Securing Timing Channels at Runtime

Paritosh Shroff Scott F. Smith
The Johns Hopkins University
{pari,scott}@cs.jhu.edu

Abstract

We propose a general purpose runtime framework to secure timing channels. Our technique supports higher-order function invocations and computations looping on secret data, features which none of the existing approaches fully allow. We provably eliminate external and internal timing channels in both sequential and concurrent settings, in presence of deterministic as well as nondeterministic schedulers. There is a price to be paid, however – the high computation may have to be disrupted; the low computation is nevertheless guaranteed to be unaffected. We illustrate how our approach can be realized on standard computing platforms.

1. Introduction

Secure information flow analysis aims to prevent the flow of information from secure data in a program to insecure entities. A variety of covert channels [20] for insecure flow of information have been classified in the literature; Sabelfeld and Myers [37] provides a good survey. In this paper we address timing (and termination) channels.

A timing attack is a form of side-channel attack [48] wherein the attacker attempts to gain secret information by analyzing the execution time of a program; the corresponding channel of information leakage is known as a *timing channel*. Timing channels have been used to obtain secret keys of cryptographic systems by exploiting weaknesses in their implementations [17, 6], as opposed to the underlying mathematics. Various approaches and countermeasures have been proposed to eliminate or mitigate such covert attacks, falling into several categories: algorithm specific (e.g. RSA blinding, dummy extra reduction in Montgomery algorithm, quantization of RSA decryptions [6]), hardware based (e.g. fuzzy time [11], clock randomization [16], randomization of instruction set execution and/or register usage [24, 25], information flow analysis of synchronous (clocked) hardware circuits [18]), ad hoc [13], and programming language-based [43, 1]. Language-based techniques tend to be the most general, in that they are not tied to any specific algorithm or implementation. This paper presents a new language-based approach to secure timing channels.

Timing channels reveal classified information by manifesting *high* data as timing variations in *low* observable events. Timing channels are further classified as *external* and *internal* depending on the nature of those events – *external* channels are externally observable events such as the execution time of a program and the time gap between low outputs, while *internal* channels are internally observable events like the interleavings of concurrently running threads.

A substantial body of work [43, 38, 47, 34, 3] addresses internal timing channels. However, the only known language-based technique to secure both external and internal timing channels is the static method of program transformation by cross-copying branch slices [1]. This technique has limited expressiveness in that no looping executions on high data are allowable. In fact the execution time of computations looping on secret data is inherently tied to that secret data, and it does not seem likely to secure such computations by purely static approaches. Additionally, none of the existing approaches support higher-order functions. Higher-order function application is also dynamic in how the code to be executed is decided only at runtime, thus making it harder for static approaches to secure them as the exact code to be secured is itself not known statically. Dynamic dispatch in object-oriented languages presents a similar challenge. In fact all existing language-based approaches to eliminate timing channels are based on static analysis of programs. Timing behavior however by its very nature is a dynamic property, and runtime techniques hold significant promise – a runtime system has direct access to the running time of the program and can use this information to good advantage, for example by altering running times just enough to eliminate the insecure cases. Static techniques can only make approximate models of the timing behavior of programs, and hence must be inherently more conservative than a runtime one.

In this paper we propose a general purpose runtime framework to secure timing channels for a language which includes higher-order functions and looping computations on secret data, features which none of the existing approaches fully support – [41, 5, 10] do allow while-loops with secret guards but only under severe restrictions. We show how both external and internal timing channels are provably eliminated in both sequential and concurrent settings, in the presence of deterministic as well as nondeterministic schedulers, modulo the effect of caches [14, 2] and resource exhaustion [37] on timing behavior of programs. There is a price to be paid, however – the high computation may have to be disrupted; the low computation is nevertheless guaranteed to be unaffected. Section 2 provides an informal overview of our technique which elaborates on these issues.

We formulate three runtime systems: λ^{sync}_{seq}, the core system, which addresses external timing channels in sequential programs, and two extensions to it, $\lambda^{sync}_{conc_{\mathcal{D}}}$ and $\lambda^{sync}_{conc_{\mathcal{N}}}$, to address both external and internal timing channels in a concurrent setting, under arbitrary deterministic and nondeterministic schedulers, respectively. We prove a strong noninterference [9] result for λ^{sync}_{seq}, and scheduler-independent strong noninterference and probabilistic noninterference [38] results for $\lambda^{sync}_{conc_{\mathcal{D}}}$ and $\lambda^{sync}_{conc_{\mathcal{N}}}$, respectively. Our probabilistic noninterference result employs discrete probability distributions, as opposed to the continuous probabilities of [38], allowing for a more direct proof technique.

2. Overview

Consider the following program in Caml [22]-like syntax:

```
let len = λz. (* number of digits in z *) in
let multiplyBigInts =
    λa. λb. let a_len = len a in let b_len = len b in
            let i = ref 0 in let j = ref 0 in
            while (!i < a_len) {
              while (!j < b_len) { ...; j++ }; ...; i++ }; ...
in
let modBigInts =
    λc. λd. let c_len = len c in let d_len = len d in ...    in
let h = inputBit^high() in let r = ref 5^high in                    (1)
let x = inputBigInt^low() in let n = inputBigInt^low() in
let x_len = len x in let n_len = len n in
output^low("p is about to start execution");
if_p (h == 1) then
  r := modBigInts (multiplyBigInts x x) n
else
  r := modBigInts x n;
output^low("p just finished execution");
```

This program is inspired by the square-and-multiply algorithm – an efficient algorithm for modular exponentiation of large numbers used in many cryptographic implementations including RSA [33]. For simplicity of presentation we employ the naive $\mathcal{O}(n^2)$ schoolbook algorithms for multiplication and modular division of big integers; the exact algorithm used is not relevant here. The program assumes the presence of arbitrary precision arithmetic[1]. It leaks information because it changes its timing behavior based on the high input to the variable h. The *program point identifier* p on if_p labels this conditional branch point. The variable h holds a single high bit, that of the RSA private exponent for example, while x and n hold arbitrary precision low integers, say the message to be decrypted and RSA's modulus, respectively. The attacker's goal is to find the value of h by externally observing the time lag between the two low outputs across multiple program runs.

	Run 1	*Run 2*
h	1^{high}	0^{high}
$!r$, initial	5^{high}	
x, n, x_{len}, n_{len}	i_1^{low}, i_2^{low}, 20^{low}, 15^{low}	
branch taken at p	then	else
running time of p	2860 units	1275 units
$!r$, final	$(x^2 \mod n)^{high}$	$(x \mod n)^{high}$
h leaked?	**yes**	
$i_1 = 29837429873928930498$, $i_2 = 908028734093623$		

Figure 1. Example (1): A Set of Raw Runs

Figure 1 tabulates the traces of a set of runs of program (1) with differing high values for h, but the same low values of x and n. Recall the computational time complexities of *multiplyBigInts* and *modBigInts* are quadratic in the lengths of their inputs, so for a given x and n, the then-branch should take more than double the time of the else-branch, a fact borne out in Figure 1. (Note, the numbers denoting the running times in Figure 1, and in the discussion to follow, have been chosen to denote the expected *relative* running times of the respective branches; hence the use of 'unit' as the metric for measurement of time. In our formal semantics, the number of small-steps is used to express time.) Consequently, an attacker who knows the structure of program (1) and observes that the lag between the low outputs is less than half in run 2 as compared to run 1, can correctly infer that the then-branch was taken in the first run while the else in the second, that is, h was 1 in run 1, and 0 in run 2. The delay between the low outputs constitutes an external timing channel; our goal is to secure it by decoupling the time lag from the value of h.

