Abstract—Practical enforcement of secure information flow in concurrent programs remains notoriously difficult. An underexplored reason is that concurrent programs rely on a wide variety of synchronization primitives, which can leak information. However, the existing literature on security for concurrent programs has focused on a small set of such primitives.

This paper studies how channel-like synchronization primitives can leak information and describes enforcement mechanisms to prevent such leaks. Out of this study, we provide two technical contributions. First, we introduce a simple, novel analysis that bans memory races and channel contention that otherwise lead to timing channels. This analysis repurposes security labels for information flow control to be used also as capabilities for accessing resources. Compared with the fractional capabilities technique used by prior work, our analysis supports a significantly simpler inference procedure at a small expense in expressivity. Second, we generalize the security definition of observational determinism as first proposed by Zdancewic and Myers [24] to allow for a limited amount of observable nondeterminism while ensuring this cannot leak information. This security condition is designed for a synchronization primitive whose behavior is inherently nondeterministic. We integrate these technical contributions into a type system and soundness result for a concurrent language with dynamically created threads.

I. INTRODUCTION

Ensuring secure information flow for concurrent programs remains a challenging problem. Perhaps the most important reason why concurrency creates new ways for timing channels to leak information. Some timing channels are already present in sequential programs—external timing channels in which the adversary times execution outside the system. However, concurrency also leads to internal timing channels—where the interaction of threads with the scheduler becomes a potentially high-bandwidth information channel that can be exploited to leak information. Preventing leaks through internal timing channels while allowing for external timing channels is particularly important because it allows for expressive programs and can be proved secure without assumptions about scheduler behavior. Zdancewic and Myers [24] proposed observational determinism as a security condition for this setting: to close internal timing channels, programs need only to appear deterministic to adversaries. To achieve observational determinism, programs must satisfy a strong form of race freedom.

Another challenge for ensuring security in the concurrent setting is that synchronization mechanisms are used to coordinate the execution of concurrent threads, and these mechanisms can leak information. While some existing work on security for concurrent programs deals with such synchronization [2, 4, 12, 14, 21, 22, 24], concurrent languages support a wide variety of synchronization primitives beyond those studied previously. From this literature one can glean a piecemeal understanding of how synchronization primitives leak information, but the general picture remains unclear.

In this paper, we aim to initiate the systematic study of how synchronization primitives leak information and of enforcement mechanisms to prevent such leaks. In particular, we focus on the use of various channel-like primitives, and different methods of receiving from such channels. Much of the observations in this study is folklore or scattered throughout the literature, but we hope a systematization will highlight the general patterns of how such primitives leak information and the general methods for closing these leaks. Because we study programs that use expressive synchronization primitives, we focus on closing internal timing channels.

We make two main technical contributions. First, we introduce a novel analysis to prevent memory races and channel contention with a much simpler inference algorithm than the fractional capabilities approach used in previous work [14, 22] to ensure programs are race-free. The key idea is that the standard type-based technique for ensuring secure information flow can be repurposed as a capability mechanism to ban both racing accesses to memory locations and contention over channel communication. While it trades the expressiveness of fractional capabilities for simplicity of inference, we observe that they are adequate for a particular kind of channel.

Second, we generalize observational determinism and define a new security condition that allows for a limited amount of observable nondeterminism that cannot leak information. We found the necessity of this new condition from observing that a particular kind of synchronization primitive is designed explicitly to not be deterministic.

We make the following contributions:

• A study of how various channel-like synchronization primitives can leak information, and a discussion of enforcement mechanisms to prevent such leaks.
A simple, novel analysis that bans racing accesses to memory locations and contention over channels. It trades the expressiveness of fractional capabilities for simplicity of its inference procedure, which can be unified with a standard information flow analysis.

A new security condition that relaxes observational determinism to allow a limited amount of observable nondeterminism that is guaranteed to not leak information.

A type system and a soundness result integrating our new security condition and our novel analysis for banning races and contention.

II. INFORMATION FLOW IN CONCURRENT LANGUAGES

Our threat model addresses internal timing channels [24], which arise when adversaries make observations by executing processes concurrent with the program, subject to whatever constraints are placed on a process to be considered well-formed. We assume the adversary can observe some updates to low memory and the order in which these updates are performed, but cannot observe the wall-clock timing of these updates. To control external timing channels requires a different set of techniques and stronger guarantees from the execution semantics (e.g., [1, 21, 25]). Hence, the adversary also cannot directly observe program termination—it cannot use wall-clock timing to discriminate between a diverging program and a program that is delayed for a very long time.

A. Races and internal timing channels

Concurrency can create subtle information leaks arising from internal timing channels. Consider the following program written in a simple while-language extended with a construct spawn that spawns a new thread. Here, variable hi is secret and lo holds public data:

```
spawn {
  if hi then delay(100) else skip;
  lo := true
};
delay(20); lo := false
```

There are no direct flows from hi to lo, nor are there indirect leaks from control flow. However, there is still a leak: when hi is true, the spawned thread is delayed before it writes to lo, making the write likely to be executed after the write to lo in the main thread. Thus, when hi is true, the final value of lo is most likely true as well. But when hi is false, the spawned thread will not be delayed, so its write likely occurs before the write of the main thread. Thus when hi is true, the final value of lo is likely false as well. Since the final value of lo is correlated with hi, the program is insecure.

The crux of the problem is that the spawned thread and the main thread perform racing accesses to data—in particular, their writes to lo race, allowing execution time to influence the order of updates. So the value of hi becomes encoded in the order of writes to lo. Thus, in order to ensure programs in the language are secure, we must not only control the flow of information among data, but also ensure that threads do not perform side effects that interfere with other threads.

Unordered writes are not the only kind of race that violates security. In this program, a read–write race leaks secrets:

```
spawn {
  if hi == false then delay(100) else skip;
  lo1 := true
};
delay(20); lo2 := ! lo1
```

When hi is true, the spawned thread’s write to lo1 will likely occur before the main thread’s read of lo1; when hi is false, the delay causes the write to likely occur after the read. The value of lo2 is correlated with the value of hi, so this program is also insecure.

B. Synchronization

Threads can use synchronization primitives to coordinate with each other. While there are other forms of synchronization such as monitors, we study channel-like primitives as these are widely used and have variations that can leak information in subtle ways. We focus on three different kinds of channels:

- **Buffered channels.** A thread can use a buffered channel to send a value to another thread. As its name implies, the channel buffers values on a queue: sending a value enqueues it, while a thread receiving a value dequeues from the buffer. Sending threads never block, while receiving threads block until there is the buffer queue is nonempty.

- **Rendezvous channels.** Rendezvous channels are like buffered channels but do not buffer values, so a thread sending a value blocks until some thread is ready for it.

- **Semaphores.** A thread can use a semaphore to send a signal to another thread. Signals are “buffered”: sending a signal increments an internal counter in the semaphore, while waiting for a signal decrements the counter. Threads sending signals are never blocked, but the semaphore blocks threads waiting for signals, to keep the counter nonnegative at all times. Semaphores can also be seen as “unit-typed channels,” where the signals are unit values.

For all of these types of channels, a thread can send a value or a signal e1 over channel e2 through the statement form send e1 to e2 (for semaphore e it is send e). To wait on a signal or receive a value over a channel, threads can use the following statement forms. (Again the receive forms for semaphores are slightly different, as they do not bind received values to variables.)

- **recv e(x){s},** where the value received from channel e is bound to x and in scope in s. This statement blocks until there is a signal or value on the channel.

- **join (e1,e2)(x1,x2){s},** where the value received from channel e1 is bound to x1 and in scope in s. This statement blocks until there are signals or values on both
channels. This receive form is essentially a join pattern, as introduced by Fournet and Gonthier [8].

- select $e_1(x_1)|s_1 \parallel e_2(x_2)|s_2$, where the value received from $e_i$ is bound to $x_i$ and in scope in $s_i$. The statement blocks until one of the channels receives a signal or value, after which the statement will reduce to either $s_1$ or $s_2$. This receive form is essentially select as seen in Concurrent ML [19] and the Go programming language [10].

Synchronization can be used to avoid racing access to data. Consider the program below.

spawn ( lo := false; send c );
recv c; lo := true;

Both the spawned thread and the main thread write to lo, but these writes do not leak information because synchronization over channel c ensures that the writes are always performed in the same order.

At the same time, however, synchronization can introduce new ways of leaking information. In general, there are three main ways that channels can leak information: (1) through the progress of a thread, (2) through contention on a channel, and (3) through contention on a receive site. Figure 1 summarizes the ways information can leak for different channels and receive forms.

C. Leaks through thread progress

Because threads receiving on a channel block until they have received a signal or a value, their progress can depend on information at the sending context of a channel. The sending thread use secrets to choose on which channel to send, and thus the receiver can learn information about the secret from the mere fact of receiving. Consider the program below.

spawn { recv c1; lo := true };  
spawn { recv c2; lo := false };  
if hi then send c1 else send c2

Since the decision to send to either channel c1 or c2 depends on the secret value hi, it can influence the progress of the spawned threads, and thus their respective writes to lo leak the value of hi.

Note that for rendezvous channels, leaks through progress are bidirectional: information about the receiver can be leaked to the sender. This is because, unlike semaphores or buffered channels, a sender must block until a receiver waits on a channel, and thus its progress can depend on information at the receiving context. For example, in the program below, the receiver can branch on a secret value to determine from which channel to receive, and thus the value of lo correlates with hi.

spawn ( send true to c1; lo := true );  
spawn ( send true to c2; lo := false );  
if hi then recv c1(x1) { ... }  
else recv c2(x2) { ... }

For join statements, the progress of one sender depends on the other. In the program below, neither the sending nor receiving contexts for channels c1 and c4 are secret, but their receive sites block for senders at secret contexts, so it still leaks the value of hi to lo.

spawn ( send c1; lo := true );  
spawn ( send c4; lo := false );  
spawn ( join(c1,c2)(x1,x2){ ... } );  
spawn ( join(c3,c4)(x3,x4){ ... } );  
if hi send c2 else send c3

For select statements, the progress of a sender is influenced by the scheduling decision to execute one branch or another. Thus there can be a progress leak if this decision is influenced by secrets. Consider the program below, where the decision to pad delays—which influences the scheduler’s choice of which select branch to execute—depends on a secret value hi, thus allowing lo to correlate with it.

spawn (  
  if hi then delay(1000) else skip;  
  send c1(true); l1 := false  
);  
spawn (  
  delay(100); send c2(true); l1 := true  
);  
select c1(x1){ ... } || c2(x2){ ... }

D. Leaks through contention

Whereas racing accesses to memory locations allow scheduling decisions to leak information, contention along a channel or a receive site can also leak information. Contention allows observations about the relative timing between threads, information about which can be encoded in the order contention sends and receives are executed. We will first focus on contention along a channel.

1) Leaks through contention on a channel: Consider the following example of receive contention.

spawn ( // thread 1  
  if hi then delay(1000) else skip;  
  recv c; lo1 := x  
);  
spawn ( // thread 2  
  delay(100); recv c; lo2 := x  
);  
send c; send c

At thread 1, the secret value hi determines whether to pad delays before receiving from channel c. This influences whether the receive site at thread 1 or thread 2 receives from c first: if hi is true, then thread 1 is delayed and thus receives from c after thread 2. This allows value of lo1 to correlate with hi, so the program leaks information.

Naturally, the question of whether information leaks occur through send contention follows. The answer is yes—but unlike receive contention, where the leak occurs in the scheduling decision to which receive site the value or signal gets sent, the observation of relative timing between threads is encoded in the value sent along the channel. This is the case because to exploit send contention, the receive site must be able to discriminate which thread sent along the channel, and that is only possible by discriminating the value sent. Consider the program below.
<table>
<thead>
<tr>
<th>Synchronization primitive</th>
<th>Can leak information by...</th>
<th>Receive method</th>
<th>Can leak information by...</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore</td>
<td>receive contention</td>
<td>receive</td>
<td>progress</td>
</tr>
<tr>
<td>buffered channel</td>
<td>send and receive contention</td>
<td>join</td>
<td>progress</td>
</tr>
<tr>
<td>rendezvous channel</td>
<td>send and receive contention; progress leak is bidirectional</td>
<td>select</td>
<td>progress, receive site contention</td>
</tr>
</tbody>
</table>

Fig. 1: Channel types, receive methods, and the ways they can leak information.

```plaintext
spawn { // thread 1
  if hi then delay(1000) else skip;
  send false to c
}
spawn { // thread 2
  delay(100); send true to c;
}
recv c(x){ lo1 := x }; recv c(y){ lo2 := y }
```

This is similar to the receive contention example, but importantly the sends to channel c on thread 1 and thread 2 need to send a value in order to leak information. The value of hi correlates with the first value sent along c, allowing it to leak to lo1.

An immediate consequence of the observation that values must be sent to distinguish different senders is that send contention for semaphores cannot leak information. Again, one can think of signals as unit values and semaphores as unit-typed channels. Since the unit type is uniquely inhabited, it is impossible to encode information about the sender in the signal—thus on the receive site for a semaphore, it is impossible to discriminate which thread sent the signal. This is borne out in Terauchi [22] and Karbyshev et al. [14], which both allow send contention along semaphores.

2) Leaks through contention on a receive site: By contention along a receive site, we mean multiple channels sending a value to a single receive site. The scheduling decision to allow one channel or another to send first to the receive site can encode information about relative timing of the sending threads, but whether this can be exploited depends on the receive form. Clearly this kind of contention is impossible if the receive form in the site is a `recv`, since only one channel can send to the site. The `join` receive form likewise cannot exploit contention to leak information, even though multiple channels can be sending to its receive site. This is because a `join` statement blocks until it receives a value from both channels, and thus the receive site cannot discriminate on which channel a value was first sent.

On the other hand, because a `select` statement unblocks when it first receives a value from either channel, it can discriminate on which channel a value was first sent. Consider the program below:

```plaintext
spawn {
  if hi then delay(1000) else skip;
  send false to c2
};
spawn { delay(100); send true to c1 }
select c1(x1){ lo := x1 }
|| c2(x2){ lo := x2 }
```

The value of lo is correlated with hi, so this program leaks information.

### III. Enforcement Mechanisms for Preventing Information Leaks

Having explored the ways in which synchronization can leak information, we now discuss enforcement mechanisms to prevent such leaks. We discuss these mechanisms informally in this section—they are formally defined later in a type system (Section VI).

**Track influence on thread progress:** An intuitive rule to ban progress leaks is to ensure that the receiving context must be at least as secret as the sending context, so that the receiving thread cannot learn anything about the sending context that it should not be able to. This restriction means that the PC label of the sending context must flow to the PC label of the receiving context. As it turns out, this restriction is inadequate—consider the program below.

```plaintext
spawn { // thread 1
  if hi then recv c1 else recv c1;
  lo := true
}
spawn { // thread 2
  if hi then recv c2 else recv c2;
  lo := false
}
send c1 if hi else c2
```

The rule above marks the program secure, but there is a leak: if hi is true, then thread 1 becomes unblocked and writes to lo so that it is true as well; a similar situation occurs when hi is false, and thus the value of lo correlates with hi. Because the leak is propagated through the progress of the receiving thread, the threads can leak information at any point after they have received.

