logics, volume 5674 of LNCS*, pages 391–407. Springer, 2009.

[14] A. W. Roscoe. CSP and determinism in security modelling. In *In Proc. IEEE Symposium on Security and Privacy*, pages 114–127. Society Press, 1995.

[15] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proceedings of the Andrei Ershov 4th International Conference on Perspectives of System Informatics*, volume 2244, pages 225–239. Springer-Verlag, July 2001.

[16] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. of 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society, July 2000.

[17] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186. ACM, 2011.

[18] Geoffrey Smith. A new type system for secure information flow. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 115–125. IEEE, June 2001.

[19] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364. ACM, 1998.

[20] Tachio Terauchi. A type system for observational determinism. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300, Washington, DC, USA, 2008. IEEE Computer Society.

[21] Jeffrey A. Vaughan and Todd Millstein. Secure information flow for concurrent programs under total store order: Supplemental technical material. Technical Report 120007, Computer Science Department, University of California, Los Angeles, April 2012. Available from http://fmdb.cs.ucla.edu/Treports/120007.pdf.

[22] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *J. Comput. Secur.*, 7:231–253, March 1999.

[23] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 43–54. ACM, 2011.

[24] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. of 16th IEEE Computer Security Foundations Workshop*, July 2003.