[1] For technical simplicity we do not have variable length arrays for implementing big integers in the formal language syntax presented in Section 3; it should be straightforward to incorporate them if desired. We also do not formally support while-loops, but we do have higher-order functions with mutable state so loops are encodable via the 'tying the knot' technique. Lastly, the input and output statements used in the example are not a part of our official language syntax; they can, however, be easily incorporated, as is discussed in Section 8.

	Run 1′	*Run 2′*
⋮	⋮	⋮
running time of p	2860 units	
$!r$, final	$(x^2 \mod n)^{high}$	$(x \mod n)^{high}$
h leaked?	**no**	

Figure 2. Example (1): Optimally Synchronized Runs

How to secure the external timing channel? The attacker can be deceived by forcing the running time of the branching statement to be the same regardless of which branch was taken. Figure 2 tabulates runs $1'$ and $2'$, which are variants of runs 1 and 2 forced to synchronize in this manner. Synchronization can be implemented by setting a timer for 2860 units – the greater of the running times of the branching statement p in runs 1 and 2 – immediately after resolving the guard $(h == 1)$ but just before beginning execution of the selected branch. If the execution of the selected branch finishes before the timer goes off, as is the case in run $2'$, nops can be performed until the allotted time is over, thus padding the computation of p ensuring it takes exactly 2860 units. Both the then- and else-branches thus take the same time, and we say the synchronized running time (in short, the *sync time*) of p is 2860 units. Thus, with identical timing the attacker can learn nothing about the value of h by comparing runs $1'$ and $2'$.

The sync time of 2860 units for the branching statement p is in fact optimal, denoting the upper bound on the running time of p for $x = i_1$ and $n = i_2$. The knowledge of the structure of program (1) and the running times of p in runs 1 and 2 can be used to obtain the optimally synchronized runs $1'$ and $2'$. In general, however, it is impossible to always accurately determine the upper bound on the running time of a given piece of code.

Our general solution Taking cue from the above example where we used the knowledge gained from the "test" runs 1 and 2 to generate the leak-free synchronized runs $1'$ and $2'$, we propose a twofold solution to secure external timing channels: a) *Pre-deployment*: Estimate as accurately as possible the upper bounds on the running times of all branching statements with high guards, by a combination of rigorous testing on a variety of inputs and/or programmer input, and b) *Post-deployment*: Use the pre-deployment estimates as the sync times for the corresponding branching statements, in conjunction with a realtime fallback technique (presented below) for securely handling situations where the sync times are found to be less than needed.

How are sync times determined? The time complexities of the then- and else-branches can be approximated by a measurement-based approach – run program (1) on a variety of inputs for h, x, and n, measuring the corresponding running times in each of the runs, and then correlate the inputs with the measured running times of the corresponding branches. We know the computational time complexities of the functions *multiplyBigInts* and *modBigInts* are quadratic in the lengths of their inputs, that is, $\mathcal{O}(a_{len} * b_{len})$ and $\mathcal{O}(c_{len} * d_{len})$ respectively, implying the running times of the then- and else-branches of program (1) are in

turn quadratic in the lengths of x and n; say the functions approximating the running times of the then- and else-branches of program (1) are $\phi_{_T}(x_{len}, n_{len}) = 9x_{len}n_{len} + 5x_{len} + 4n_{len} + 318$ and $\phi_{_F}(x_{len}, n_{len}) = 4x_{len}n_{len} + 3x_{len} + n_{len} + 243$ respectively, each parameterized on the lengths of x and n denoted by x_{len} and n_{len} respectively. Observe for some x_{len} and n_{len} the estimated running time of the then-branch, $\phi_{_T}(x_{len}, n_{len})$ units, is always greater than that of the else-branch, $\phi_{_F}(x_{len}, n_{len})$ units; hence we can use $\phi_{_T}$ as the sync timer function.

One can imagine the programmer having supplied the computational complexities for the then- and else-branches, with only the constant factors in $\phi_{_T}$ and $\phi_{_F}$ determined empirically above. It also may be possible to automate the process of computing the sync timer functions by employing curve fitting techniques on top of empirical data. The problem of estimating time bounds has been studied extensively in the field of Worst-Case Execution Time (WCET) analysis [45]. In general any methodology which delivers the desired accuracy may be used to derive the sync timer functions – the security guarantees of our system are independent of the precision of the sync times; however the closer to optimal the sync times can be made to be, the better would be the practical usefulness of our system.

	Run 3	*Run 4*
h	1^{high}	0^{high}
$!r$, initial	5^{high}	
x, n, x_{len}, n_{len}	i_3^{low}, i_4^{low}, 25^{low}, 20^{low}	
branch taken at p	then	else
running time of p	4705 units	2095 units
$!r$, final	$(x^2 \mod n)^{high}$	$(x \mod n)^{high}$
h leaked?	**yes**	
$i_3 = 502823472010871287812032 4$ $i_4 = 62598712387561314011$		

Figure 3. Example (1): Another Set of Raw Runs

	Run 3′	*Run 4′*
⋮	⋮	⋮
running time of p	$\phi_{_T}(x_{len}, n_{len})$ = 5023 units	
$!r$, final	$(x^2 \mod n)^{high}$	$(x \mod n)^{high}$
h leaked?	**no**	

Figure 4. Example (1): Synchronized Runs, Adequate Sync Time

What if the sync time is more than the raw running time? Prior to deployment one must strive to tune the sync timer functions to be as close to optimal as possible, so as to minimize the padding of synchronized computations – besides the needless slow-down of program computation, overestimated sync times have no undesirable side-effects. Consider runs 3 and 4 tabulated in Figure 3, with the corresponding synchronized runs $3'$ and $4'$ shown in Figure 4. For $x = i_3$ and $n = i_4$, $\phi_{_T}$ provides an overestimation of the raw running times for both of p's branches. Hence in Figure 4, each of the branches completes its execution before the sync timer goes off; the residual sync time is padded away with nops as before. The attacker again learns nothing of h from timing information gleaned from runs $3'$ and $4'$. The parameterization of the sync timer functions on low input values is an important aspect of our system.

The program transformation approach by Agat [1] exhibits the same expressiveness on this program by a static technique of cross-copying slices of branches, which implicitly encodes the above parameterization in the transformed code. Agat's technique, however, would not be applicable if the functions *multiplyBigInts* and *modBigInts* were part of a library where their source code was not available; our dynamic synchronization technique factors in only the running times and does not need the source code, and so would remain applicable to such library calls. Our technique is more general in other ways as well, a topic that will be taken up later in this section.

What if the sync time is less than the raw running time? Say the programmer had expected program (1) to be used mainly on inputs of lengths less than or equal to 32 digits for x and n; hence, prior to deployment she had ensured that the sync timer function $\phi_{_T}$ was appropriate for all inputs to x and n of lengths 32 digits or less. Recall the lengths of i_3 and i_4 in runs 3 and 4 were both less than 32 digits.