A progress label [2, 4] can give an upper bound to the information that has influenced the progress of a thread up to that program point. Like the PC label standard in information flow control type systems, this label acts as a lower bound on the memory locations that can be safely updated. We also add a label on channel types to give an upper bound on the PC label of a channel’s sending or receiving contexts. By ensuring this label flows to the progress label after a receive (and after a send for rendezvous channels), we ensure that information from the sending context cannot leak by the receiving thread’s progress.
Ban data races and contention: We can ban memory races and channel contention to prevent information encoded in scheduling decisions from being leaked. Some prior work [14, 22] used fractional capabilities (also known as fractional permissions), an expressive type discipline introduced by Boyland [6], to ban data races and contention. As its name implies, fractional capabilities allow threads to own fractions of capabilities to access resources. For example, a thread may only write to a memory location if it has the full capability to write to that location, ensuring that no other thread can access the location concurrently. If a thread only reads from a location, however, it can share the capability to do so with other threads. A thread must have full capability to send or receive over a channel to prevent other threads from doing the same. Concretely, a thread’s capability to access a resource is represented as a rational number between 0 and 1; at every program point, the capability for a resource across all live threads sums to 1. Synchronization allows capability values to be transferred between threads.

Although fractional capabilities are expressive, prior work has observed some limitation. Fractional capabilities force programmers to annotate declarations with arbitrary rational numbers [5]. Some papers [11, 17] have attempted to ameliorate this by implementing fractional-style analyses that hide the actual fractions from the programmer, but inference algorithms for type systems using fractional capabilities are complicated. For example, the type system by Terauchi [22] defends observational determinism using fractional capabilities. The inference procedure for the type system has three phases and requires solving separate constraint systems for aliasing (unification constraints), information flow (lattice constraints), and fractional capabilities (linear inequality constraints).

We make the observation that the full expressive power of fractional capabilities is only useful for semaphores and buffered channels. Buffering allows these channels to take advantage of the fact that fractional capabilities can “count” and thus buffer capabilities to be transferred to other threads. For example, in the race-free program below, the spawned threads each have capability 1/2 for \( c \), which they transfer to the main thread using semaphore \( c \). The main thread can receive two signals from \( c \) to recover the full capability for \( x \), and thus can subsequently write to it.

```plaintext
spawn \( \{ y := !x; \text{send} \ c \} \);
spawn \( \{ z := !x; \text{send} \ c \} \);
recv \ c \; \text{recv} \ c \; x := \text{false}
```

For a language with rendezvous channels, in Section VI we introduce a novel analysis for banning races and contention with a much simpler inference procedure than that of Terauchi’s for fractional capabilities. Instead of representing capabilities to access data or channels as rational numbers, our analysis represents them as lattice elements. Inference for this capability analysis can easily be integrated with inference for a standard information flow analysis (Section VI-D).

Track influence on scheduler: Banning contention prevents synchronization from leaking information. However, banning all contention makes the use of some synchronization constructs impossible. Specifically, it makes no sense to ban contention for `select` statements by ensuring only one channel at a time can send to such a receive site—`select` statements are designed exactly to allow such contention. To allow using `select`, instead of banning contention at receive sites we instead ensure that the information encoded in scheduling decisions cannot be leaked by the branches of `select`.

To allow races on low data without leaking secrets, prior work [9, 14] make the simplifying assumption that data can only influence scheduling decisions by influencing control flow. The insecure examples discussed above all satisfy this assumption, as they use secret information to determine when to pad delays into the execution of threads. Races on low data are secure and cannot leak secrets so long as secrets have not influenced control flow up to that program point.

We can adapt techniques to allow such low races to allow the secure use of `select`. We introduce a scheduler label that acts as an upper bound on the information that has influenced control flow within any thread up to that program point. We then ensure that the scheduler label lower-bounds the possible updates in the branches of a `select` statement, meaning that the information encoded in the scheduler’s decision to execute one branch over another cannot be leaked.

We allow the scheduler label to be raised or lowered through the new primitives lower_schedule and raise_schedule respectively. Following prior work that allows downgrading the influence on the scheduler [4, 14], lower_schedule acts as a barrier where the thread calling the primitive waits for all other threads to finish execution (i.e., their bodies are skip). Once all live threads have passed through the barrier, scheduler state can be reset, erasing the influence of secret information.

IV. A SIMPLE CONCURRENT LANGUAGE

To explore our approach more formally, we use a simple concurrent language whose abstract syntax is shown in Figure 2. The language has booleans as primitive data, and provides operations over these (\( \otimes \)). The language contains first-class references, which can be allocated with an initial value (newref \( \ell \) \( e \)), written (\( e_1 := e_2 \)), or read (\( \ell e \)). References are tagged with a label \( \ell \) denoting security level of the data it contains. Allocated references evaluate to heap locations (loc). Existing threads can spawn new ones (spawn \( \{ s \} \)). The language also has standard let-binding forms (let \( x = e \) in \( s \)) and control structures (if \( e \) then \( s_1 \) else \( s_2 \), while \( e \) do \( s \)).

The language also supports allocation of first-class rendezvous channels (newchan \( e, r \)). We limit the language to use rendezvous channels because we aim to prove the soundness of the Interactive Lambda Calculus [15] (ILC) has a similar construct to `select` called `ch` (choice). In well-typed ILC programs it is guaranteed that only one process at a time will send messages (is “activated”), and thus there can be no contention at receive sites using `ch`. ILC has this unique metatheoretical property because its design goal is to capture the execution model of the Universal Composability framework [7] for proving the security of cryptographic protocols. Since this property is not desirable for general programming, we make no such restrictions for our language and thus cannot guarantee the absence of contention at receive sites using `select`.2

2The Interactive Lambda Calculus [15] (ILC) has a similar construct to `select` called `ch` (choice). In well-typed ILC programs it is guaranteed that only one process at a time will send messages (is “activated”), and thus there can be no contention at receive sites using `ch`. ILC has this unique metatheoretical property because its design goal is to capture the execution model of the Universal Composability framework [7] for proving the security of cryptographic protocols. Since this property is not desirable for general programming, we make no such restrictions for our language and thus cannot guarantee the absence of contention at receive sites using `select`.2
of our novel capability analysis in a type system (Section VI). Adding semaphores and buffered channels to the language and switching out the capability analysis to use fractional capabilities in the type system is straightforward.

The language includes receive forms \texttt{recv} and \texttt{join} as they are described above. For the \texttt{select} receive form, two annotations are added: $\xi$ uniquely identifies the \texttt{select} within the program text, while security label $\ell$ marks the influence on scheduling where the \texttt{select} is executed.

The operational semantics, defined in Figure 3, is a largely standard small-step reduction relation over configurations $(S,H,\Delta)$, where $H : \text{Loc} \rightarrow \text{Val}$ is a shared memory heap and $\Delta : \text{Tid} \rightarrow \text{Stmt}$ is a thread pool mapping thread IDs to the current statement being executed by the thread. The reduction relation is stratified into a “global” relation $(S,H,\Delta) \rightarrow (S',H',\Delta')$ that defines execution of the entire thread pool and a “local” relation $(H,\Delta) \rightarrow_{\ell} (H',\Delta')$ that defines execution of a single thread. A rule translates a local step for a single thread to a global step for the thread pool. The semantics is nondeterministic to capture all possible scheduling decisions.

Component $S$ is a scheduler for select statements. It is a function $\Xi \rightarrow \text{Sched} \times \{\leftarrow,\rightarrow\}$ from select identifiers (drawn from set $\Xi$) to a pair containing a new scheduler and either $\leftarrow$ or $\rightarrow$, which denotes the decision to execute the left or right branches of the select respectively. We assume that scheduler decisions between identifiers are independent: given that $\pi_i$ projects the $i$-th component of a tuple and select identifiers $\xi$ and $\xi'$ where $\xi \neq \xi'$, $\pi_i(S(\xi))(\xi') = S(\xi')$.

Schedulers are not meant to faithfully model the actual scheduling of threads. Rather, they are a means to resolve nondeterminism in the choice of which branch of a select statement to execute. They provide a minimal contract—namely, our “narrowness” assumption—for the actual concurrency implementation to follow in order to prevent insecure information flows. We use them to formulate our security condition (Section V).

V. SECURITY CONDITION

A. OBSERVATIONAL DETERMINISM

The security condition observational determinism (henceforth OD) as proposed by Zdancewicz and Myers [24] most closely captures the requirements needed for programs to be secure in our threat model. Intuitively, a program is observationally deterministic if the sequence of updates to low memory locations is invariant to scheduling decisions and initial values of high memory locations. The Zdancewicz–Myers definition of OD is distinguished from previous formulations by Roscoe [20] and McLean [16] by being “pointwise” and only considering equality of traces for individual locations: only updates to the same low location need to be deterministic, while updates to different low locations can be reordered. The definition is also termination-insensitive, in that sequences of updates to low locations need only be prefixes of each other across two different runs to satisfy OD. This security condition admits expressive programs that update low memory concurrently, and can be defended with a typing discipline.

Huisman et al. [13] have pointed out that the combination of being “pointwise” and allowing location traces to be prefixes of each other admits insecure programs that leak more than a bit, as is claimed in the presence of a termination channel. They construct a program that satisfies OD but leaks the value of a secret number. In response, they propose a stronger version of OD that requires location traces across two different runs to be stutter-equivalent, not just prefixes of each other.

As has been pointed out [3, 14, 22], however, defending such a security condition is hard, particularly if one is interested in using a typing discipline without draconian restrictions. Askarov et al. [3] also prove that the brute-force approach used in Huisman et al.’s counterexample, which increments a low variable to the value of the secret, is the best an adversary can do to exploit termination-insensitive conditions if it can observe intermediate states (like location traces).

Terauchi [22] responds to Huisman et al.’s counterexample by giving up the “pointwise” property instead of prefixing: that is, they give a security condition that requires updates to be deterministic across different low memory locations. But this cuts against the original motivation of the Zdancewicz–Myers definition, as it forces the restriction that only one thread at a time can modify low memory.

So there are several security conditions at play here, each with their own set of trade-offs. Because our focus on this work is the study of how the use of synchronization creates internal timing channels, in a sense the choice doesn’t matter. We give a security condition that is parametric between the Zdancewicz–Myers definition and Terauchi’s definition of OD, as these are both defendable using type systems. The security

---

1However, we note that their counterexample relies on excluding certain low (adversarial) observations from the traces considered. See Appendix A for details.
Evaluation contexts  
\[ E ::= [], E \circ \ell \mid v \circ E \mid !E \mid newref_\ell E \mid newchan_{\ell,r} \mid \text{let } x = E \text{ in } s \mid E := e \mid v := E \mid \text{send } E \text{ to } e \mid \text{send } v \text{ to } E \mid \text{recv } E(x)\{s\} \mid \text{join } (E,e_2)(x_1,x_2)\{s\} \mid \text{join } (v_1,E)(x_1,x_2)\{s\} \mid \text{select}_{\ell,E} E(x_1)\{s_1\} \parallel e_2(x_2)\{s_1\} \mid \text{select}_{\ell,E} v(x_1)\{s_1\} \parallel E(x_2)\{s_1\} \]

\[ \langle H, \Delta \rangle \rightarrow \langle H', \Delta' \rangle \]
\[ \langle H, s \rangle \rightarrow_L \langle H', s' \rangle \]
\[ (H, e) \rightarrow_L \langle H', e' \rangle \]
\[ \text{loc} \not\in \text{dom}(H) \]
\[ \langle H, \text{newref}_\ell e \rangle \rightarrow_L (H[\text{loc} \mapsto v], \text{loc}) \]
\[ \langle H, s \rangle \rightarrow_L (S, H[\text{loc} \mapsto \text{chan}], \text{loc}) \]

\[ \langle H, v_1 \otimes v_2 \rangle \rightarrow_L \langle H, \otimes(v_1, v_2) \rangle \]
\[ \langle H, \text{loc} \rangle \rightarrow_L \langle H, \text{loc} \rangle \]
\[ \langle H, \text{loc} := v \rangle \rightarrow_L \langle H[\text{loc} \mapsto v], \text{skip} \rangle \]
\[ \langle H, \text{if true then } s_1 \text{ else } s_2 \rangle \rightarrow_L \langle H, s_1 \rangle \]
\[ \langle H, \text{while } e \text{ do } s \rangle \rightarrow_L \langle H, \text{if then } (s; \text{while } e \text{ do } s) \text{ else skip} \rangle \]

\[ \langle H, e \rangle \rightarrow_L \langle H', e' \rangle \]
\[ \langle H, s' \rangle \rightarrow_L \langle H', E[s'] \rangle \]
\[ \langle H, \Delta[\text{tid}] \rangle \rightarrow_L \langle H', s' \rangle \]

\[ \Delta[\text{tid}] = E[\text{spawn } \{s\}], \text{tid'} \not\in \text{dom}(H) \]
\[ \langle S, H, \Delta \rangle \rightarrow \langle S, H, \Delta[\text{tid} \mapsto E[\text{skip}]][\text{tid'} \mapsto s] \rangle \]

\[ H = H'[\text{loc} \mapsto \text{chan}] \]
\[ \Delta = \Delta'[\text{tid} \mapsto E[\text{send } v \text{ to } \text{loc}]][\text{tid'} \mapsto E'[\text{recv } \text{loc}(x)\{s\}]] \]
\[ \langle S, H, \Delta \rangle \rightarrow \langle S, H, \Delta'[\text{tid} \mapsto E[\text{skip}]][\text{tid'} \mapsto E'[\text{loc} \{x/v\}]] \rangle \]

\[ H = H'[\text{loc}_1 \mapsto \text{chan}][\text{loc}_2 \mapsto \text{chan}] \]
\[ \Delta = \Delta'[\text{tid}_1 \mapsto E_1[\text{send } v_1 \text{ to } \text{loc}_1]][\text{tid}_2 \mapsto E_2[\text{send } v_2 \text{ to } \text{loc}_2]][\text{tid}_3 \mapsto E_3[\text{join } (\text{loc}_1, \text{loc}_2)(x_1, x_2)\{s\}]] \]
\[ \langle S, H, \Delta \rangle \rightarrow \langle S, H, \Delta'[\text{tid}_1 \mapsto E_1[\text{skip}]][\text{tid}_2 \mapsto E_2[\text{skip}]][\text{tid}_3 \mapsto E_3[s[x_1/v_1][x_2/v_2]] \rangle \]

\[ H = H'[\text{loc}_1 \mapsto \text{chan}] \]
\[ \Delta = \Delta'[\text{tid} \mapsto E[\text{send } v \text{ to } \text{loc}]][\text{tid'} \mapsto E'[\text{select}_{\ell,E} \text{loc}_1(x_1)\{s_1\} \parallel \text{loc}_2(x_2)\{s_2\}]] \]
\[ \langle S, H, \Delta \rangle \rightarrow \langle S', H, \Delta'[\text{tid} \mapsto E[\text{skip}]][\text{tid'} \mapsto E'[\text{select}_{\ell,E} \text{loc}_1(x_1)\{s_1\} \parallel \text{loc}_2(x_2)\{s_2\}]] \rangle \]
\[ \forall \text{tid'} \in \text{dom}(\Delta)[\text{tid} \mapsto \text{skip}] \]
\[ \Delta[\text{tid} \mapsto E[\text{raise}\_\text{schedule } \ell]] \]
\[ \langle S, H, \Delta \rangle \rightarrow \langle S, H, \text{tid} \mapsto E[\text{skip}] \rangle \]

Fig. 3: Operational semantics.

Condition partitions low memory and requires that the order of updates to different locations within the same equivalence class is deterministic. If the partition is induced by the “identity” relation that only relates locations to themselves, we get the original Zdancewic–Myers definition; if it is induced by the “all” relation that relates all locations to all locations, we get Terauchi’s definition. The type system we define in Section VI can be easily modified to defend either one of these conditions.

**B. Observational determinism modulo select**

That select is inherently non-deterministic because of contention is in tension with the definition of OD. Even with secure use of select, we can no longer guarantee programs “look” deterministic. Instead we generalize OD to a new security condition we call observational determinism modulo select (ODMS). Intuitively, a program satisfying ODMS must look deterministic to adversaries, except possibly for the scheduler’s choice to execute one branch of a select or another. We guarantee that the non-determinism observable by the
adversary could not have been influenced by secrets. This means that if a select is executed in a low-observable context but the scheduling choice of which branch is taken could have been influenced by secrets, the adversary should not be able to determine which branch is taken.

Before we formally define ODMS, some preliminaries. \( (S, H, \Delta) \Downarrow T \) denotes a sequence of reductions

\[
(S_1, H_1, \Delta_1) \rightarrow \cdots \rightarrow (S_n, H_n, \Delta_n)
\]

where \( (S, H, \Delta) = (S_1, H_1, \Delta_1) \) and \( T \) is a sequence of states such that \( T = [H_1, \ldots, H_n] \), \( H|_K \) denotes the restriction of a heap for a particular set of locations \( K \), while \( T|_K \) lifts heap restrictions into traces. Given sequences \( V_1 \) and \( V_2 \), \( V_1 \leq V_2 \) denotes \( V_1 \) is a prefix of \( V_2 \), up to stuttering. Given traces \( T_1 \) and \( T_2 \) and set of locations \( M \), \( T_1 \) is an \( M \)-prefix of \( T_2 \)—denoted as \( T_1 \leq M T_2 \)—when \( T_1|_M \leq T_2|_M \).

\( S_1 \sim S_2 \) denotes scheduler equivalence over a set of select identifiers \( I \):

\[
\forall \xi \in I. \pi_1(S_1(\xi)) \sim_I \pi_1(S_2(\xi)) \land \pi_2(S_1(\xi)) = \pi_2(S_2(\xi)).
\]

We can now formally define our new security condition.

**Definition 1 (Observational determinism modulo select):** Let \( s \) be a program with no free variables and set of free locations \( M \). Let \( ML \subseteq M \) be the set of \( \ell \)-visible memory locations and \( P \) be a partition of \( ML \). Let \( SL \) be a set of \( \ell \)-visible select identifiers, and let \( \Delta = [0 \mapsto s] \) be a thread pool consisting of only \( s \). Then \( s \) is observationally deterministic modulo select relative to \( P \) and \( SL \) when for any heaps \( H, H' \) and schedulers \( S_1, S_2 \) such that (1) \( ML \subseteq \text{dom}(H) \cap \text{dom}(H') \), (2) \( H|_{ML} = H'|_{ML} \), (3) \( S_1 \sim SL S_2 \), (4) \( (S_1, H, \Delta) \Downarrow T \), (5) \( \langle S_2, H', \Delta \rangle \Downarrow T' \), then for all \( p \in P \), either \( T \leq_p T' \) or \( T' \leq_p T \).

### VI. Type System

We combine the enforcement mechanisms described above into a type system that defends ODMS. The type system has both an information flow control analysis that ensures secrets do not flow to public values and a capability analysis that ensures resources such as memory locations and channels are used without races or contention.

Types are drawn from grammar

\[
\tau ::= \text{bool}_\ell | \text{ref}^\psi_\ell \tau | \text{chan}_\ell \Psi_1 \Psi_2 \tau
\]

and decorated with labels drawn from a bounded, distributive lattice \( L \). These labels can either represent security levels (\( \ell \)), capabilities (\( \psi \)) to access resources, or the resources themselves (\( \psi \)). Channel types are also decorated with label pairs (\( \Psi \)) that contain resources or capabilities used by the send (\( \Psi^s \)) and receive (\( \Psi^r \)) endpoints of a channel respectively. A function \( L(\tau) \) returns the security label \( \ell \) associated with type \( \tau \).

Threads are associated with a scheduler label \( (pc_\ell) \), bounding the assumed influence on scheduling decisions; a PC or control flow label \( (pc_\ell) \), bounding the influence over control flow; a progress label \( (pc_\ell) \), bounding the influence over the execution progress of a thread; and a capability label \( (\psi) \) that determines what resources the thread can access.

A global scheduler resource label \( (\psi_{\text{sched}}) \) coordinates the use of scheduler labels among threads. The scheduler resource ensures that threads’ scheduler labels are consistent, and allows the scheduler label to be lowered or raised.

### A. Capability Analysis

We now give intuition for how our novel capability analysis works. One can interpret capability and resource labels as denoting sets of resource tokens, which we call regions. The top element (\( \top \)) of the lattice represents an empty region, while the bottom element (\( \bot \)) represents the region containing all resource tokens. The join (\( \cup \)) and meet (\( \cap \)) lattice operations then coincide with region intersection and union respectively, while the flows-to relation (\( \Rightarrow \)) coincides with set inclusion.

The type system then defends the following invariants:

- **Resource labels denote non-empty regions.** Given resource label \( \psi \), this invariant is captured by \( \top \neq \psi \).
- **Capability labels for concurrent threads denote disjoint regions.** For concurrent threads with capability labels \( \psi_1 \) and \( \psi_2 \), this invariant is captured by \( \top = \psi_1 \cup \psi_2 \).
- **To have exclusive access to a resource, a thread’s capability label should denote a region that contains the resource label’s denoted region.** Given thread with label \( \psi_1 \) and resource with label \( \psi_r \), this invariant is captured by \( \psi_1 \subseteq \psi_r \).
- **To have shared access to a resource, a thread’s capability label should denote a region that overlaps with the resource label’s denoted region.** Given thread with label \( \psi_2 \) and resource with label \( \psi_r \), this invariant is captured by \( \top \neq \psi_2 \cup \psi_r \).

Defending all of these invariants allows the type system to prevent races on memory and contention over channels. For example, a write–write race for some memory location means that the resource label for the location (\( \psi \)) must be contained inside the regions of capability labels (\( \psi_1, \psi_2 \)) for two concurrent threads. Since \( \psi_1 \) and \( \psi_2 \) are disjoint, this implies that \( \psi \) is empty, violating the first invariant. Hence defending the invariants above prevents write–write races. On the other hand, if the concurrent threads are only reading the location, then this would imply that \( \psi \) overlaps both \( \psi_1 \) and \( \psi_2 \), which need not imply that \( \psi \) is empty. Hence concurrent reads are allowed while defending the invariants above.

### B. Typing Rules

There are four judgment forms in the type system:

- \( \vdash \tau \) denotes a well-formedness judgment over types. These judgments ensure that resource labels are non-empty.
- \( \tau_1 \leq \tau_2 \) denotes a subtyping judgment between two expression types. Subtyping is covariant over security labels—types can be upcasted to use higher (more restrictive) security labels—and invariant over resource and capability labels.
Fig. 4: Type well-formedness, subtyping, and expression typing.

Fig. 5: Statement typing for receive forms.
\[ \Gamma; pc_s; pc_t; \psi \vdash e : \tau \] denotes a well-typedness judgment for expressions.

\[ \Gamma; pc_s; pc_t; \psi \vdash s + pc'_s; pc'_t; \psi' \] denotes a well-typeness judgment for statements. Note that the scheduler label, progress label, and capability label are "threaded through" and appear in output contexts since statements can modify these.

We now describe the typing rules. Expression typing is mostly straightforward—the only interesting rule is DEREFF, which ensures that the thread can safely read the reference by checking if it has shared access to the reference resource.

\[ \begin{align*}
\text{ASSIGN} & \quad pc_s \sqcup pc_t \sqcup L(\tau) \sqsubseteq \ell \quad \psi \sqsubseteq \psi_r \\
\Gamma; pc_s; pc_t; \psi \vdash e : ref^{\psi_r}_\ell & \quad \tau \\
\Gamma; pc_s; pc_t; \psi \vdash e_2 : \tau \\
\Gamma; pc_s; pc_t; \psi \vdash e_1 := e_2 \dashv pc_s; pc_t; \psi
\end{align*} \]

The IF rule ensures that the thread has exclusive capability to the reference resource that is being written.

\[ \begin{align*}
\Gamma; pc_s; pc_t; \psi & \vdash \psi \neq \psi \sqsubseteq \psi_r \\
\Gamma; pc_s; pc_t; \psi \vdash e : bool & \\
\Gamma; pc_s; pc_t; \psi \vdash s_1 + pc'_s; pc'_t; \psi' \\
\Gamma; pc_s; pc_t; \psi \vdash s_2 + pc'_s; pc'_t; \psi'
\end{align*} \]

The WHILE rule ensures that the thread has exclusive capability to the reference resource that is being written.

\[ \begin{align*}
\Gamma; pc_s; pc_t; \psi & \vdash \psi \neq \psi \sqsubseteq \psi_r \\
\Gamma; pc_s; pc_t; \psi \vdash e : bool & \\
\Gamma; pc_s; pc_t; \psi \vdash s + pc_s; pc_t; \psi
\end{align*} \]

The SPAN rule allows a thread to spawn off a new thread by splitting its capability label (\(\psi\)) into two disjoint labels (\(\psi'\), \(\psi''\)), assigning one label to the new thread and the other to the continuation of the current thread. That the new capability labels are contained within the original is important for compositionality: it ensures that for a thread with capability label \(\psi''\) that is concurrent with the spawning thread, the type system treats this thread as concurrent with the new thread and the continuation—namely, \(T = \psi' \sqcap \psi'' \implies T = \psi' \sqcap \psi'' \wedge T = \psi'' \sqcap \psi''\).

The SEND rule allows a thread to send a value to another thread through a channel. Given channel type \(\gamma_{chan} \Psi_1 \Psi_2 \tau\), \(\Psi'_1\) and \(\Psi'_2\) are the send and receive endpoint resources of the channel respectively. A thread sending to or receiving from the channel must have exclusive capability over these respective resources. Recall that channels can transfer capabilities at synchronization points; channel types facilitate this by including \(\Psi'_2\) and \(\Psi'_2\) as capabilities transferred by sending and receiving threads respectively. So when a thread sends through a channel, \(\Psi'_2\) is "subtracted out" from its capability label and \(\Psi'_2\) is "added in." The rule also ensures that \(\ell\) upper-bounds the influence on control flow and progress at the sending context, and that it leaks the channel's influence on progress to the output progress label.

The RECV rule allows a thread to receive a value from another thread through a channel. The rule ensures that the receiving thread has exclusive access to the receive endpoint resource associated with the channel. The rule also subtracts out \(\Psi'_2\) from the thread's capability label and adds in \(\Psi'_2\). Finally, the channel security label that upper-bounds influence in the sending context flows to the output progress label, ensuring that the receiving thread's continuation will not leak information through progress. The JOIN rule is similar to the RECV rule, except that it works over two channels instead of one. Note that the progress labels for the two channels must lower-bound each other, since the sender for one channel is blocked until the sender on the other channel also sends.

The SELECT rule also is similar to the RECV and JOIN rules in that it transfer capabilities between sending and receiving threads and makes the channel security labels flow to the output progress label. The rule ensures that the label with which a select is tagged exactly matches the scheduler label. Note that the channel progress labels must be lower-bounded by the scheduler label since influence on which branch to execute influence the progress of senders. The rule also ensures that shared access to the global scheduler resource can be acquired.

Finally, the RSCHED and LSCHED rules are for the primitives that raise and lower the scheduler label. Both require exclusive
access to the global scheduler resource as they change the scheduler label. Raising the scheduler label can be done in any context, but (following [4, 14]) lowering the scheduler label can only be done in a completely public context, otherwise it can leak information. Because lowering the scheduler label kills all other threads, the output capability label in LSCHEd is unrelated to the input capability label—it has capability to access any resource.

C. Soundness

Theorem 1 (Well-typed programs satisfy observational determinism modulo select): Given an adversary label $\ell_a$ and

$$\Gamma; \psi \sqsubseteq \ell_a \triangleright e : \alpha \rightsquigarrow C$$

define the set of low locations $ML$ as

$$ML = \{loc \mid \Gamma[loc] = \text{ref}^{\psi, \ell}_a \tau, \ell \sqsubseteq \ell_a\}$$

and the partition $P$ of $ML$ as induced by equivalence relation

$$\Gamma[loc_1] = \text{ref}^{\psi_1, \ell_1}_a \tau_1, \Gamma[loc_2] = \text{ref}^{\psi_2, \ell_2}_a \tau_2, \psi_1 = \psi_2$$

and the set of low selects $SL$ as

$$SL = \{\xi \mid (\xi, \ell) \in \text{selects}(s), \ell \sqsubseteq \ell_a\}$$

where selects(s) are the set of identifier-label pairs for all select statement that occur in the program text of $s$. Then $s$ satisfies observational determinism modulo select relative to $P$ and $SL$.

Note that the type system naturally defends the Zdancewic–Myers definition of OD, as any partition of low memory defends it. We can easily modify the type system to defend Terauchi’s definition by modifying the well-formed rule for reference types:

$$\vdash \tau \quad L(\tau) \subseteq \ell \quad \tau \neq \psi \quad \ell \sqsubseteq \ell_a \implies \psi = \psi_{low}$$

where $\psi_{low}$ is defined to be a global low memory resource label.

The proof of Theorem 1 can be found in Appendix B. Its structure is similar to the proofs given in prior work such as Terauchi [22], combining a low-simulation argument that shows that configurations that only differ in the secret fragments of heaps have the same low-observable execution and a confluence argument that shows that simulations have deterministic executions.

D. Type inference

We define an inference procedure for well-typedness of programs. The procedure generates a set of constraints from inference rules that closely match the declarative typing rules, then attempts to solve these constraints; if a solution exists, then the program is well-typed. Constraints are of three types: lattice equations over label variables ($\ell$), and unification constraints and subtyping constraints over type variables ($\alpha_i$).

Type variables are decorated with label variables corresponding to security labels.

Inference rules have two forms, one for expressions and another for statements:

- $\Gamma; l_s; l_c; l_p; e : \alpha \rightsquigarrow C$ takes environment $\Gamma$, $l_s$, $l_c$, $l_p$, and $e$ as input and returns type variable $\alpha$ and constraint set $C$.

A selection of inference rules are shown in Figure 6.

Constraint generation is straightforward as the rules are mostly syntax-directed, with the exception of the subsumption rule for subtyping. Subsumption is uniformly applied for the DEREF, ASSIGN and SEND rules: for the latter two, to ensure the written (resp. sent) values match what is given in the reference (resp. channel) type. Since reference and channel subtyping is invariant over capability components of labels, subsumption introduces equality constraints that effectively perform an alias analysis. Well-formed constraints are uniformly applied for newref and newchan expressions.