	Run 5′	*Run 6′*
h	0^{high}	1^{high}
$!r$, initial	5^{high}	
x, n, x_{len}, n_{len}	i_5^{low}, i_6^{low}, 40^{low}, 30^{low}	
branch taken at p	else	then
running time of p†	$\phi_{_T}(x_{len}, n_{len})$ = 11438 units	
branch execution aborts?	*no*	***yes***‡
$!r$, final	$(x \mod n)^{high}$	5^{high}‡
h leaked?	**no**	
$i_5 = 3321654873210031296870149807841203498791$ $i_6 = 10321651060648941203065498432 1$ †raw running time of then-branch is 12003 units, and else-branch is 5837 units. ‡computation is aborted after the allotted sync time is over leaving the logically incorrect value 5^{high} in $!r$.		

Figure 5. Example (1): Synchronized Runs, Inadequate Sync Time

Now consider the synchronized runs $5'$ and $6'$ tabulated in Figure 5; the sync time of $\phi_{_T}(x_{len}, n_{len}) = 11438$ units for $x = i_5$ and $n = i_6$, is less than the corresponding raw running time of the then-branch but is greater than that of the else-branch. The run $5'$ terminates with $(x \mod n)^{high}$ in $!r$; however, if the then-branch in run $6'$ *were* allowed to complete, the attacker would observe a timing difference between runs $6'$ and $5'$, and thus learn the value of h. Hence in order to bluff the attacker the computation of the then-branch is simply *aborted* in run $6'$ when the allotted sync time is over, and the subsequent low output performed. Once aborted, the high computation (the portion of the program computation manipulating high data) has gone logically wrong, and the program has entered a *bluff computation* state, the purpose of which is to hide the abortion from a low observer, and to keep him from gaining information on h.

As the execution of the then-branch was terminated before the assignment to r could take place, the old and now logically incorrect value 5^{high} remains in $!r$ at the end of run $6'$; this value is to be viewed as a "placeholder" in that it is not useful but prevents information leakage. Note, only branching statements with high guards are ever synchronized in our system, implying only high branch computations can ever be aborted; hence placeholder values, which are a result of such abortions, are guaranteed to be always high.

The bluffing technique can be viewed as a generalization of the lenient execution model [7] for out-of-bounds array indices.

Bluff computation is only a fallback, not a norm Given that the programmer had not expected an input of length as large as 40 digits for x, i_5 could have been a malicious input fed by an attacker trying to covertly gain classified information on h. In fact the programmer should have disallowed invalid values greater than 32 digits for x and n in the first place by inserting

```
if (x_len > 32 || n_len > 32) then
  output^low("invalid input"); exit 1;
```

immediately after the line 'let $x_{len} = len\ x$ in let $n_{len} = len\ n$ in' in program (1). This change would eliminate the need for bluff computation due to too-large an input.

In general, if the programmer is careful enough to disallow all spurious low values from reaching timing-sensitive pieces of code with running times dependent on low data, he can be quite certain the bluff computation will not be needed for them upon deployment.

Handling computations looping on high data In program (1) the value of $!r$ was never used. Now consider program (2), which is program (1) followed by the addendum below.

$$
\begin{array}{l}
\ldots \\
\mathsf{let}\ y = !r\ \mathsf{in}\ \mathsf{let}\ y_{\mathit{len\text{-}max}} = n_{\mathit{len}}\ \mathsf{in} \\
\mathsf{let}\ h' = \mathsf{inputBit}^{\mathit{high}}()\ \mathsf{in}\ \mathsf{let}\ r' = \mathsf{ref}\ 7^{\mathit{high}}\ \mathsf{in} \\
\mathsf{output}^{\mathit{low}}(\text{“}\mathsf{p}'\text{ is about to start execution”}); \\
\mathsf{if}_{\mathsf{p}'}\ (h' == 1)\ \mathsf{then} \\
\quad r' := \mathit{modBigInts}\ (\mathit{multiplyBigInts}\ y\ y)\ n \\
\mathsf{else} \\
\quad r' := \mathit{modBigInts}\ y\ n; \\
\mathsf{output}^{\mathit{low}}(\text{“}\mathsf{p}'\text{ just finished execution”});
\end{array}
\tag{2}
$$

Note, y is high in program (2) as r is set under the high guard $(h == 1)$ in program (1). Also note the computations of both *modBigInts* (*multiplyBigInts* $y\ y$) n and *modBigInts* $y\ n$ are looping on the lengths of y and n, so the running time of p' is parametric in y and n. We use ϕ_T as the sync timer function for p' as well; however, parameterizing the sync time of p' on the length of y would reveal the secret value of h – the lag between the low outputs on each side of p' would then depend on the value of y which in turn depends on the secret bit in h. Recall the sync times of p in runs $3'$–$6'$ were solely parameterized on the low lengths of the low values x and n; hence despite the difference in the sync times in Figures 4 and 5 no information was leaked. In order to prevent information leakage the sync time of p' must be parameterized exclusively on low values.

Note the only values possible for y in program (2) are either 5^{high}, $(x \mod n)^{\mathit{high}}$, or $(x^2 \mod n)^{\mathit{high}}$, implying its value can be no bigger than that of n, that is, the length of y is equal to the length of n in the worst case. Given this publicly known worst-case length for y we can conservatively parameterize the sync time of p' on $y_{\mathit{len\text{-}max}}$ (which is equal to n_{len}) and n_{len}. Figure 6 tabulates a set of such synchronized runs for program (2) as extensions to run $6'$. The attacker again learns nothing since runtimes are uniform.

	Run $7'$	*Run* $8'$
h	1^{high}	
$!r$, initial	5^{high}	
x, n, x_{len}, n_{len}	i_5^{low}, i_6^{low}, 40^{low}, 30^{low}	
branch taken at p	then	
running time of p	$\phi_T(x_{\mathit{len}}, n_{\mathit{len}})$ = 11438 units	
branch execution aborts?	**yes**	
$!r$, final	$5^{\mathit{high}\dagger}$	
y, $y_{\mathit{len\text{-}max}}$	$5^{\mathit{high}\dagger}$, n_{len}	
h'	1^{high}	0^{high}
$!r'$, initial	7^{high}	
branch taken at p'	then	else
running time of p'	$\phi_T(y_{\mathit{len\text{-}max}}, n_{\mathit{len}})$ = 8688 units	
branch execution aborts?	no	
$!r'$, final	$(y^2 \mod n)^{\mathit{high}} = 25^{\mathit{high}\ddagger}$	$(y \mod n)^{\mathit{high}} = 5^{\mathit{high}\ddagger}$
h or h' leaked?	**no**	

†high placeholder value ‡garbled high data

Figure 6. Example (2): Worst-Case Sync Time

The above example illustrates our general technique for preventing timing leaks in presence of computations looping on secret data: parameterize the corresponding sync times over low conservative approximations of those high data.

Agat [1] does not allow looping computations with high guards. While-loops with secret guards are supported in [41, 5], but only in concurrent settings, and only if such loops are not followed by low events. Our technique has the advantage of support for high looping computations, in both sequential and concurrent settings, and without the aforementioned restriction.

Our approach of parameterizing the sync times of high looping computations with the worst-case low approximations of the corresponding high guards is analogous to the "worst-case principle" proposed by Agat and Sands [2]; the latter, however, focuses on rewriting algorithms with low termination conditions corresponding to the worst-case executions of such high loops, while we achieve a similar effect dynamically.

Placeholder values may garble high data Observe the final values of $!r'$ in runs $7'$ and $8'$ are logically incorrect due to the logical incorrectness of y. In general once a branch computation is aborted the resulting placeholder values may pollute the high computation with garbled high data – taking the analogy of program (1) to an RSA implementation, the decrypted message may be garbled. In general, an execution entering a bluff computation state may not be able to complete some high computation tasks due to garbled data, and programs must be written with this contingency in mind, not unlike how the case of "network down" needs to be accounted for programatically. Note, any high computation task not influenced by the garbled data will nonetheless function correctly.

A Higher-Order Example Examples (1) and (2) were simple first-order programs. Now consider the following higher-order variant of program (1), where '_' is a shorthand for any variable not found free in the body of the corresponding function.

$$
\begin{array}{l}
\ldots \\
\mathsf{if}_{\mathsf{p}''}\ (h == 1)\ \mathsf{then} \\
\quad f := \big(\lambda\_.\, r := \mathit{modBigInts}\ (\mathit{multiplyBigInts}\ x\ x)\ n\big) \\
\mathsf{else} \\
\quad f := (\lambda\_.\, r := \mathit{modBigInts}\ x\ n); \\
\mathsf{output}^{\mathit{low}}(\text{“}\mathsf{p}'''\text{ is about to start execution”}); \\
(!f)\ ()_{\mathsf{p}'''}; \\
\mathsf{output}^{\mathit{low}}(\text{“}\mathsf{p}'''\text{ just finished execution”});
\end{array}
\tag{3}
$$

The program point identifier p''' denotes the corresponding function application statement; function application in the presence of first-class functions is a form of branching – the code to be executed next depends on the function flowing into the application site. Analogously, the running times of such function application statements depend on the exact functions flowing into them, just as the running times of branching statements depend on the actual guard flowing into them. Hence, akin to runs 1–4, program (3) will also leak the value of h by exhibiting variation in the running time of p''' based on h's value. The synchronization technique discussed above for conditional branching statements is applicable here as well; ϕ_T could be used as the sync timer function for p'''.

We are not aware of any existing techniques for securing timing channels which support higher-order functions; we believe this is a new contribution of our approach.

Securing timing channels in presence of concurrency We now show how our technique also may be used to secure timing channels in concurrent programs. Consider the following program in Caml-like syntax, adapted from [47],

$$
\begin{array}{l}
\mathsf{let}\ \mathit{spin} = \lambda n.\, \mathsf{let}\ i = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{while}\ (!i < n)\ \{\, i\mathsf{++}\,\}\ \mathsf{in} \\
\mathsf{let}\ l = \mathsf{ref}\ 0^{\mathit{low}}\ \mathsf{in} \\
\left(\begin{array}{l} \mathsf{if}_{\mathsf{p}''''}\ (h == 1)\ \mathsf{then}\ \mathit{spin}(500^{\mathit{low}})\ \mathsf{else}\ (); \\ \mathsf{output}^{\mathit{low}}(!l) \end{array}\ \middle\|\ \begin{array}{l} \mathit{spin}(50^{\mathit{low}}); \\ l := 1^{\mathit{low}} \end{array}\right)
\end{array}
\tag{4}
$$

$$
\begin{array}{lcll}
n & \in & \mathbb{N} & \textit{small-step semantics up-counter}\\
d & \in & \mathbb{N}\cup\{\infty\} & \textit{down-counter}\\
\mathsf{p} & & & \textit{program point identifier}\\
x & & & \textit{variable}\\
i & \in & \mathbb{Z} & \textit{integer}\\
b & ::= & \mathsf{true} \mid \mathsf{false} & \textit{boolean}\\
\mathit{loc} & & & \textit{heap location}\\
\oplus_a & ::= & + \mid - \mid * \mid / & \textit{arithmetic binary operator}\\
\oplus_r & ::= & < \mid > \mid == \mid != & \textit{relational binary operator}\\
\oplus & ::= & \oplus_a \mid \oplus_r & \textit{binary operator}\\
v & ::= & x \mid i \mid b \mid \lambda x.\, p \mid \mathit{loc} & \textit{value}\\
p & ::= & v \mid p \oplus p \mid \mathsf{let}\ x = p\ \mathsf{in}\ p & \textit{program (source syntax)}\\
 & & \mid \mathsf{if}^{v}_{\mathsf{p},\overline{v}}\ p\ \mathsf{then}\ p\ \mathsf{else}\ p \mid p\,(p)^{v}_{\mathsf{p},\overline{v}} & \\
 & & \mid \mathsf{ref}\ p \mid\ !p \mid p := p & \\
\mathrm{R} & ::= & \bullet \mid \mathrm{R} \oplus p \mid v \oplus \mathrm{R} \mid \mathsf{let}\ x = \mathrm{R}\ \mathsf{in}\ p & \textit{reduction context}\\
 & & \mid \mathsf{if}^{v}_{\mathsf{p},\overline{v}}\ \mathrm{R}\ \mathsf{then}\ p\ \mathsf{else}\ p \mid \mathrm{R}\,(p)^{v}_{\mathsf{p},\overline{v}} \mid v\,(\mathrm{R})^{v}_{\mathsf{p},\overline{v}} & \\
 & & \mid \mathsf{ref}\ \mathrm{R} \mid\ !\mathrm{R} \mid \mathrm{R} := p \mid v := \mathrm{R} & \\
s & ::= & \langle p\rangle^{v} & \textit{synchronization construct}\\
e & ::= & p \mid \mathrm{R}[s] & \textit{expression (runtime syntax)}\\
\mathrm{H} & : & \{\overline{\mathit{loc}}\} \to \{\overline{v}\} & \textit{heap (memory)}\\
\mathsf{C} & ::= & \big(\mathrm{H}, e\big)_{d} & \textit{runtime configuration}\\
 & & \overbrace{\mathbb{Z}\times\ldots\times\mathbb{Z}}^{n}\to\mathbb{N} & \\
\phi^{n} & : & & \textit{sync timer function}\\
\Phi & : & \{\overline{\mathsf{p}}\} \to (\phi^{n} \vee \varnothing) & \textit{table of sync timer functions}\\
\hat{v} & ::= & i \mid b & \textit{integer/boolean value}
\end{array}
$$

Figure 7. λ^{sync}_{seq}: Syntax Grammar

where the infix operator $\|$ denotes parallel composition. The variable h holds a single high bit, while the reference variable l is shared between the program fragments to be executed concurrently.

Observe the raw running time of the then-branch in the left fragment is considerably more than that of $\mathit{spin}(50^{low})$ in the right fragment; the converse is true for the else-branch. Hence under most schedulers (e.g. round-robin), if h is 1 then the assignment $l := 1^{low}$ is likely to precede $\mathsf{output}^{low}(!l)$, resulting in 1 being the likely output; conversely, if h is 0 the output is likely to be 0. In summary, h can affect the *internal timing behavior* of program (4), which in turn can affect the value of $!l$ that is output to a low observer, thus indirectly leaking the value of h. Such leaks due to variations in the internal timing behavior of concurrent programs are referred to as *internal timing leaks*, with the internal timing behavior constituting the *internal timing channel*. Note, unlike external timing leaks, the attacker need not have a stop-watch to record the timings of the low outputs, the values of the low outputs themselves reveal classified information. Hence internal timing channels are easier to secure than external timing channels – only the values of the low outputs need to be decoupled from high data to secure internal timing channels, whereas both the low outputs and their timings need to be insulated from secret information to close external timing channels.