Once constraints have been generated, inference proceeds to solve them in two stages. First, unification constraints are solved to compute base types. Note that subtyping constraints are treated like unification constraints at this stage. If these have a solution, then additional lattice constraints are added: unification constraints induce equality constraints between all labels, while subtyping constraints induce flows-to constraints for security labels and equality constraints for resource and capability labels. Finally, lattice constraints are then discharged by encoding them as first-order logic formulas, the satisfiability of which can be determined by an SMT solver [18]. Importantly, the lattice constraints are encoded in the “effectively propositional” fragment of FOL, which is decidable.

VII. Related Work

A. Information flow security for concurrent programs

This paper belongs to the long line of work on scheduler-independent approaches for information flow security in concurrent programs that aim to prevent internal timing channels. Specifically, this paper builds on the formulation of observational determinism as proposed by Zdancewic and Myers, which aim to prevent races for individual locations; in turn, they build on notions of observational determinism by Roscoe [20] and McLean [16].

Subsequent work after Zdancewic and Myers include Huisman et al. [13], which points out the possible leak to the original definition by Zdancewic and Myers that was discussed above (Section V-A). The paper also gives a new definition of observational determinism in terms of infinite traces, but as Terauchi [22] and Karbyshev et al. [14] point out, it is hard to defend using a typing discipline without being overly conservative. Because we focus on type-based enforcement mechanisms for observational determinism, we build from the original definition instead.
Giffhorn and Snelting [9] defines a whole-program analysis to defend security for concurrent programs. The security condition allows for races on low locations by making the simplifying assumption, as we do, that scheduling decisions can only be influenced by influencing control flow.

The two papers most closely related to our work are Terauchi [22] and Karbyshev et al. [14]. Terauchi defines a concurrent language with semaphores for synchronization and a type system defending observational determinism. The type system uses fractional capabilities to provide the race-freedom analysis missing from the type system defined in Zdancewic and Myers. It restricts channel communication to occur in low contexts only, allowing it to eschew the need for a progress label at the cost of expressivity. The definition of observational determinism given by Terauchi differs from the original definition by Zdancewic and Myers in that it forces updates to different low locations to occur in the same order regardless of scheduling or high initial values—a stricter condition than the original, which only enforces a “pointwise” ordering of updates to the same low location. This is in response to Huisman et al. [13], which pointed out that the combination of only requiring a prefix relation between low fragments of traces and the “pointwise” restriction on low updates can leak information. The stricter definition by Terauchi prevents this leak (see [22], Appendix A).

Like Terauchi, Karbyshev et al. defines a concurrent language with semaphores for synchronization. They define a type system that also uses fractional capabilities to prevent races and contention. They relax the complete ban on races and contention along channels by introducing a scheduler label that tracks the influence on the scheduler. A global scheduler resource governs the access of threads to their scheduler labels. Shared access to the scheduler resource means that a thread can ensure its scheduler label reflects the actual influence to the scheduler, while exclusive access allows the thread to raise or lower the scheduler label. In our type system, instead of allowing benign races and contention along channels, we use the same technique of thread-local scheduler labels and a global scheduler resource to allow secure contention along a receive site for select statements.

Unlike observational determinism, the security condition that the type system of Karbyshev et al. defends is a “batch-job” condition that only relates the initial and final memories of executed programs (where “final” is defined as the state of memory when all threads are done executing, i.e., have program body skip). While such a condition might be reasonable in sequential programs, it defines many insecure concurrent programs to be vacuously secure (the presence of one blocked thread makes the premise false), and assumes adversaries cannot observe intermediate states of a running concurrent program.

B. Enforcement mechanisms for race freedom

One of our contributions is our novel capability analysis for banning races on memory and contention over channels. It is quite reminiscent of fractional capabilities as proposed by Boyland [6], and as used by Terauchi [22] and Karbyshev et al. [14] in their type systems. Fractional capabilities are defined as maps from resource identifiers to rational numbers between 0 and 1, denoting the fraction of the resource owned by a thread. Compare this with our capability analysis, where both resources and capabilities are represented by lattice elements. Our analysis is less expressive in that it cannot “count” like fractional capabilities—in a technical sense, the operation for combining capabilities in our analysis is idempotent, whereas this operator for fractional capabilities is not—but has a much simpler inference procedure.

Terauchi and Aiken [23] define a concurrent language with several types of channels and a type system that ensures well-typed programs are deterministic. Our channel types take inspiration from this type system. The paper also supports “input-buffered” channels, which we do not support since its semantics for a language with dynamically created threads is unclear.

VIII. CONCLUSION

We have studied how various channel-like synchronization mechanisms leak information and discussed enforcement mechanisms to prevent such leaks. For rendezvous or unbuffered channels, we give a novel analysis for banning memory races and channel contention with a much simpler
inference procedure than previous approaches using fractional capabilities. We have also defined a new security condition for programs using select that relaxes observational determinism with a limited amount of nondeterminism guaranteed to not leak information.

NSF grant CNS-1704788
also the NSF Medium TWC grant

REFERENCES

Huisman et al. [13] argue that the program below satisfies the Zdancewic–Myers definition of observational determinism [24] and yet is clearly insecure:

\begin{verbatim}
lo1 := 0; lo2 := 0;
while lo1 < hi do { lo1 := lo1 + 1 }
lo2 := 1
\end{verbatim}

When hi == 1, the low-observable trace is

\[ [lo1 = 0, lo2 = 0, lo1 = 1, lo2 = 1] \]

while when hi == 2, the low-observable trace is

\[ [lo1 = 0, lo2 = 0, lo1 = 1, lo1 = 2, lo2 = 1] \]

Projecting over lo1 and lo2 individually allow the program to satisfy the Zdancewic-Myers definition, as the traces for lo1 and lo2 satisfy the prefix relation between the two runs:

\begin{verbatim}
lo1: [0, 1] and [0, 1, 2]
lo2: [0, 1] and [0, 1]
\end{verbatim}

In general, the trace for lo1 will be \([0, 1, \ldots, hi]\). So for any two runs where the value of hi differs, one trace of lo1 will be a prefix of the other. The trace of lo2 will always be \([0, 1, \ldots, hi]\).

Huisman et al. further argue that the leak in the program above can be exploited by composing the program with a thread that prints the value of lo1, which is guaranteed to equal the value of hi, as follows.

\begin{verbatim}
spawn { wait (lo2 == 1); print lo1 }
\end{verbatim}

\begin{verbatim}
lo1 := 0; lo2 := 0;
while lo1 < hi do { lo1 := lo1 + 1 }
lo2 := 1
\end{verbatim}

We note, however, that this composition arguably does not satisfy the Zdancewic–Myers definition. In their original paper [24], adversary observations are modeled uniformly as observations of updates to low memory—there is no print statement in the language. Since print is an adversary observation, it is tantamount to updating another low memory location. So we can rewrite the example as follows:

\begin{verbatim}
spawn { wait (lo2 == 1); lo3 := lo1 }
\end{verbatim}

\begin{verbatim}
lo1 := 0; lo2 := 0;
while lo1 < hi do { lo1 := lo1 + 1 }
lo2 := 1
\end{verbatim}

When hi == 1, the low-observable trace is

\[ [lo1 = 0, lo2 = 0, lo1 = 1, lo2 = 1, lo3 = 1] \]

while when hi == 2, the low-observable trace is

\[ [lo1 = 0, lo2 = 0, lo1 = 1, lo1 = 2, lo2 = 1, lo3 = 2] \]

which clearly does not satisfy the Zdancewic–Myers definition: the projected traces over the locations are

\begin{verbatim}
lo1: [0, 1] and [0, 1, 2]
lo2: [0, 1] and [0, 1]
lo3: [1] and [2]
\end{verbatim}

We make the observation above to highlight the subtle interactions between the security condition and the model of the adversary.

**APPENDIX B**

**PROOF OF OBSERVATIONAL DETERMINISM MODULO SELECT**

Our proof of correctness closely follows the structure of the one given by Terauchi [22]. Our proof of confluence is new, since instead of using fractional capabilities we use a novel technique to guarantee race-freedom.
A. Lifting well-typedness from programs to configurations

For simplicity, in the definitions and proofs below we assume programs are sequentially composed as \( s_1; (s_2; (s_3; \cdots)) \)—this does not restrict the programs considered because sequential composition is associative, and thus any program can be rewritten to follow this pattern.

We then define well-typedness of heaps, denoted \( \Gamma \vdash H \), with the judgments below.

\[
\begin{align*}
\Gamma &\vdash H &\Gamma; pc_e; pc_t; \psi \vdash v : \tau &\quad \Gamma; pc_e; pc_t; \psi \vdash \text{loc} : \text{ref}^\psi \tau
\end{align*}
\]

Definition 2 (Well-typed configurations): A configuration \( \langle S, H, \Delta \rangle \) is well-typed relative to type environment \( \Gamma \), denoted as \( \Gamma \vdash \langle S, H, \Delta \rangle \), when the following conditions hold:

- \( \Gamma \vdash H \)
- There exists \( \{ \psi_{id_1}, \ldots, \psi_{id_n} \} \) where \( \{id_1, \ldots, id_n\} = \text{dom}(\Delta) \) such that \( \top = \psi_{id_1} \sqcup \psi_{id_2} \) for all \( i \neq j \), and for all \( i \in \text{dom}(\Delta) \) it holds that \( \Gamma; pc_e^i; pc_{id}; \psi_{id} \vdash \Delta[i \mapsto \top] \vdash pc_e^i; pc_{id} \psi' \) for some \( pc_e^i, pc_{id}, pc_{id}', \psi_{id}, \psi' \).

Furthermore, all assumed scheduler labels are consistent: given \( \psi_{id_1}, \psi_{id_2} \) such that \( \top \neq \psi_{id_1} \sqcup \psi_{id_2} \) and \( \top \neq \psi_{id_1} \sqcup \psi_{id_2} \).

Lemma 1 (Well-typedness of initial configurations): Given a scheduler \( S \) and well-typed program \( s \) such that \( \Gamma \vdash H \) and \( \Gamma; pc_e; pc_{id}; \psi \vdash s \vdash pc_e; pc_{id} \psi' \) for some \( pc_e, pc_{id}, \psi, pc_{id}', \psi' \), then \( \Gamma \vdash \langle S, H, [0 \mapsto s] \rangle \).

Proof: Immediate from the definitions.

Lemma 2 (Structure of contexts with statement holes): Given \( E[s] \) for some statement \( s \), then \( E = [\cdot] \) or \( E = [\cdot]; s' \).

Proof: By inspection on the structure of evaluation contexts.

Lemma 3 (PC label of contexts with statement holes): Given \( \Gamma; pc_e; pc_{id}; \psi \vdash E[s] \vdash pc_{id}'; pc_{id} \psi \) then \( \Gamma; pc_e; pc_{id}; pc_{id}; \psi \vdash s \vdash pc_{id}''; pc_{id}'; pc_{id}'' \psi'' \). Conversely, given

\[
\begin{align*}
\Gamma; pc_e; pc_{id}; \psi \vdash E[s] &\vdash pc_{id}'; pc_{id} \psi' \\
\Gamma; pc_e; pc_{id}; pc_{id}; \psi \vdash s &\vdash pc_{id}''; pc_{id}'; pc_{id}'' \psi'' \\
\Gamma; pc_e; pc_{id}; pc_{id}; \psi &\vdash s \vdash pc_{id}''; pc_{id}'; pc_{id}'' \psi''
\end{align*}
\]

then \( \Gamma; pc_e; pc_{id}; \psi \vdash E[s'] \vdash pc_{id}'; pc_{id} \psi' \).

Proof: Immediate from Lemma 2.

Lemma 4 (PC label of contexts with expression holes): Given \( \Gamma; pc_e; pc_{id}; \psi \vdash E[e] \vdash pc_{id}'; pc_{id} \psi \) then \( \Gamma; pc_e; pc_{id}; \psi \vdash e : \tau \). Conversely, given

\[
\begin{align*}
\Gamma; pc_e; pc_{id}; \psi \vdash E[e] &\vdash pc_{id}'; pc_{id} \psi' \\
\Gamma; pc_e; pc_{id}; \psi &\vdash e : \tau \\
\Gamma; pc_e; pc_{id}; \psi &\vdash e' : \tau
\end{align*}
\]

then \( \Gamma; pc_e; pc_{id}; pc_{id}; \psi \vdash E[e'] \vdash pc_{id}'; pc_{id} \psi' \).

Proof: By inspection on the structure of evaluation contexts.

Lemma 5 (Substitution lemma): Given

\[
\begin{align*}
\Gamma[x \mapsto \tau]; pc_e; pc_{id}; \psi &\vdash s \vdash pc_{id}'; pc_{id} \psi' \\
\Gamma[x \mapsto \tau]; pc_e; pc_{id}; \psi &\vdash e' : \tau \\
\Gamma; pc_e; pc_{id}; \psi &\vdash e : \tau
\end{align*}
\]

then

\[
\begin{align*}
\Gamma; pc_e; pc_{id}; pc_{id}; \psi &\vdash s[x/e] \vdash pc_{id}'; pc_{id} \psi' \\
\Gamma; pc_e; pc_{id}; \psi &\vdash e'[x/e] : \tau
\end{align*}
\]

Proof: By induction on typing derivation.

Lemma 6 (PC subsumption): The following rules hold:

\[
\begin{align*}
\frac{pc_e' \subseteq pc_e &\quad \Gamma; pc_e; pc_{id}; \psi \vdash s \vdash pc_{id}'; pc_{id} \psi'}{\Gamma; pc_e; pc_{id}; pc_{id}; \psi \vdash s \vdash pc_{id}'; pc_{id} \psi'} &\quad \frac{pc_e' \subseteq pc_e &\quad \Gamma; pc_e; pc_{id}; \psi \vdash e : \tau}{\Gamma; pc_e; pc_{id}; \psi \vdash e : \tau}
\end{align*}
\]

Proof: By induction on the typing derivation. Intuitively, all side conditions involving \( pc_e \) are of the form \( pc_e \subseteq \ell \), so \( pc_e' \subseteq \ell \) by transitivity.
Theorem 2 (Preservation of well-typedness over configurations): Given $\Gamma \vdash \langle S, H, \Delta \rangle$ and $\langle S, H, \Delta \rangle \rightarrow \langle S', H', \Delta' \rangle$, then $\Gamma' \vdash \langle S', H', \Delta' \rangle$ where $\Gamma' \subseteq \Gamma'$.

Proof: By induction on the derivation. We case on the possible reductions.