Observe that the root of the timing leak in program (4) is the asymmetry in the execution times of the then- and else-branches of the branching statement p''''. Hence our dynamic synchronization technique will also secure the internal timing channel here. In general, the technique secures all internal and external timing channels in arbitrary concurrent programs, as is proved in Sections 5 and 6.

3. The λ^{sync}_{seq} Runtime System

We now formalize the λ^{sync}_{seq} language and define its semantics.

λ^{sync}_{seq} is a sequential language with mutable state, higher-order functions, conditional branchings and let-bindings. The language syntax appears in Figure 7. Note the overbar notation denotes comma-separated sequence of zero or more items, so for example $\overline{v}$ denotes some $v_1, \ldots, v_n$; the subscripted overbar notation denotes a sequence of fixed length as indicated by the subscript, so for example $\overline{v_k} = v_1, \ldots, v_k$ and $\{\overline{loc_k \mapsto v_k}\} = \{loc_1 \mapsto v_1, \ldots, loc_k \mapsto v_k\}$. The runtime heap H is a partial function from heap locations to values. Program point identifiers p are used to uniquely label conditional branching and function application sites. They are intended to be automatically generated, but we embed them in program syntax for technical convenience. Also we use the terms 'program point' and 'program point identifier' interchangeably throughout this paper.

The table of sync timer functions Φ associates each program point with either a sync timer function ϕ^n, or $\varnothing$ denoting lack of such a function – sync timer functions need to be defined only for timing-sensitive statements whose running times depend on high data. The sync timer function ϕ^n is parameterized on an n-tuple of integers and returns a natural number denoting the corresponding sync time; recall for example how the sync timer function ϕ_{τ} in Section 2 was parameterized on a pair of integers.

The subscripts $\overline{v}$ on the conditional branching and application sites must evaluate to low-security integral parameters during computation, and are then fed to the associated sync timer function.

The superscript v on the branching and application sites denotes the default placeholder value for the expression – in case the synchronized execution is aborted the associated default placeholder value is inserted in its place to allow the rest of the computation to proceed, thus hiding the abortion from a low observer. The default placeholder value can be any value of the same type as the computation it is intended to replace – this is ensured by our type system.

The synchronization construct $\langle p\rangle^v$ is a runtime construct denoting the synchronized computation of the enclosed program p with the default placeholder holder value v; $\langle p\rangle^v$ occurs only at runtime, and so we distinguish the runtime expressions e from source programs p.

The set of free variables is defined as follows,

Definition 3.1 (Free Variables)**.**

1. *(Value).*
   (a) *(Variable).* $\mathit{free}(x) = \{x\}$*; and*
   (b) *(Integer, Boolean, Heap Location).* $\mathit{free}(i) = \mathit{free}(b) = \mathit{free}(\mathit{loc}) = \emptyset$*; and*
   (c) *(Function).* $\mathit{free}(\lambda x.\, p) = \mathit{free}(p) - \{x\}$.
2. *(Program).*
   (a) *(Binary Operation).* $\mathit{free}(p \oplus p') = \mathit{free}(p) \cup \mathit{free}(p')$*; and*
   (b) *(Let).* $\mathit{free}(\mathsf{let}\ x = p\ \mathsf{in}\ p') = \mathit{free}(p) \cup (\mathit{free}(p') - \{x\})$*; and*
   (c) *(If).* $\mathit{free}\big(\mathsf{if}^{v}_{\mathsf{p},\overline{v_k}}\ p\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2\big) = \mathit{free}(p) \cup \mathit{free}(p_1) \cup \mathit{free}(p_2) \cup \mathit{free}(v_1) \cup \ldots \cup \mathit{free}(v_k) \cup \mathit{free}(v)$*; and*
   (d) *(App).* $\mathit{free}\big(p\,(p')^{v}_{\mathsf{p},\overline{v_k}}\big) = \mathit{free}(p) \cup \mathit{free}(p') \cup \mathit{free}(v_1) \cup \ldots \cup \mathit{free}(v_k) \cup \mathit{free}(v)$*; and*
   (e) *(Ref).* $\mathit{free}(\mathsf{ref}\ p) = \mathit{free}(p)$*; and*
   (f) *(Deref).* $\mathit{free}(!p) = \mathit{free}(p)$*; and*
   (g) *(Assign).* $\mathit{free}(p := p') = \mathit{free}(p) \cup \mathit{free}(p')$.
3. *(Synchronization Construct).* $\mathit{free}\big(\langle p\rangle^v\big) = \mathit{free}(p) \cup \mathit{free}(v)$.
4. *(Expression with a Synchronization Construct).* $\mathit{free}\big(\mathrm{R}[s]\big) = \mathit{free}(s) \cup (\mathit{free}(\mathrm{R}[x]) - \{x\})$*, where* $\mathrm{R}[s] = (\mathrm{R}[x])[s/x]$.
5. *(Heap).* $\mathit{free}(\mathrm{H}) = \bigcup_{loc \in dom(\mathrm{H})} \mathit{free}\big(\mathrm{H}(\mathit{loc})\big)$.
6. *(Heap, Expression).* $\mathit{free}(\mathrm{H}, e) = \mathit{free}(\mathrm{H}) \cup \mathit{free}(e)$.

We write $e[e'/x]$ to denote the capture-avoiding substitution of all free occurrences of x in e with e', and $\mathrm{H}[e/x]$ to denote the heap H' such that $\mathit{dom}(\mathrm{H}') = \mathit{dom}(\mathrm{H})$ and $\forall \mathit{loc} \in \mathit{dom}(\mathrm{H}).\ \mathrm{H}'(\mathit{loc}) = \mathrm{H}(\mathit{loc})[e/x]$. In addition we use $(\mathrm{H}, e)[e'/x]$ as shorthand for $\big(\mathrm{H}[e'/x], e[e'/x]\big)$, and the multi-substitution $\mathbf{X}[\overline{v_k/x_k}]$, where $\mathbf{X} ::= \mathrm{H} \mid e$, as shorthand for

$\big(\big((\mathbf{X}[v_1/x_1])[v_2/x_2]\big)\ldots\big)[v_k/x_k]$. Also we write "$e$ is a program" if $e = p$, for some p, "e is a value" if $e = v$, for some v, and "e has a sync construct" if there exists a R and a s, such that $e = \mathrm{R}[s]$.

For technical ease we sometimes view functions as a set of mappings from the elements in its domain to those in its range; so for example $loc \mapsto v \in \mathrm{H}$ is equivalent to asserting $\mathrm{H}(loc) = v$. The complement operator $\backslash$ on a set of mappings, say H, is defined as $\mathrm{H}\backslash loc = \{loc' \mapsto v' \mid loc' \mapsto v' \in \mathrm{H} \wedge loc \neq loc'\}$; the update operation is then defined as $\mathrm{H}[loc \mapsto v] = \mathrm{H}\backslash loc \cup \{loc \mapsto v\}$. Further for any binary relation $\mathcal{R}$ such that $\mathcal{R} \subseteq A \times B$, $A = \{\overline{a}\}$ and $B = \{\overline{b}\}$, we use the shorthand '$\overline{a_k}\ \mathcal{R}\ b$' for '$\forall 1 \leq i \leq k.\ a_i\ \mathcal{R}\ b$', and symmetrically, '$a\ \mathcal{R}\ \overline{b_k}$' for '$\forall 1 \leq i \leq k.\ a\ \mathcal{R}\ b_i$'.