Case **BINARY-OP**.

$$\langle S, H, \Delta[tid \mapsto E[v_1 \otimes v_2]] \rangle \rightarrow \langle S, H[\text{loc} \mapsto v], \Delta[tid \mapsto E[\otimes(v_1, v_2)]] \rangle$$

By inversion and Lemma 4,

$$\Gamma; {pc_s}; {pc_c}; \psi \vdash E[v_1 \otimes v_2] \vdash p'_{c_s}; p_{c'_s}; \psi'$$

$$\Gamma; {pc_{c_c}}; {pc_{c'_c}}; \psi \vdash \text{true} \otimes \text{false} : \text{bool}_\ell$$

Then let $\Gamma' = \Gamma[\text{loc} \mapsto \text{ref}_\ell^\psi \tau]$, and by Lemma 4 and weakening,

$$\Gamma'; {pc_s}; {pc_c}; {pc_{c'_c}}; \psi \vdash E[\text{new}_\ell^{\psi} v] \vdash p'_{c_s}; p_{c'_s}; \psi'$$

as needed.

Case **REFERENCE-NEW**.

$$\langle S, H, \Delta[tid \mapsto E[\text{new}_\ell^{\psi} v]] \rangle \rightarrow \langle S, H[\text{loc} \mapsto v], \Delta[tid \mapsto E[\text{loc}]] \rangle$$

Then by inversion and Lemma 4,

$$\Gamma; {pc_s}; {pc_c}; {pc_{c'_c}}; \psi \vdash E[\text{new}_\ell^{\psi} v] \vdash p'_{c_s}; p_{c'_s}; \psi'$$

Thus by inversion,

$$\Gamma'; {pc_{c_c}}; {pc_{c'_c}}; \psi \vdash \text{true} \otimes \text{false} : \text{bool}_\ell$$

Then let $\Gamma' = \Gamma[\text{loc} \mapsto \text{ref}_\ell^\psi \tau]$, and by Lemma 4 and weakening,

$$\Gamma'; {pc_s}; {pc_c}; {pc_{c'_c}}; \psi \vdash E[\text{new}_\ell^{\psi} v] \vdash p'_{c_s}; p_{c'_s}; \psi'$$

as needed.

Case **DEREFERENCE**.

$$\langle S, H, \Delta[tid \mapsto E[\text{loc}]] \rangle \rightarrow \langle S, H, \Delta[tid \mapsto E[H[\text{loc}]]] \rangle$$

Then by inversion, $\Gamma \vdash H$ and Lemma 4.

$$\Gamma; {pc_s}; {pc_c}; {pc_{c'_c}}; \psi \vdash E[\text{loc}] \vdash p'_{c_s}; p_{c'_s}; \psi'$$

Thus by inversion,

$$\Gamma'; {pc_{c_c}}; {pc_{c'_c}}; \psi \vdash \text{true} \otimes \text{false} : \text{bool}_\ell$$

Then let $\Gamma' = \Gamma[\text{loc} \mapsto \text{ref}_\ell^\psi \tau]$, and by Lemma 4 and weakening,

$$\Gamma'; {pc_s}; {pc_c}; {pc_{c'_c}}; \psi \vdash E[\text{loc}] \vdash p'_{c_s}; p_{c'_s}; \psi'$$

as needed.

Case **CHANNEL-NEW**. Similar to the case for **REFERENCE-NEW**.

Case **LET-BIND**.

$$\langle S, H, \Delta[tid \mapsto E[\text{let } x = v \text{ in } s]] \rangle \rightarrow \langle S, H, \Delta[tid \mapsto E[s[x/v]]] \rangle$$

Immediate from Lemma 5.

Case **ASSIGN**.

$$\langle S, H, \Delta[tid \mapsto E[\text{loc} := v]] \rangle \rightarrow \langle S, H[\text{loc} \mapsto v], \Delta[tid \mapsto E[\text{skip}]] \rangle$$

By inversion we know that

$$\Gamma; {pc_c}; {pc_{c'_c}}; \psi \vdash \text{true} \otimes \text{false} : \text{bool}_\ell$$

so we know $\Gamma \vdash H'$, as needed.
Case **CONDITIONAL-TRUE**.

\( \langle S, H, \Delta[\text{tid} \mapsto E[\text{if true then } s_1 \text{ else } s_2]] \rangle \rightarrow \langle S, H, \Delta[\text{tid} \mapsto E[s_1]] \rangle \)

Then by inversion, Lemma 3, and Lemma 6 we have

\[
\Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{if true then } s_1 \text{ else } s_2] \vdash pc'_s; pc'_c; \psi' \\
\Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{if true then } s_1 \text{ else } s_2 \vdash pc''_s; pc''_c; \psi'' \\
\Gamma; pc_s; pc_c; \psi \vdash s_1 \vdash pc''_s; pc''_c; \psi'' \\
\Gamma; pc_s; pc_c; pc_t; \psi \vdash E[s_1] \vdash pc'_s; pc'_c; \psi'
\]

as needed.

**Case CONDITIONAL-FALSE.** Similar to the **CONDITIONAL-TRUE** case.

**Case WHILE.**

\( \langle S, H, \Delta[\text{tid} \mapsto E[\text{while } v \text{ do } s]] \rangle \rightarrow \langle S, H, \Delta[\text{tid} \mapsto E[\text{if } v \text{ then } (s; \text{while } v \text{ do } s) \text{ else } \text{skip}]] \rangle \)

By inversion and Lemma 3 we know

\[
\Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{while } v \text{ do } s] \vdash pc'_s; pc'_c; \psi' \\
\Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{while } v \text{ do } s \vdash pc_s; pc_c; \psi \\
\Gamma; pc_c; pc_t; \psi \vdash v : \text{bool} \ell \\
\Gamma; pc_s; pc_c \sqcup \ell; pc_t; \psi \vdash s \vdash pc_s; pc_c; \psi
\]

Then we can construct the following derivation (note that \( pc_c \sqcup \ell \sqcup \ell = pc_c \sqcup \ell \)):

\[
A \quad \Gamma; pc_s; pc_c \sqcup \ell; pc_t; \psi \vdash s \vdash pc_s; pc_c; \psi \\
B \quad \Gamma; pc_c; pc_t; \psi \vdash \text{if } v \text{ then } (s; \text{while } v \text{ do } s) \text{ else } \text{skip} \vdash pc_c; pc_t; \psi
\]

**Case SPAWN.**

\( \langle S, H, \Delta[\text{tid} \mapsto E[\text{spawn } \{s\}]] \rangle \rightarrow \langle S, H, \Delta[\text{tid} \mapsto E[\text{skip}]] \rangle[\text{tid}' \mapsto s] \)

By inversion and Lemma 3,

\[
\Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{spawn } \{s\}] \vdash pc'_s; pc'_c; \psi' \\
\Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{spawn } \{s\} \vdash pc_s; pc_c; \psi_e \\
\Gamma; pc_c; pc_t; \psi_f \vdash s \vdash pc_s; pc_c; \psi'_f \\
\Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{skip}] \vdash pc'_s; pc'_c; \psi'
\]

To preserve the condition on PC labels for well-typed configurations, we must show for any \( \psi_t \) where \( T = \psi \sqcup \psi_t \) that \( T = \psi_c \cup \psi_t \) and \( T = \psi_f \cup \psi_t \). We already know \( T = \psi_c \cup \psi_f \) and \( \psi \subseteq \psi_c \cap \psi_f \) by inversion. Then

\[
\psi \subseteq \psi_c \\
T = \psi \sqcup \psi_t \subseteq \psi_c \cup \psi_t \\
\psi \subseteq \psi_f \\
T = \psi \sqcup \psi_t \subseteq \psi_f \cup \psi_t
\]
as needed.

**Case recv.**

\[ \langle S, H, \Delta \rangle \rightarrow \langle S, H, \Delta', \langle \text{recv loc}\{s\} \rangle \rangle \rightarrow \langle S, H, \Delta' \rangle \]

We know by inversion, Lemma 3 and Lemma 5 that

\[ \Gamma ; \psi \vdash \text{loc} : \Psi_1 \Psi_2 \tau \]

\[ \Gamma ; \psi \vdash \text{send} \psi_1 \rightarrow \text{loc} \vdash \psi_1' \psi_2' \]

\[ \Gamma ; \psi \vdash \text{recv} \psi_1 \rightarrow \text{loc} \vdash \psi_1' \psi_2' \]

So to preserve the condition for PC labels in well-typed configurations, we must show \( T = \psi_1' \cup \psi_2' \), and for any \( \psi_3 \) such that \( T = \psi_3 \cup \psi_1 \) and \( T = \psi_3 \cup \psi_2 \), it must hold that \( T = \psi_3 \cup \psi_1' \) and \( T = \psi_3 \cup \psi_2' \).

The following constraints hold by inversion:

\[ T = \psi_1 \cup \psi_2 \]

\[ \psi_1 \in \Psi_1 \cap \psi_1', T = \psi_1' \cup \psi_2', \psi_1' \cap \psi_2' \subseteq \psi_1' \]

\[ \psi_2 \in \Psi_2 \cap \psi_2', T = \psi_1' \cup \psi_2', \psi_2' \cap \psi_2' \subseteq \psi_2' \]

Then

\[ T = \psi_1 \cup \psi_2 \subseteq \psi_1' \cup \psi_2' \]

\[ T = \psi_1 \cup \psi_2 \subseteq \psi_1' \cup \psi_2' \]

\[ (\psi_1' \cap \psi_2') \cup (\psi_1'' \cap \psi_2'') = (\psi_1' \cup \psi_2') \cap (\psi_1'' \cup \psi_2'') \cap (\psi_1' \cup \psi_2'') \cap (\psi_1'' \cup \psi_2'') \]

\[ = T \cap T \cap T \cap T = T \]

\[ T = (\psi_1' \cap \psi_2') \cup (\psi_1'' \cap \psi_2'') \subseteq \psi_1' \cup \psi_2' \]

and

\[ \psi_1 \cap \psi_2 \subseteq \psi_1' \cup \psi_2' \]

\[ \psi_1 \cap \psi_2 \subseteq \psi_1' \cup \psi_2' \]

\[ \psi_3 \cup (\psi_1 \cup \psi_2) = (\psi_3 \cup \psi_1) \cap (\psi_3 \cup \psi_2) = T \cap T = T \]

\[ T = \psi_3 \cup (\psi_1 \cup \psi_2) \subseteq \psi_3 \cup \psi_1' \]

\[ T = \psi_3 \cup (\psi_1 \cup \psi_2) \subseteq \psi_3 \cup \psi_2' \]

as needed.

**Case join.** Similar to recv.

**Case select-left.** Similar to recv.

**Case select-right.** Similar to recv.

**Case raise-schedule.**

\[ \langle S, H, \Delta \rangle \rightarrow \langle S, H, \Delta' \rangle \rightarrow \langle S, H, \Delta'' \rangle \]

...
\(\langle H, \bullet \rangle \rightarrow_L \langle H, \bullet \rangle\)

\(\langle H, \bullet \rangle \rightarrow_L \langle H, \text{skip} \rangle\)

\(\langle H, \bullet := v \rangle \rightarrow_L \langle H, \text{skip} \rangle\)

\(\langle H, \text{if } \bullet \text{ then } s_1 \text{ else } s_2 \rangle \rightarrow_L \langle H, \bullet \rangle\)

\((H, \text{while } \bullet \text{ do } s) \rightarrow_L \langle H, \text{if } \bullet \text{ then } \bullet \text{ else } \bullet \rangle\)

\((H, \text{send } v \text{ to } \bullet) \rightarrow_L \langle H, \text{skip} \rangle\)

\((H, \text{recv } \bullet \{ x \} \{ s \}) \rightarrow_L \langle H, \bullet \rangle\)

\(\langle H, \text{join } (\bullet, \bullet)(x_1, x_2) \{ s \} \rangle \rightarrow_L \langle H, \bullet \rangle\)

\(\langle H, \text{select}_\xi, \ell, v_1(x_1) \{ s_1 \} \parallel v_2(x_2) \{ s_2 \} \rangle \rightarrow_L \langle H, \bullet \rangle\)

\(\langle S, H, \Delta[\text{tid} \mapsto E[\text{select}_\xi, \ell, v_1(x_1) \{ s_1 \} \parallel v_2(x_2) \{ s_2 \}]] \rangle \rightarrow \langle S', H, \Delta[\text{tid} \mapsto E[\bullet]] \rangle\)

\(\langle S, H, \Delta[\text{tid} \mapsto E[\text{select}_\xi, \ell, v_1(x_1) \{ s_1 \} \parallel v_2(x_2) \{ s_2 \}]] \rangle \rightarrow \langle S', H, \Delta[\text{tid} \mapsto E[\bullet]] \rangle\)

Fig. 7: Extended operational semantics for \(\ell\)-erased programs.

\(\Gamma; pc_c; pc_e; pc_t; \psi \vdash \bullet : \tau\)

\(\Gamma; pc_a; pc_c; pc_t; \psi \vdash \bullet \not\vdash pc_a; pc_t; \psi\)

Fig. 8: Typing rules for \(\ell\)-erased expressions and statements.

By inversion and Lemma 3,

\(\Gamma; pc_a; pc_c; pc_t; \psi \vdash E[\text{raise\_schedule} \; \ell] \vdash pc'_a; pc'_c; \psi'\)

\(\Gamma; pc_a; pc_c; pc_t; \psi \vdash \text{raise\_schedule} \; \ell \vdash \ell; pc_t; \psi\)

\(\Gamma; \ell; pc_a; pc_c; pc_t; \psi \vdash \text{skip} \vdash \ell; pc_t; \psi\)

\(\Gamma; \ell; pc_a; pc_c; pc_t; \psi \vdash E[\text{skip}] \vdash pc'_a; pc'_c; \psi'\)

We know that \(\psi \sqsubseteq \psi_{\text{sched}}\), so \text{tid} is the only thread that has an assumption about the scheduler label; thus the scheduler invariant for well-typed configurations is preserved.

**Case LOWER-SCHEDULE.** Similar to RAISE-SCHEDULE. The output scheduler and capability labels can be arbitrary because after reduction the thread calling raise_schedule is the only one alive.

\[\square\]

**B. Simulation**

We use a simulation argument to prove that two configurations differing only in high parts of memory have the same low behavior. To define simulations, we extend the original language to an “erased” language with erased expressions and statements, denoted as \(\bullet\). Operational semantics and typing rules for erased expressions and statements are shown in Figure 7 and Figure 8 respectively. We now define an erasure judgment, the rules for which are shown in Figure 9.

**Definition 3 (Erasure):** A configuration \(\Gamma \vdash \langle S_2, H_2, \Delta_2 \rangle\) is an \(\ell\)-erasure of \(\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle\), written as

\[\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle \preceq_\ell \langle S_2, H_2, \Delta_2 \rangle\]

when the following hold:

- For all \(\text{tid} \in \text{dom}(\Delta_1)\), \(\Gamma; pc_a; pc_c; pc_t; \psi \vdash \Delta_1[\text{tid}] \preceq_\ell \Delta_2[\text{tid}] \vdash pc'_a; pc'_c; \psi'\) for some \(pc_a, pc_c, pc_t, \psi, pc'_a, pc'_c, \psi'\).
- \(\Gamma \vdash H_1 \preceq_\ell H_2\)
- \(S_1 \sim_\ell S_2\)

Note that we still need to show that erasures are well-typed erased programs.

**Lemma 7 (Erasures are well-typed erased configurations):** \(\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle \preceq_\ell \langle S_2, H_2, \Delta_2 \rangle\) implies that \(\Gamma \vdash \langle S_2, H_2, \Delta_2 \rangle\).