Operational Semantics Figure 8 gives the syntax-directed small-step operational operational semantics for λ^{sync}_{seq}. The small-step reduction relation $\longrightarrow_\Phi$, parameterized by a table of sync timer functions Φ, relates heap/expression configurations $\big(\mathrm{H}, e\big)_d$. Here d is the *down-counter* and is decremented at each step; it denotes the maximum number of small-steps available for the expression e to complete its execution. If $d = \infty$, there is an unbounded number of available steps (we consider $(\infty - 1) = \infty$). The n-step reflexive and transitive closure of $\longrightarrow_\Phi$ is denoted as $\longrightarrow^n_\Phi$. We use the small-step counter n as our abstraction for execution time; however, our techniques are generalizable regardless of the granularity of the abstraction used.

The computation of program p starts in an initial configuration $\big(\emptyset, p\big)_\infty$. During the course of computation, if the execution time of a branching or application statement needs to be synchronized (*i.e.*, the premise $\Phi(\mathsf{p})(\overline{i_k}) = n$ holds in the IF-SYNC and APP-SYNC rules), the corresponding code is placed in a synchronization construct $\big(\langle p_i \rangle^v$ in IF-SYNC, and $\langle p[v/x] \rangle^{v'}$ in APP-SYNC$\big)$ with the associated default placeholder value (v and v', respectively); the down-counter is then set to the indicated sync time (n). Since $n \in \mathbb{N}$, synchronized computation is limited to a finite number of steps by the SYNC rule – when the down-counter reaches 0, the SYNC rule is no longer applicable as -1 is not a valid down-counter.

If the computation being synchronized finishes before the sync time is over, nop's are performed via the SYNC-PAD rule until the down-counter is 0, thus ensuring the overall synchronization takes exactly the sync time number of steps to complete from an external observer's point of view; the computed value is then unwrapped from the synchronization construct and the down-counter is reset to ∞ via the SYNC-DONE rule, while the default placeholder value is simply thrown away. However, if the computation under synchronization does not converge to a value in the allotted sync time, then it is aborted via the SYNC-ABORT rule. The default placeholder value is inserted in place of the unfinished computation, and the down-counter is reset to ∞ indicating the synchronization has completed and non-synchronized computation can resume.

Definition 3.2 (Canonical Configuration)**.** *A configuration $\big(\mathrm{H}, e\big)_d$ is canonical iff either e is a program and $d = \infty$, or e has a synchronization construct and $d \neq \infty$.*

Definition 3.3 (Canonical Derivation)**.** *A derivation $\big(\mathrm{H}, e\big)_d \longrightarrow^n_\Phi \big(\mathrm{H}', e'\big)_{d'}$ is canonical iff the configuration at each node of its derivation tree is canonical.*

An important aspect of our synchronization technique is that only top-level branching and function application statements are synchronized, as indicated in the IF-SYNC and APP-SYNC rules by the infinite down-counter on the left side of the reduction relation, and the premise $d \neq \infty$ in the IF and APP rules. Hence if the a branching or function application site is visited (revisited) inside a synchronized computation, as for example in case of nested (recursive) computations, the sync time is *not* reset at each such visit; the corresponding code is already under synchronization as part of the encapsulating computation.

The premise $\Phi(\mathsf{p}) = \varnothing$ in the IF rule (and correspondingly in the APP rule) indicates the guard flowing into the corresponding branching site is of low security, and so its computation is not to be synchronized. A type system, to be presented in Section 3.1, ensures the table of sync timer functions Φ correctly identifies such branching and application statements based on the security levels of the guards and the functions flowing into them.

We now formally state some properties of the operational semantics.

Lemma 3.4 (Properties of Operational Semantics)**.**

1. *(Canonicality of Derivation). If $\big(\mathrm{H}, p\big)_\infty \longrightarrow^n_\Phi \big(\mathrm{H}', e'\big)_{d'}$ then this derivation is canonical.*
2. *(Program to Program). If $\big(\mathrm{H}, p\big)_d \longrightarrow^n_\Phi \big(\mathrm{H}', p'\big)_{d'}$ then $d' = d - n$. (Hence, if $d = \infty$ then $d' = \infty$.)*
3. *(A Stuck State). For any H and p, there does not exist a Φ, H′, e', and d', such that $\big(\mathrm{H}, p\big)_0 \longrightarrow_\Phi \big(\mathrm{H}', e'\big)_{d'}$.*
4. *(Time-Fixed Computation). If $\big(\mathrm{H}, \langle p \rangle^v\big)_n \longrightarrow^{n'}_\Phi \big(\mathrm{H}', e'\big)_{d'}$ then either,*
   (a) For some p', $e' = \langle p' \rangle^v$ and $d' = n - n'$; or
   (b) For some v', $e' = v'$, $n' = n + 1$ and $d' = \infty$.
5. *(IF-SYNC). If $\Phi(\mathsf{p})(\overline{i_k}) = n$ and $\big(\mathrm{H}, \mathsf{if}^v_{\mathsf{p},\overline{i_k}}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2\big)_\infty \longrightarrow^{n'}_\Phi \big(\mathrm{H}', v'\big)_{d'}$ then $n' = n + 2$ and $d' = \infty$. (Hence, if $n = 0$ then $n' = 2$.)*
6. *(APP-SYNC). If $\Phi(\mathsf{p})(\overline{i_k}) = n$ and $\big(\mathrm{H}, (\lambda x.\, p)\,(v')^v_{\mathsf{p},\overline{i_k}}\big)_\infty \longrightarrow^{n'}_\Phi \big(\mathrm{H}', v''\big)_{d'}$ then $n' = n + 2$ and $d' = \infty$. (Hence, if $n = 0$ then $n' = 2$.)*

Proof. 1. *(Canonicality of Derivation)*. By induction on the derivation of $\big(\mathrm{H}, p\big)_\infty \longrightarrow^n_\Phi \big(\mathrm{H}', e'\big)_{d'}$.
2. *(Program to Program)*. By induction on the derivation of $\big(\mathrm{H}, p\big)_d \longrightarrow^n_\Phi \big(\mathrm{H}', p'\big)_{d'}$.
3. *(A Stuck State)*. By induction on the structure of p given the syntax-directedness of the semantics rules in Figure 8.
4. *(Time-Fixed Computation)*. By induction on the derivation of $\big(\mathrm{H}, \langle p \rangle^v\big)_n \longrightarrow^{n'}_\Phi \big(\mathrm{H}', e'\big)_{d'}$.
5. (IF-SYNC). By IF-SYNC and Lemma 3.4[4].
6. (APP-SYNC). By APP-SYNC and Lemma 3.4[4].

□

3.1 A Type System for λ^{sync}_{seq}

As discussed above, the operational semantics of λ^{sync}_{seq} relies on the table of sync timer functions Φ to indicate whether the computation of a branching or application statement is to be synchronized or not – if a sync timer function is defined then the corresponding statement is to be synchronized, otherwise not. In order to prevent timing leaks only the branching and application statements with running times dependent on high data need to be synchronized, that is, those with high guards or functions flowing into them. For simplicity we employ a basic monomorphic type system to identify such statements in this paper; more expressive polymorphic type systems [32] could be employed to better effect, as could dynamic techniques [39].

Let $(\mathcal{L}, \leq)$ be a lattice whose elements, denoted by ℓ and pc, represent *security levels*. Following Denning [8] we typically use the meta-variable pc, rather than ℓ, when considering information obtained by observing the value of the "program counter". We write $\bot$ for $\mathcal{L}$'s least element. The type grammar and subtyping

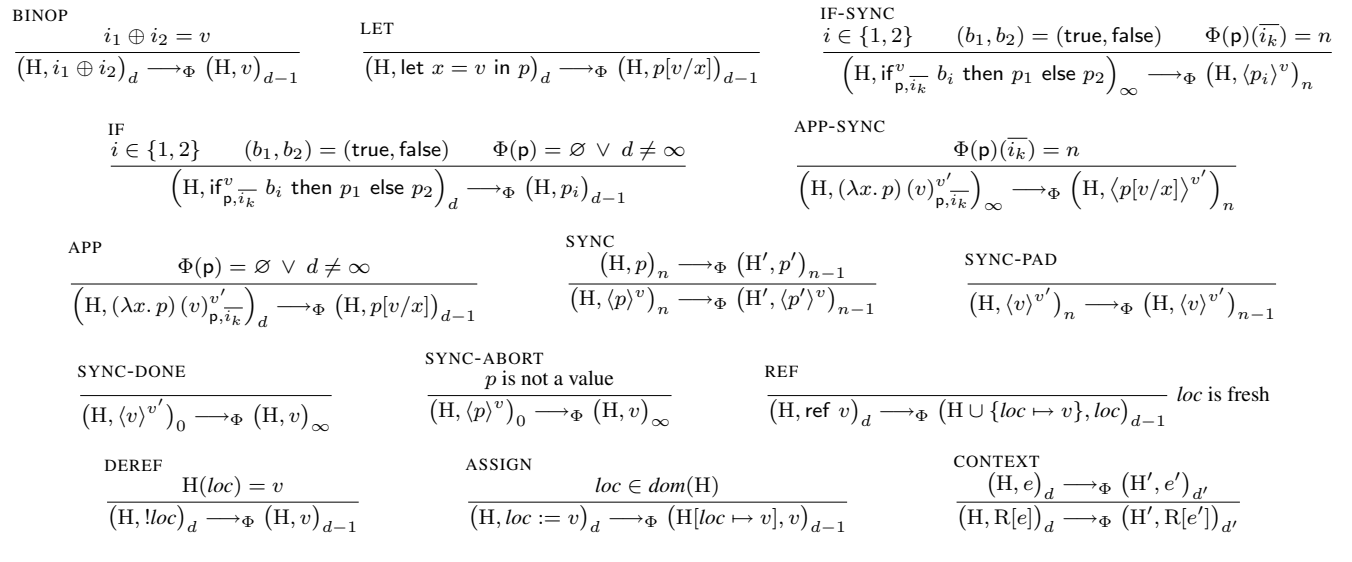

Figure 8. λ^{sync}_{seq}: Syntax-Directed Small-Step Operational Semantics

t	$::=$	$int \mid bool \mid \tau \to \tau \mid \tau\ ref$	*ground type*
τ	$::=$	t^ℓ	*type*
Γ	$:$	$\{\overline{x}\} \to \{\overline{\tau}\}$	*type of environment*
$\mathcal{H}$	$:$	$\{\overline{loc}\} \to \{\overline{\tau}\}$	*type of heap*
$\hat{t}$	$::=$	$int \mid bool$	*ground int/bool type*
$\hat{\tau}$	$::=$	$\hat{t}^\ell$	*int/bool type*

$$\frac{\ell \le \ell'}{\hat{t}^\ell \le \hat{t}^{\ell'}} \qquad \frac{\ell \le \ell' \qquad \ell' \lhd \tau}{\tau\ ref^\ell \le \tau\ ref^{\ell'}} \qquad \frac{\tau_1' \le \tau_1 \qquad \tau_2 \le \tau_2'}{(\tau_1 \to \tau_2)^\ell \le (\tau_1' \to \tau_2')^\ell}$$

Figure 9. λ^{sync}_{seq}: Type Grammar and Subtyping Rules

rules are defined in Figure 9. The annotation ℓ on the ground type t in type τ reflects the security level of the information the corresponding expression may carry. The function *seclevel* on types returns the associated security levels: $seclevel(t^\ell) = \ell$, for any t and ℓ. The binary predicate $\ell \lhd \tau$ (read: ℓ guards τ) holds iff $\ell \le seclevel(\tau)$. The type rules appear in Figure 10. In a type judgement $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$, pc denotes the security level of the context, while ℓ represents the least upper bound of all "low" security levels in $\mathcal{L}$, that is, for any $\ell' \in \mathcal{L}$, ℓ' is considered low iff $\ell' \le \ell$, otherwise ℓ' is high. The function *type* on binary operators, used in the type rule *binop*, is defined as, $type(\oplus_a) = int$ and $type(\oplus_r) = bool$. We use the following shorthand: '$pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \overline{e_k} : \tau$' abbreviates '$\forall 1 \le i \le k.\ pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e_i : \tau$', '$pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \overline{e_k : \tau_k}$' abbreviates '$\forall 1 \le i \le k.\ pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e_i : \tau_i$', and '$pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \overline{e_k, e_k' : \tau_k}$' abbreviates '$\forall 1 \le i \le k.\ pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e_i, e_i' : \tau_i$'.

The type system is fairly standard with much of the notation adapted from [32]. It mainly serves to identify the branching and application statements needing synchronization to secure timing channels. It does so by ensuring the existence of associated sync timer functions, with appropriate arities, for such statements via the type rules *if-sync* and *app-sync*; the nonexistence of sync timer functions for all other branching and application statements not needing synchronization is ensured by the *if* and *app* rules, respectively. As pointed out in the discussion on the operational semantics, only top-level branching and application statements, that is, only those that appear in low contexts, with high guards or functions flowing into them, need to be synchronized for securing timing channels; this is captured by the premise $pc \le \ell \wedge \ell' \not\le \ell \wedge \Phi(\mathsf{p}) = \phi^k$ in the *if-sync* and *app-sync* rules, while the premise $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \overline{v_k} : int^\ell$ ensures all parameters to the corresponding sync timer function ϕ^k are low integers. The other statements, not needing synchronization, are indicated by the premise $\left(pc \le \ell \wedge \ell' \le \ell \wedge \Phi(\mathsf{p}) = \varnothing\right) \vee pc \not\le \ell$ in the *if* and *app* rules; the subscripts $\overline{v_k}$ and the superscript v are in fact unused in the *if* and *app* rules.

We now formally state some properties of the type system and then prove the type safety of λ^{sync}_{seq}. We start by defining some notation. The subtyping relation between environments $\Gamma \le \Gamma'$ holds iff $dom(\Gamma) = dom(\Gamma')$ and $\forall x \in dom(\Gamma).\ \Gamma(x) \le \Gamma'(x)$. The type judgement $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (\mathrm{H}, e) : \tau$ holds iff $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}$ and $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$ hold.