**Proof:** Assume we have for some thread \(\text{tid} \in \Delta_1\) and \(\Delta_2\), \(\Gamma; pc_a; pc_c; pc_t; \psi \vdash s_1 \preceq_\ell s_2 \vdash pc'_a; pc'_c; \psi'\) for some \(pc_a, pc_c, pc_t, \psi, pc'_a, pc'_c, \psi'\). Then we need to show \(\Gamma; pc_a; pc_c; pc_t; \psi \vdash s_2 \vdash pc''_a; pc''_c; \psi''\) for some \(pc''_a, pc''_c, \psi''\). This is mostly immediate, with the exception of erased branches for conditional statements, since well-typed erased statements are assumed to have the same output contexts.
\[ \Gamma; pc_s; pc_t; v \vdash e : \tau \quad \Gamma; pc_s; pc_t; v \vdash e : \tau \quad \Gamma; pc_s; pc_t; v \vdash s \leq_{\ell} s' + pc_s''; pc_t''; \psi' \]

\[ \Gamma \vdash H_1 \leq_{\ell} H_2 \]

\[ \Gamma; pc_s; pc_t; v \vdash e : \tau \quad L(\tau) \subseteq \ell \quad \Gamma; pc_s; pc_t; v \vdash e : \tau \quad L(\tau) \not\subseteq \ell \]

\[ \Gamma; pc_s; pc_t; v \vdash s \leq_{\ell} s' + pc_s''; pc_t''; \psi' \]

\[ \Gamma \vdash H_1 \leq_{\ell} H_2 \quad L(\Gamma[\text{loc}]) \subseteq \ell \quad \Gamma \vdash H_1 \leq_{\ell} H_2 \quad L(\Gamma[\text{loc}]) \not\subseteq \ell \]

\[ \Gamma \vdash H_1[\text{loc} \mapsto v] \leq_{\ell} H_2[\text{loc} \mapsto v] \]

\[ \Gamma \vdash H_1[\text{loc} \mapsto v] \leq_{\ell} H_2[\text{loc} \mapsto \bullet] \]

Fig. 9: Selected rules for erasure.

(for scheduler, progress, and capability labels) as their input contexts. That is, we assume that any changes to the scheduler, progress and capability labels performed inside an erased statement can be ignored.

\[ s_1 = E_1[\text{if } v \text{ then } s_{11} \text{ else } s_{12}] \]

\[ s_2 = E_2[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet] \]

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash E_1[\text{if } v \text{ then } s_{11} \text{ else } s_{12}] + pc_s''; pc_t''; \psi' \]

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash E_2[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet] + pc_s''; pc_t''; \psi'. \]

Then by inversion

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash \text{if } v \text{ then } s_{11} \text{ else } s_{12} + pc_s''; pc_t''; \psi' \]

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash \text{if } v : \tau \]

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash s_{11} + pc_s''; pc_t''; \psi' \]

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash s_{12} + pc_s''; pc_t''; \psi' \]

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash \bullet + pc_s; pc_t; \psi \]

\[ \Gamma; pc_s; pc_c; pc_t; v \vdash \bullet + pc_s; pc_t; \psi \]

Note that if \( pc_s'' = pc_s \), \( pc_t'' = pc_t \), and \( \psi'' = \psi \) then the proof is immediate because we can just replace either \( s_{11} \) and \( s_{12} \) with \( \bullet \) directly. Here we cover the cases when these three equalities are not satisfied.

**Case:** \( \psi'' \neq \psi \). This is only possible when then is synchronization in \( s_{11} \) and \( s_{12} \). But that would imply \( pc_t \not\subseteq \ell \), and thus the continuation after the conditional is also erased; that is, \( E_2 = \emptyset \). Then

\[ \Gamma; pc_s; pc_t; pc_c; v \vdash E_2[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet] + pc_s; pc_t; \psi \]

as needed.

**Case:** \( pc_t'' \neq pc_t \). This is only possible when there is synchronization in \( s_{11} \) and \( s_{12} \), and thus this is the same as the first case.
Case: $pc_s'' \neq pc_s$. This is only possible when there is at least one `raise_schedule` command in $s_{11}$ and $s_{12}$; note that there cannot be a `lower_schedule` command in either of these because it can only be executed in completely public contexts—i.e. it requires a PC label and progress label of $\bot$, and $\bot \subset \bot \subseteq \ell$ for any $\ell$. There are then two subcases:

**Subcase:** $\psi \nsubseteq \psi_{\text{sched}}$. Then there must have been synchronization inside of $s_{11}$ and $s_{12}$, since there are `raise_schedule` commands inside of these and it requires $\psi$ to have exclusive access to $\psi_{\text{sched}}$. Thus this is the same as the first two cases.

**Subcase:** $\psi \subseteq \psi_{\text{sched}}$. We can assume there is no synchronization in $s_{11}$ and $s_{12}$, because we are in the first subcase. Then let $s_1 = \bullet; \text{raise\_schedule } pc_s''$ and we know

\[
\Gamma; pc_s; pc_c; pc_{\ell}; \psi \vdash s_1 \vdash pc_s''; pc_{\ell}; \psi
\]

\[
\Gamma; pc_s; pc_c; pc_{\ell}; \psi \vdash E[\text{if } \bullet then s_2 else s_2] \vdash pc_s''; pc_{\ell}; \psi'
\]

since $pc_s'' = pc_s$ and $\psi'' = \psi$, otherwise we are in the first two cases. Given that $E[\text{if } \bullet then s_2 else s_2]$ is semantically equivalent to $E[\text{if } \bullet then \bullet else \bullet]$ we can assume the erasure of $s_1$ is really the former—since `raise_schedule` is a no-op, this will not affect the simulation argument (Lemma 12) below. Thus we have discharged the proof obligation.

On the other hand, an erased program being well-typed does not imply that it is an erasure for some program. Thus we need a syntactic way to distinguish actual erasures by checking that “the bullets are in the right place.”

**Definition 4 (Correct bullet placement):** A program $s$ satisfies correct bullet placement, written as $\text{CBP}(s)$, when all of the following syntactic patterns hold in $s$. Note we can lift this notion for a thread pool $\Delta$ such that $\text{CBP}(\Delta)$ in the obvious way.

\[
\begin{align*}
\text{if } \bullet \text{ then } s_1 \text{ else } s_2 & \implies s_1 = \bullet, s_2 = \bullet \\
\text{while } \bullet \text{ do } s & \implies s = \bullet \\
\text{recv } \bullet(x)\{s_1\}; s_2 & \implies s_1 = \bullet, s_2 = \bullet \\
\text{join } (\bullet, e_2)(x_1, x_2)\{s_1\}; s_2 & \implies s_1 = \bullet, s_2 = \bullet \\
\text{join } (e_1, \bullet)(x_1, x_2)\{s_1\}; s_2 & \implies s_1 = \bullet, s_2 = \bullet \\
\text{select}_{\xi, \ell} \bullet(x_1)\{s_1\} \parallel e_2(x_2)\{s_2\}; s_3 & \implies s_1 = \bullet, s_2 = \bullet, s_3 = \bullet \\
\text{select}_{\xi, \ell} e_1(x_1)\{s_1\} \parallel (x_2)\{s_2\}; s_3 & \implies s_1 = \bullet, s_2 = \bullet, s_3 = \bullet
\end{align*}
\]

**Lemma 8 (Erasures have correct bullet placement):** Given $\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle \preceq \langle S_2, H_2, \Delta_2 \rangle$ then $\text{CBP}(\Delta_2)$.

**Proof:** Immediate from Definition 3 and Definition 4.

**Lemma 9 (Preservation of well-typed erasures):** Given well-typed erased configuration and step

\[
\begin{align*}
\Gamma \vdash \langle S, H, \Delta \rangle \\
\langle S, H, \Delta \rangle \rightarrow \langle S', H', \Delta' \rangle
\end{align*}
\]

such that $\text{CBP}(\Delta)$, then $\Gamma' \vdash \langle S', H', \Delta' \rangle$ and $\text{CBP}(\Delta')$ where $\Gamma \subseteq \Gamma'$.

**Proof:** By induction on derivation. Most of the cases are the same as in Theorem 2, so we only cover the cases involving erased expressions and statements here.

**Case BULLET-BULLET.** Immediate; the thread steps to the same statement.

**Case BULLET-SKIP.**

\[
\langle S, H, \Delta'[\text{tid } \mapsto E[\bullet]] \rangle \rightarrow \langle S, H, \Delta''[\text{tid } \mapsto E[\text{skip}]] \rangle
\]

We know by inversion and Lemma 3

\[
\begin{align*}
\Gamma; pc_s; pc_c; pc_{\ell}; \psi \vdash E[\bullet] \vdash pc_s''; pc_{\ell}; \psi'' \\
\Gamma; pc_s; pc_c; pc_{\ell}; \psi \vdash \bullet \vdash pc_s; pc_{\ell}; \psi \\
\Gamma; pc_s; pc_c; pc_{\ell}; \psi \vdash \text{skip} \vdash pc_s; pc_{\ell}; \psi \\
\Gamma; pc_s; pc_c; pc_{\ell}; \psi \vdash E[\text{skip}] \vdash pc_s; pc_{\ell}; \psi''
\end{align*}
\]

as needed. Intuitively, because the continuation after the erased statement $\bullet$ type-checks assuming that $\bullet$ did not change the scheduler, progress, or capability label, keeping these the same preserves typability.

**Case IF-ERASED.**

\[
\langle S, H, \Delta''[\text{tid } \mapsto E[\text{if } \bullet then \bullet else \bullet]] \rangle \rightarrow \langle S, H, \Delta''[\text{tid } \mapsto E[\bullet]] \rangle
\]
We know by inversion, Lemma 3 and Lemma 6

\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet] \vdash pc_s''; pc_t''; \psi'' \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{if } \bullet \text{ then } \bullet \text{ else } \bullet \vdash pc_s; pc_t; \psi \]
\[ \Gamma; pc_c; pc_t; \psi \vdash \bullet : \tau \]
\[ \Gamma; pc_s; pc_c \sqcup L(\tau); pc_t; \psi \vdash \bullet \vdash pc_s; pc_t; \psi \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash \bullet \vdash pc_s; pc_t; \psi \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\bullet] \vdash pc_s''; pc_t''; \psi'' \]

as needed.

**Case while-erased.**

\[ \langle S, H, \Delta''[\text{tid } \mapsto E[\text{while } \bullet \text{ do } \bullet]] \rangle \rightarrow \langle S, H, \Delta''[\text{tid } \mapsto E[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet]] \rangle \]

We know by inversion and Lemma 3

\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{while } \bullet \text{ do } \bullet] \vdash pc_s''; pc_t''; \psi'' \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{while } \bullet \vdash pc_s; pc_t; \psi \]
\[ \Gamma; pc_c; pc_t; \psi \vdash \bullet : \tau \]
\[ \Gamma; pc_s; pc_c \sqcup L(\ell); pc_t; \psi \vdash \bullet \vdash pc_s; pc_t; \psi \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{if } \bullet \text{ then } \bullet \text{ else } \bullet \vdash pc_s; pc_t; \psi \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet] \vdash pc_s''; pc_t''; \psi'' \]

as needed.

**Case send-erased.**

\[ \langle S, H, \Delta''[\text{tid } \mapsto E[\text{send } v \text{ to } \bullet]] \rangle \rightarrow \langle S, H, \Delta''[\text{tid } \mapsto E[\text{skip}]] \rangle \]

We know by inversion and Lemma 3

\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash E[\text{send } v \text{ to } \bullet] \vdash pc_s''; pc_t''; \psi'' \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{send } v \text{ to } \bullet \vdash pc_s; pc_t; \psi \]

We know by Lemma 2 either \( E = [\cdot] \) or \( E = [\cdot]; s \) when it is the former, then \( \Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{skip } \vdash pc_s; pc_t; \psi \) as needed. When it is the latter, then \( s = \bullet \) by correct bullet placement—we know \( pc_t' \sqsubseteq \ell \) since it is lower bounded by the label on the channel \( \bullet \). Then \( \Gamma; pc_s; pc_c; pc_t; \psi \vdash \text{skip } \vdash pc_s; pc_t; \psi \) as needed. The intuition is that the continuation of the send on an erased channel is erased, thus we can keep the original scheduler and capability labels; any modification of these in high contexts is not visible. This preserves the capability and scheduler invariants for well-typed configurations.

**Case recv-erased.**

**Case join-erased.**

**Case select-left-erased.**

**Case select-right-erased.**

**Case select-high-erased.** Similar to send-erased.

---

**Definition 5 \((\ell, \text{-partition of an environment})\):** The \( \ell \)-partition of an environment \( \ell \), written as \( P_\ell(\Gamma) \), is defined as the partition \( P \) of \( \ell \)-visible locations \( M \) defined in \( \Gamma \), where

\[ M = \{ \text{loc} | \Gamma[\text{loc}] = \text{ref}_{t_r}^{\psi_r}, \tau_r \sqsubseteq \ell \} \]

and \( P \) is induced by the equivalence relation

\[ \text{loc}_1 \sim \text{loc}_2 \iff \Gamma[\text{loc}_1] = \text{ref}_{t_1}^{\psi_1}, \tau_1, \Gamma[\text{loc}_2] = \text{ref}_{t_2}^{\psi_2}, \tau_2, \psi_1 = \psi_2. \]

**Lemma 10 (Statement erasure substitution):** Given

\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash E[s_1] \leq_\ell E[s_1'] \vdash pc_s''; pc_t''; \psi'' \]
\[ \Gamma; pc_s; pc_c; pc_t; \psi \vdash s_1 \leq_\ell s_1' \vdash pc_s; pc_t; \psi' \]
\[ \Gamma; pc_s''; pc_t''; pc_c''; \psi'' \vdash s_2 \leq_\ell s_2' \vdash pc_s; pc_t; \psi' \]

then \( \Gamma; pc_s''; pc_t''; pc_c''; \psi'' \vdash E[s_2] \leq_\ell E[s_2'] \vdash pc_s; pc_t; \psi' \).
Proof: Follows from Lemma 3 and erasure definition.

We prove a confinement lemma that shows statements typeable in a high context step to low-equivalent configurations. In the lemma definition we lift scheduler equivalence and heap restriction to be over labels in the obvious way.

Lemma 11 (Confinement): Given adversary label ℓ, let \(\langle S, H, \Delta \rangle\) such that thread tid with program \(s\) and \(\Gamma; pc_s; pc_c; pc_t; \psi \vdash \tau\) \(\triangleright s; pc'_t; pc'\psi\) where \(pc_c \cup pc_t \not\subseteq \ell\) steps to \(\langle S', H', \Delta' \rangle\). Then \(S \sim_\ell S'\) and \(H|\ell = H'|\ell\).

Proof: The only interesting cases are for allocation of new references and channels, assignment and receiving on selects, as these are the only ones that modify state.

Case NEWCHAN.

Case SELECT. We know from the SELECT typing rule that \(pc_c \cup pc_t\) lower bound the reference label \(\ell_r\) but since we assume \(pc_c \cup pc_t \not\subseteq \ell\) we also know \(\ell_r \not\subseteq \ell\). That means the updated location is in the high fragment of the heap, and thus \(H|\ell = H'|\ell\) as needed.

Case SELECT-LEFT.

Case SELECT-RIGHT.

Case ASSIGN. We know from the ASSIGN typing rule that \(pc_c \cup pc_t\) lower-bounds the label \(\ell_s\) with which the select statement is tagged. Since \(pc_c \cup pc_t \not\subseteq \ell\), we know also that \(\ell_s \not\subseteq \ell\). This fact combined with the “narrowness” of schedulers immediately gives us \(S \sim_\ell S'\) as needed.

Lemma 12 (One-step simulation): Given

- \(\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle \preceq_\ell \langle S_2, H_2, \Delta_2 \rangle\)
- \(\langle S_1, H_1, \Delta_1 \rangle \rightarrow \langle S'_1, H'_1, \Delta'_1 \rangle\)
- \(\Gamma \vdash \langle S'_1, H'_1, \Delta'_1 \rangle\)

then there exists \(\langle S'_2, H'_2, \Delta'_2 \rangle\) such that \(\langle S_2, H_2, \Delta_2 \rangle \rightarrow \ast \langle S'_2, H'_2, \Delta'_2 \rangle\) and \(\Gamma \vdash \langle S'_1, H'_1, \Delta'_1 \rangle \preceq_\ell \langle S'_2, H'_2, \Delta'_2 \rangle\). Furthermore, given the trace \(T_2\) induced from \(\langle S_2, H_2, \Delta_2 \rangle \rightarrow \ast \langle S'_2, H'_2, \Delta'_2 \rangle\), \(\{H_1, H_1'\} =_{p_e(\Gamma)} T_2\).

Proof: By induction on the derivation. We case on the possible reductions. Note that we only consider statements typeable in low contexts here, as the confinement lemma above (Lemma 11) makes one-step simulation be immediate for statements typeable in high contexts.

Case ASSIGN.

\(\langle S_1, H_1, \Delta'_1[\text{tid} \rightarrow E_1[\text{loc} := v]] \rangle \rightarrow \langle S_1, H_1[\text{loc} \rightarrow v], \Delta'_1[\text{tid} \rightarrow E_1[\text{skip}]] \rangle\)

We know \(\Gamma[\text{loc}] = \text{ref}\_\psi, \tau\). The proof is immediate when \(\ell_r \not\subseteq \ell\) since the erased configuration can take the exact same step, so we only consider the case when \(\ell_r \subseteq \ell\). In this case the erased configuration can take the step

\(\langle S_2, H_2, \Delta'_2[\text{tid} \rightarrow E_2[\bullet := v]] \rangle \rightarrow \langle S_2, H_2, \Delta'_2[\text{tid} \rightarrow E_2[\text{skip}]] \rangle\)

We know \(\Gamma \vdash \text{H} \preceq_\ell \text{H}_2\) so \(\text{H}_2[\text{loc}] = \bullet\) and thus \(\Gamma \vdash \text{H}_1[\text{loc} \rightarrow v] \preceq_\ell \text{H}_2\). By Lemma 10 we have

\(\Gamma; pc_s; pc_c; pc_t; \psi \vdash E_1[\text{skip}] \preceq_\ell E_2[\text{skip}] \vdash pc'_s; pc'_t; \psi'\)

as needed. Furthermore, since \(\ell_r \not\subseteq \ell\) then \text{loc} is not part of \(\ell\)-visible memory. Thus \([H_1, H'_1] =_{p_e(\Gamma)} [H_2, H_2]\) as needed.

Case CONDITIONAL-TRUE.

\(\langle S_1, H_1, \Delta'_1[\text{tid} \rightarrow E_1[\text{if} \text{ true then } s_1 \text{ else } s_2]] \rangle \rightarrow \langle S_1, H_1, \Delta'_1[\text{tid} \rightarrow E_1[\text{skip}]] \rangle\)

Assume the guard label \(\ell_g\) such that \(\ell_g \not\subseteq \ell\) when \(\ell_g \not\subseteq \ell\) the erased configuration can take the exact same step). Then the erased configuration can take the step

\(\langle S_2, H_2, \Delta'_2[\text{tid} \rightarrow E_2[\text{if} \bullet \text{ then } \bullet \text{ else } \bullet]] \rangle \rightarrow \langle S_2, H_2, \Delta'_2[\text{tid} \rightarrow E_2[\bullet]] \rangle\)

By Lemma 10 we have \(\Gamma; pc_s; pc_c; pc_t; \psi \vdash E_1[s_1] \preceq_\ell E_2[\bullet] \vdash pc'_s; pc'_c; \psi'\) as needed. There are no heap modifications so trace equivalence of the steps is immediate.

Case CONDITIONAL-FALSE. Similar to CONDITIONAL-TRUE.

Case WHILE.

\(\langle S_1, H_1, \Delta'_1[\text{tid} \rightarrow E_1[\text{while } v \text{ do } s]] \rangle \rightarrow \langle S_1, H_1, \Delta'_1[\text{tid} \rightarrow E_1[\text{if } v \text{ then } (s; \text{while } v \text{ do } s) \text{ else } \text{skip}]] \rangle\)

Assume the guard label \(\ell_g\) such that \(\ell_g \not\subseteq \ell\). Then the erased configuration can take the step

\(\langle S_2, H_2, \Delta'_2[\text{tid} \rightarrow E_2[\text{while } \bullet \text{ do } s]] \rangle \rightarrow \langle S_2, H_2, \Delta'_2[\text{tid} \rightarrow E_2[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet]] \rangle\)
Notice that the branches of the conditional erase to \( \bullet \) since their PC label does not flow to \( \ell \) since \( \ell_g \not\subseteq \ell \). Thus
\[
\Gamma; \text{pc}_s; \text{pc}_t; \psi \vdash E_1[\text{if } v \text{ then } (s; \text{while } v \text{ do } s) \text{ else skip}] \preceq_{\ell} E_2[\text{if } \bullet \text{ then } \bullet \text{ else } \bullet] \vdash \text{pc}_s'; \text{pc}_t'; \psi'
\]
as needed. There are no heap modifications so trace equivalence of the steps is immediate.

**Case SPAWN.** Immediate; the erased program can take the same step. Intuitively, no part of a spawn command can be erased—otherwise the entire command would have been erased already because of a high PC or progress label. There are no heap modifications so trace equivalence of the steps is immediate.

**Case RECV.**
\[
\langle S_1, H_1, \Delta''_s | [\text{tid}_1 \mapsto E_{11}[\text{send } v \text{ to } \text{loc}]] | [\text{tid}_2 \mapsto E_{12}[\text{recv } \text{loc}(x)\{s\}]] \rangle
\]
We know \( \Gamma[\text{loc}] = \text{chan}_{\ell_1} \Psi_1 \Psi_2 \tau \). Assume \( \ell_c \not\subseteq \ell \). Then we know loc and \( s \) both erase to \( \bullet \) (the latter because of a high progress label). So we know the erased program can take steps
\[
\langle S_2, H_2, \Delta''_s | [\text{tid}_1 \mapsto E_{21}[\text{send } v \text{ to } \bullet]] | [\text{tid}_2 \mapsto E_{22}[\text{recv } \bullet(x)\{\bullet\}]] \rangle
\]
and by two applications of Lemma 10 we know
\[
\Gamma; \text{pc}_s; \text{pc}_c_1; \text{pc}_c_2; \psi \vdash E_{11}[\text{skip}] \preceq_{\ell} E_{21}[\text{skip}] \vdash E_{22}[\bullet] \vdash E_{22}[\bullet]
\]
as needed. There are no heap modifications so trace equivalence of the steps is immediate.

**Case SELECT-LEFT.**
\[
\langle S_1, H_1, \Delta''_s | [\text{tid}_1 \mapsto E_{11}[\text{send } v \text{ to } \text{loc}]] | [\text{tid}_2 \mapsto E_{12}[\text{select}_{\ell_s} \text{loc}_x(x_1)\{s_1\} \parallel \text{loc}_x(x_2)\{s_2\}]] \rangle
\]
We know \( \Gamma[\text{loc}_1] = \text{chan}_{\ell_1} \Psi_1 \Psi_1 \tau_1 \) and \( \Gamma[\text{loc}_2] = \text{chan}_{\ell_s} \Psi_2 \Psi_2 \tau_2 \). There are three subcases involving erased statements and/or expressions.

**Subcase: \( \ell_1 \not\subseteq \ell \) and \( \ell_s \subseteq \ell \).** Then the first channel is erased and the erased configuration can take steps
\[
\langle S_2, H_2, \Delta''_s | [\text{tid}_1 \mapsto \bullet] | [\text{tid}_2 \mapsto E_{22}[\text{select}_{\ell_s} \bullet(x_1)\{\bullet\} \parallel \text{loc}_x(x_2)\{\bullet\}]] \rangle
\]
Since \( S_1 \sim_{\ell} S_2 \), we know \( \pi_1(S_1(\xi)) \sim_{\ell} \pi_2(S_2(\xi)) \), and thus \( S'_1 \sim_{\ell} S'_2 \) as needed. By two applications of Lemma 10 we know
\[
\Gamma; \text{pc}_s; \text{pc}_c_1; \text{pc}_c_2; \psi \vdash E_{11}[\text{skip}] \preceq_{\ell} E_{21}[\text{skip}] \vdash E_{22}[\bullet] \vdash E_{22}[\bullet]
\]
as needed.

**Subcase: \( \ell_2 \not\subseteq \ell \) and \( \ell_s \subseteq \ell \).** Similar to the first subcase; select will step to \( \bullet \) also because the progress label of both branches are lower bounded by both channel labels.

**Subcase: \( \ell_s \not\subseteq \ell \).** Since \( \ell_s = \text{pc}_c \subseteq \ell_1 \) and \( \ell_s = \text{pc}_c \subseteq \ell_2 \), it must be the case that \( \ell_1 \not\subseteq \ell \) and \( \ell_2 \not\subseteq \ell \). Thus the channels and branches of the select will be erased. Then the erased configuration can take steps
\[
\langle S_2, H_2, \Delta''_s | [\text{tid}_1 \mapsto E_{21}[\text{send } v \text{ to } \bullet]] | [\text{tid}_2 \mapsto E_{22}[\text{select}_{\ell_s} \bullet(x_1)\{\bullet\} \parallel \text{loc}_x(x_2)\{\bullet\}]] \rangle
\]
By two applications of Lemma 10 we know
\[
\Gamma; \text{pc}_s; \text{pc}_c; \psi \vdash E_{11}[\text{skip}] \preceq_{\ell} E_{21}[\text{skip}] \vdash E_{22}[\bullet] \vdash E_{22}[\bullet]
\]
as needed. Notice that since $\ell_s \not\subseteq \ell$, the erased configuration does not actually use the scheduler, so we need to prove $S_1 \sim_{\ell} S_2$. Since schedulers are “narrow,” we know for all $\xi \neq \xi'$ that $S_1(\xi) = S_2(\xi')$. Since $\xi$ is not $\ell$-visible and $S_1 \sim_{\ell} S_2$, it must be the case that $S_1' \sim_{\ell} S_2$ as needed. There are no heap modifications so trace equivalence of the steps is immediate.

**Case SELECT-LEFT.** Similar to SELECT-LEFT.

**Case LSCHED.** Immediate; the erased program can take the same step. There are no heap modifications so trace equivalence of the steps is immediate.

---

**Theorem 3 (Simulation):** Given

- $\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle \preceq_{\ell} \langle S_2, H_2, \Delta_2 \rangle$
- $\langle S_1, H_1, \Delta_1 \rangle \rightarrow^* \langle S_1', H_1', \Delta_1' \rangle$
- $\Gamma_1 \vdash \langle S_1', H_1', \Delta_1' \rangle$

then there exists $\langle S_2', H_2', \Delta_2' \rangle$ such that $\langle S_2, H_2, \Delta_2 \rangle \rightarrow^* \langle S_2', H_2', \Delta_2' \rangle$ and $\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle \preceq_{\ell} \langle S_2, H_2, \Delta_2 \rangle$.

Furthermore, given traces $T_1, T_2$ induced from $\langle S_1, H_1, \Delta_1 \rangle \rightarrow^* \langle S_1', H_1', \Delta_1' \rangle$ and $\langle S_2, H_2, \Delta_2 \rangle \rightarrow^* \langle S_2', H_2', \Delta_2' \rangle$ respectively, $T_1 \equiv_{P_i(\Gamma')} T_2$.

**Proof:** By repeated applications of Lemma 12. ■

---

**C. Determinism**

We now show that well-typed erasures are deterministic.

**Lemma 13 (One-step confluence of well-typed erased configurations):** Given well-typed $\ell$-erased configuration $\Gamma \vdash \langle S, H, \Delta \rangle$ and such that $\text{CBP}(\Delta)$ and $\langle S, H, \Delta \rangle \rightarrow (S_1, H_1, \Delta_1)$ and $\langle S, H, \Delta \rangle \rightarrow (S_2, H_2, \Delta_2)$, there exists $(S_3, H_3, \Delta_3)$ such that $(S_1, H_1, \Delta_1) \rightarrow (S_3, H_3, \Delta_3)$ and $(S_1, H_2, \Delta_2) \rightarrow (S_3, H_3, \Delta_3)$. Furthermore, the traces $T_1, T_2$ respectively induced by the two series of reduction steps

$$
\langle S, H, \Delta \rangle \rightarrow \langle S_1, H_1, \Delta_1 \rangle \rightarrow \langle S_3, H_3, \Delta_3 \rangle
$$

$$
\langle S, H, \Delta \rangle \rightarrow \langle S_2, H_2, \Delta_2 \rangle \rightarrow \langle S_3, H_3, \Delta_3 \rangle
$$

are stutter-equivalent relative to the partition $P$ induced by the regions of configuration typing—i.e. $T_1 \equiv_{P_i(\Gamma')} T_2$.

**Proof:** By induction over the reduction relation. The intuition is that though well-typed configurations can step to different configurations (by virtue of the operational semantics being non-deterministic), such steps can always commute with each other. This is the case because the capability-tracking of the type system ensures that the side effects performed by threads cannot interfere with other threads.

Here we focus on the important cases, the ones involving references and channels.

**Case: no write-race.** Let $\Delta[\text{tid}_1] = E_1[\text{loc}_1 := v_1]$ and $\Delta[\text{tid}_2] = E_2[\text{loc}_2 := v_2]$ such that

$$
\langle S, H_1, \Delta_1 \rangle = \langle S, H[\text{loc}_1 \mapsto v_1], \Delta[\text{tid} \mapsto E_1[\text{skip}]] \rangle
$$

$$
\langle S, H_2, \Delta_2 \rangle = \langle S, H[\text{loc}_2 \mapsto v_2], \Delta[\text{tid'} \mapsto E_2[\text{skip}]] \rangle
$$

It suffices to show $\text{loc}_1 \neq \text{loc}_2$ such that the updates commute and

$$
\langle S, H_3, \Delta_3 \rangle = \langle S, H[\text{loc}_1 \mapsto v_1][\text{loc}_2 \mapsto v_2], \Delta[\text{tid} \mapsto E[\text{skip}]][\text{tid'} \mapsto E'[\text{skip}]] \rangle
$$

$$
\langle S, H_3, \Delta_3 \rangle = \langle S, H[\text{loc}_1 \mapsto v_1][\text{loc}_2 \mapsto v_2], \Delta[\text{tid} \mapsto E[\text{skip}]][\text{tid'} \mapsto E'[\text{skip}]] \rangle
$$

Assume $\text{loc}_1 = \text{loc}_2$. By inversion on $\Gamma \vdash (S, H, \Delta)$ we know that

$$
\Gamma; \text{pc}_1^2; \text{pc}_1^1; \text{pc}_1^0; \psi_1 \vdash \Delta[\text{tid}_1] \vdash \text{pc}_1^1; \text{pc}_1^0; \psi_1
$$

$$
\Gamma; \text{pc}_2^2; \text{pc}_2^1; \text{pc}_2^0; \psi_2 \vdash \Delta[\text{tid}_2] \vdash \text{pc}_2^1; \text{pc}_2^0; \psi_2
$$

for some $\psi_1, \psi_2$ such that $T = \psi_1 \sqcup \psi_2$. By Lemma 3 we know

$$
\Gamma; \text{pc}_1^1; \text{pc}_1^0; \text{psi}_1 \vdash \text{loc}_1 := v_1 \vdash \text{pc}_1^1; \text{pc}_1^0; \psi_1
$$

$$
\Gamma; \text{pc}_2^1; \text{pc}_2^0; \psi_2 \vdash \text{loc}_2 := v_2 \vdash \text{pc}_2^1; \text{pc}_2^0; \psi_2
$$

By inversion, we know that $\Gamma[\text{loc}] = \text{ref}_{\psi}^\tau$, $\psi_1 \subseteq \psi$, and $\psi_2 \subseteq \psi$. Then $T = \psi_1 \sqcup \psi_2 \subseteq \psi$ so $T = \psi$ but by well-formedness of reference types we know that $T \neq \psi$, a contradiction. So $\text{loc}_1 \neq \text{loc}_2$.

Additionally, with the same reasoning we know that $\Gamma[\text{loc}] = \text{ref}_{\psi'}^\tau'$ where $\psi \neq \psi'$. This implies that the updates occur in different equivalence classes of $P$, and thus $[H, H_1, H_3] \equiv_{P_i(\Gamma')} [H, H_2, H_3]$—namely, $T_1 \equiv_{P_i(\Gamma')} T_2$, as needed.
Case: no read-write races. Let $\Delta[tid_1] = E_1[\text{loc} := v_1]$ and $\Delta[tid_2] = E_2[\text{loc} := v_2]$ such that

$$\langle S, H, \Delta_1 \rangle = \langle S, H[\text{loc} \mapsto v_1], \Delta[tid_1 \mapsto E_1[\text{skip}]] \rangle$$

$$\langle H_2, \Delta_2 \rangle = \langle S, H, \Delta[tid_2 \mapsto E_2[H[\text{loc} := v_2]]] \rangle$$

It suffices to show $\text{loc} \neq \text{loc}^\prime$ such that

$$\langle S, H_3, \Delta_3 \rangle = \langle S, H[\text{loc} \mapsto v_1], \Delta[tid_1 \mapsto E_1[\text{skip}][\text{tid} \mapsto E_2[H[\text{loc} := v_2]]] \rangle$$

Assume $\text{loc} = \text{loc}^\prime$. Using similar reasoning to the case of write-write races above, we know that

$$\Gamma; pc_a^1; pc_c^1; pc_3; \psi_1 \vdash loc := v_1 \vdash pc_a^1; pc_3; \psi_1'$$

$$\Gamma; pc_a^2; pc_c^2; \psi_2 \vdash loc := v_2 \vdash pc_a^2; pc_3; \psi_2'$$

$$\Gamma[\text{loc}] = ref^\psi_1 \tau$$

where $\top = \psi_1 \sqcup \psi_2$, $\psi_1 \sqsubseteq \psi$, and $\top \neq \psi_2 \sqcup \psi$ by inversions on configuration, statement and expression typing respectively. Then

$$\psi_2 \sqsubseteq \psi_2 \sqcup \psi$$

$$\psi_1 \sqsubseteq \psi_2 \sqcup \psi$$

$$\top = \psi_1 \sqcup \psi_2 \sqsubseteq \psi_2 \sqcup \psi$$

So $\top = \psi_2 \sqcup \psi$ but we know above that $\top \neq \psi_2 \sqcup \psi$, a contradiction. Additionally, we know $\Gamma[\text{loc}^\prime] = ref^\psi_2 \tau'$ where $\psi \neq \psi'$. This implies $[H, H_1, H_3] \equiv \Gamma[\text{loc}^\prime]$, or $T_1 \equiv \Gamma[\text{loc}^\prime] T_2$ as needed.

Case: recv. Let

$$\Delta[tid_1] = E_1[\text{send} v \text{ to loc}]$$

$$\Delta[tid_2] = E_2[\text{recv loc(x)} \{s\}]$$

$$\langle S, H, \Delta_1 \rangle = \langle S, H[\text{loc} := v_1], \Delta[tid_1 \mapsto E_1[\text{skip}]][\text{tid} \mapsto E_2[H[\text{loc} := v_2]]] \rangle$$

$$\Delta[tid_3] = E_3[\text{send} v' \text{ to loc'}]$$

$$\Delta[tid_4] = E_4[\text{recv loc'(y)} \{s'\}]$$

$$\langle S, H_2, \Delta_2 \rangle = \langle S, H, \Delta[tid_3 \mapsto E_3[\text{skip}]][\text{tid} \mapsto E_4[\text{loc'} \{y/v\}]] \rangle$$

There are three subcases:

Subcase: $\text{tid} \neq \text{tid}_1, \text{tid}_2 \neq \text{tid}_4, \text{loc} \neq \text{loc}'$. The communication events commute together.

Subcase: $\text{tid} \neq \text{tid}_1, \text{loc} = \text{loc'}$ (send contention). This case is impossible—by Lemma 3 we know

$$\Gamma; pc_a^1; pc_c^1; pc_3; \psi_1 \vdash \text{send} v \text{ to loc } \vdash pc_a; pc_1; \psi_1'$$

$$\Gamma; pc_a^2; pc_c^2; pc_3; \psi_2 \vdash \text{send} v \text{ to loc } \vdash pc_a; pc_1; \psi_2'$$

and by inversion

$$\Gamma[\text{loc}] = \text{chan}_v \Psi_1 \Psi_2 \tau$$

$$\top = \psi_1 \sqcup \psi_3$$

$$\psi_1 \subseteq \Psi_1 \sqcap \psi_3 \subseteq \Psi_1 \sqcup \Psi_1 \sqcup \Psi_1$$

$$\psi_3 \subseteq \Psi_1 \sqcap \psi_3 \subseteq \Psi_1 \sqcup \Psi_1 \sqcap \Psi_1$$

$$\top = \psi_1 \sqcup \psi_3 \subseteq \Psi_1 \sqcup \Psi_1 = \Psi_1$$

thus $\top = \Psi_1$, but well-formedness of types $\top \neq \Psi_1$, a contradiction.

Subcase: $\text{tid} \neq \text{tid}_4, \text{loc} = \text{loc'}$ (receive contention). This case is impossible. We know

$$\Gamma; pc_a^1; pc_c^1; pc_3; \psi_1 \vdash \text{send} v \text{ to loc } \vdash pc_a; pc_1; \psi_1'$$

$$\Gamma; pc_a^2; pc_c^2; pc_3; \psi_2 \vdash \text{send} v' \text{ to loc' } \vdash pc_a; pc_1; \psi_2'$$
Then we know

\[ \Gamma[\text{loc}] = \text{chan}_{T} \Psi_{1} \Psi_{2} \tau \]
\[ \tau = \psi_{2} \sqcup \psi_{4} \]
\[ \psi_{2} \subseteq \Psi_{1} \sqcap \psi_{2} \subseteq \Psi_{1} \sqcup \Psi_{1} \]
\[ \psi_{4} \subseteq \Psi_{1} \sqcap \psi_{4} \subseteq \Psi_{1} \sqcup \Psi_{1} \]
\[ \tau = \psi_{2} \sqcup \psi_{4} \sqsubseteq \Psi_{1} \sqcup \Psi_{1} = \Psi_{1} \]

thus \( \tau = \Psi_{1} \) but by well-formedness of types \( \tau \neq \Psi_{1} \), a contradiction.

The arguments for the impossibility of channel contention for \( \text{join} \) and \( \text{select} \) are similar to the one above. The only thing left to argue is that scheduler modifications from \( \text{select} \) of schedulers: given \( \xi, \xi' \) such that \( \xi \neq \xi' \), \( \pi_{1}(S(\xi))(\xi') = S(\xi') \) and \( \pi_{1}(S(\xi'))(\xi) = S(\xi) \).

\[ \text{Theorem 4 (Confluence of well-typed erasures):} \] Given well-typed \( \ell \)-erasure

\[ \Gamma \vdash (S, H, \Delta) \preceq_{\ell} (S_{11}, H_{11}, \Delta_{11}) \]
\[ (S_{11}, H_{11}, \Delta_{11}) \rightarrow^{*} (S_{m1}, H_{m1}, \Delta_{m1}) \]
\[ (S_{11}, H_{11}, \Delta_{11}) \rightarrow^{*} (S_{1n}, H_{1n}, \Delta_{1n}) \]

there exists \( (S_{mn}, H_{mn}, \Delta_{mn}) \) such that \( (S_{m1}, H_{m1}, \Delta_{m1}) \rightarrow^{*} (S_{mn}, H_{mn}, \Delta_{mn}) \) and \( (S_{1n}, H_{1n}, \Delta_{1n}) \rightarrow^{*} (S_{mn}, H_{mn}, \Delta_{mn}) \).

Furthermore, the traces \( T_{1}, T_{2} \) respectively induced by the two reduction series

\[ (S_{11}, H_{11}, \Delta_{11}) \rightarrow^{*} (S_{m1}, H_{m1}, \Delta_{m1}) \rightarrow^{*} (S_{mn}, H_{mn}, \Delta_{mn}) \]
\[ (S_{11}, H_{11}, \Delta_{11}) \rightarrow^{*} (S_{1n}, H_{1n}, \Delta_{1n}) \rightarrow^{*} (S_{mn}, H_{mn}, \Delta_{mn}) \]

are stutter-equivalent configuration typing—i.e. \( T_{1} \equiv_{P, (\Gamma)} T_{2} \).

\[ \text{Proof:} \] By repeated applications of Lemma 9 and Lemma 13. Figure 10 gives a visual representation of this. The traces \( T_{1} \) and \( T_{2} \) induced by any two reduction paths from \( (S_{11}, H_{11}, \Delta_{11}) \) to \( (S_{mn}, H_{mn}, \Delta_{mn}) \) are such that \( T_{1} \equiv_{P, (\Gamma)} T_{2} \).

\[ \text{Lemma 14 (Trace prefix consistency):} \] Given \( T_{1} \preceq_{P} T_{3} \) and \( T_{2} \preceq_{P} T_{3} \), then either \( T_{1} \preceq_{P} T_{2} \) or \( T_{2} \preceq_{P} T_{1} \).

\[ \text{Corollary 1 (Determinism of well-typed erased configurations):} \] Given well-typed erased configuration \( \Gamma \vdash (S, H, \Delta) \)
\[ (S, H, \Delta) \downarrow T_{1}, (S, H, \Delta) \downarrow T_{2} \], and partition \( P \) from configuration typing, either \( T_{1} \preceq_{P, (\Gamma)} T_{2} \) or \( T_{2} \preceq_{P, (\Gamma)} T_{1} \).

\[ \text{Proof:} \] Let the following reduction steps induce \( T_{1} \) and \( T_{2} \) respectively:

\[ (S, H, \Delta) \rightarrow^{*} (S_{11}, H_{11}, \Delta_{11}) \]
\[ (S, H, \Delta) \rightarrow^{*} (S_{11}, H_{11}, \Delta_{11}) \]

Fig. 10: Visual representation of the proof for Theorem 4, where \( C_{ij} = \langle S_{ij}, H_{ij}, \Delta_{ij} \rangle \).
By Theorem 4 there exists \((S_3, H_3, \Delta_3)\) such that
\[
\langle S_1, H_1, \Delta_1 \rangle \rightarrow^* \langle S_3, H_3, \Delta_3 \rangle \\
\langle S_2, H_2, \Delta_2 \rangle \rightarrow^* \langle S_3, H_3, \Delta_3 \rangle.
\]
Let these reduction steps induce traces \(T'_1\) and \(T'_2\) respectively. Then we know \(T_1 T'_1 \equiv_{P(\Gamma)} T_2 T'_2\) and \(T_1 \leq_{P(\Gamma)} T_1 T'_1\) and \(T_2 \leq_{P(\Gamma)} T_2 T'_2\). This implies \(T_1 \leq_{P(\Gamma)} T_2 T'_2\) and \(T_2 \leq_{P(\Gamma)} T_1 T'_1\), and by Lemma 14 either \(T_1 \leq_{P(\Gamma)} T_2\) or \(T_2 \leq_{P(\Gamma)} T_1\), as needed.

We can now combine the simulation and determinism arguments to prove our main result.

**Theorem 5 (Observational determinism modulo select):** Given adversary label \(\ell\), let \(\Delta = [0 \mapsto s]\) and \(\Gamma \vdash \langle S_1, H_1, \Delta \rangle\) and \(\Gamma \vdash \langle S_2, H_2, \Delta \rangle\) such that \(S_1 \sim_\ell S_2\) and \(H_1 |\ell = H_2 |\ell\). Then for any traces \(T_1, T_2\) such that \(\langle S_1, H_1, \Delta \rangle \downarrow T_1\) and \(\langle S_1, H_1, \Delta \rangle \downarrow T_2\), either \(T_1 \leq_{P(\Gamma)} T_2\) or \(T_2 \leq_{P(\Gamma)} T_1\).

**Proof:** Let \(S_e\) be an “erased” scheduler such that \(S_e \sim_\ell S_1\) and \(S_e \sim_\ell S_2\). (It does not matter what \(S_e\) does for high select statements, as the erased semantics does not use it for those statements.) Let \(H_e\) and \(\Delta_e\) form an \(\ell\)-erasure of both configurations such that
\[
\Gamma \vdash \langle S_1, H_1, \Delta_1 \rangle \leq_\ell \langle S_e, H_e, \Delta_e \rangle \\
\Gamma \vdash \langle S_2, H_2, \Delta_2 \rangle \leq_\ell \langle S_e, H_e, \Delta_e \rangle.
\]
By Theorem 3, there exists \(\langle S_e, H_e, \Delta_e \rangle \downarrow T'_1\) and \(\langle S_e, H_e, \Delta_e \rangle \downarrow T'_2\) such that \(T_1 \equiv_{P(\Gamma)} T'_1\) and \(T_2 \equiv_{P(\Gamma)} T'_2\). By Corollary 1, either \(T'_1 \leq_{P(\Gamma)} T'_2\) or \(T'_2 \leq_{P(\Gamma)} T'_1\) so \(T_1 \leq_{P(\Gamma)} T_2\) or \(T_2 \leq_{P(\Gamma)} T_1\) as needed. 

\[\blacksquare\]