Lemma 3.5 (Properties of the Type System)**.**

1. *(Program Counter).*
   (a) *(Typing Invariant). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$ *then* $pc \lhd \tau$.
   (b) *(Value). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau$ *and* $pc' \lhd \tau$ *then* $pc', \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau$.
   (c) *(Bump-Up). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau$, $pc \le pc'$ *and* $pc' \lhd \tau$ *then* $pc', \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau$.
   (d) *(Bump-Down).*
      i. *("Low" to further "Low"). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$ *and* $pc_{sub} \le pc \le \ell$ *then* $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$.
      ii. *("High" to lesser "High"). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau$, $pc_{sub} \le pc$ *and* $pc_{sub}, pc \not\le \ell$ *then* $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau$.
2. *(Subtyping).*
   (a) *(Type of Environment). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$ *and* $\Gamma_{sub} \le \Gamma$ *then* $pc, \Gamma_{sub}, \mathcal{H} \vdash_{\Phi\ell} e : \tau$.
   (b) *(Expression). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$ *and* $\tau \le \tau'$ *then* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau'$.
3. *(Heap Update). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}$ *and* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \mathcal{H}(loc)$ *then* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}[loc \mapsto v]$.
4. *(Extension of Type of Heap). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$ *and* $\mathcal{H} \subseteq \mathcal{H}'$ *then* $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} e : \tau$.

$$\textit{var}\ \frac{\Gamma(x) \leq \tau \qquad pc \lhd \tau}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} x : \tau} \qquad \textit{int}\ \frac{pc \leq \ell'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} i : int^{\ell'}} \qquad \textit{bool}\ \frac{pc \leq \ell'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} b : bool^{\ell'}}$$

$$\textit{func}\ \frac{pc \leq \ell' \qquad \ell', \Gamma[x \mapsto \tau], \mathcal{H} \vdash_{\Phi\ell} p : \tau'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \lambda x.\, p : (\tau \to \tau')^{\ell'}}$$

$$\textit{location}\ \frac{pc \leq \ell' \qquad \mathcal{H}(loc) = \tau \qquad \ell' \lhd \tau}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} loc : \tau\ ref^{\ell'}} \qquad \textit{binop}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p_1, p_2 : int^{\ell'}}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p_1 \oplus p_2 : type(\oplus)^{\ell'}}$$

$$\textit{let}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau \qquad pc, \Gamma[x \mapsto \tau], \mathcal{H} \vdash_{\Phi\ell} p' : \tau'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathsf{let}\ x = p\ \mathsf{in}\ p' : \tau'}$$

$$\textit{if-sync}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : bool^{\ell'} \qquad pc \leq \ell \ \wedge\ \ell' \not\leq \ell \ \wedge\ \Phi(\mathsf{p}) = \phi^k \qquad pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \overline{v_k} : int^{\ell} \qquad \ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} p_1, p_2, v : \tau}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathsf{if}^{v}_{\mathsf{p},\overline{v_k}}\ p\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 : \tau}$$

$$\textit{app-sync}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : (\tau \to \tau')^{\ell'} \qquad pc \leq \ell \ \wedge\ \ell' \not\leq \ell \ \wedge\ \Phi(\mathsf{p}) = \phi^k \qquad pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p' : \tau \qquad pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \overline{v_k} : int^{\ell} \qquad \ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p\,(p')^{v}_{\mathsf{p},\overline{v_k}} : \tau'}$$

$$\textit{if}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : bool^{\ell'} \qquad \big(pc \leq \ell \ \wedge\ \ell' \leq \ell \ \wedge\ \Phi(\mathsf{p}) = \varnothing\big) \ \vee\ pc \not\leq \ell \qquad \ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} p_1, p_2 : \tau}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathsf{if}^{v}_{\mathsf{p},\overline{v_k}}\ p\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 : \tau}$$

$$\textit{app}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : (\tau \to \tau')^{\ell'} \qquad \big(pc \leq \ell \ \wedge\ \ell' \leq \ell \ \wedge\ \Phi(\mathsf{p}) = \varnothing\big) \ \vee\ pc \not\leq \ell \qquad pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p' : \tau \qquad \ell' \lhd \tau'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p\,(p')^{v}_{\mathsf{p},\overline{v_k}} : \tau'}$$

$$\textit{sync}\ \frac{pc \leq \ell' \qquad pc \leq \ell \ \wedge\ \ell' \not\leq \ell \qquad \ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} p, v : \tau}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \langle p \rangle^{v} : \tau}$$

$$\textit{ref}\ \frac{pc \leq \ell' \qquad pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau \qquad \ell' \lhd \tau}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathsf{ref}\ p : \tau\ ref^{\ell'}} \qquad \textit{deref}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau\ ref^{\ell'} \qquad \ell' \lhd \tau \qquad \tau \leq \tau'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} !p : \tau'}$$

$$\textit{assign}\ \frac{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau\ ref^{\ell'} \qquad \ell' \lhd \tau \qquad pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p' : \tau_{sub} \qquad \tau_{sub} \leq \tau, \tau'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p := p' : \tau'}$$

$$\textit{sync-in-context}\ \frac{\mathrm{R}[s] = \big(\mathrm{R}[x]\big)[s/x] \qquad pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} s : \tau \qquad pc, \Gamma[x \mapsto \tau], \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[x] : \tau'}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[s] : \tau'}$$

$$\textit{heap}\ \frac{dom(\mathcal{H}) = dom(\mathrm{H}) \qquad \forall loc \in dom(\mathrm{H}).\, pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}(loc) : \mathcal{H}(loc)}{pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}}$$

Figure 10. λ^{sync}_{seq}: Syntax-Directed Monomorphic Type Rules

5. *(Update of Type of Environment). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (\mathrm{H}, e) : \tau$ *and* $x \notin$ *free*(H, e) *then for any* τ', $pc, \Gamma[x \mapsto \tau'], \mathcal{H} \vdash_{\Phi\ell} (\mathrm{H}, e) : \tau$.
6. *(Redex Typing). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[e_{sub}] : \tau$ *then there exists a* τ_{sub} *such that* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e_{sub} : \tau_{sub}$.
7. *(Redex Replacement). If* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[e] : \tau$, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau_{sub}$ *and* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e' : \tau_{sub}$ *then* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[e'] : \tau$.

Proof. 1. *(Program Counter)*. By induction on the respective type derivations.
2. *(Subtyping)*. By induction on the respective type derivations.
3. *(Heap Update)*. By the type rule *heap*.
4. *(Extension of Type of Heap)*. By induction on the derivation of $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e : \tau$.
5. *(Update of Type of Environment)*. By induction on the derivation of $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (\mathrm{H}, e) : \tau$.
6. *(Redex Typing)*. By induction on the derivation of $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[e_{sub}] : \tau$.
7. *(Redex Replacement)*. By induction on the derivation of $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[e] : \tau$.

□

Lemma 3.6 (Value Substitution in a Program)**.** *If* $pc, \Gamma[x \mapsto \tau_x], \mathcal{H} \vdash_{\Phi\ell} p : \tau$, $pc_{sub} \leq pc$ *and* $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau_x$ *then* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p[v/x] : \tau$.

Proof. By induction on the derivation of $pc, \Gamma[x \mapsto \tau_x], \mathcal{H} \vdash_{\Phi\ell} p : \tau$. Following are the type rules from Figure 10 applicable at the root of this derivation.

1. *var*. So for some y and τ_{sub}, $e = y$ and $\Gamma[x \mapsto \tau_x](y) = \tau_{sub}$, $\tau_{sub} \leq \tau$ and $pc \lhd \tau$. Now there are two possible cases depending on whether x is equal to y or not:
   (a) $x = y$. So $\tau_{sub} = \tau_x$ and $y[v/x] = v$. By Lemma 3.5[2b,1c] $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau$, that is, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e[v/x] : \tau$, the requisite result.
   (b) $x \neq y$. So $y \in dom(\Gamma)$ and $y[v/x] = y$. Then by the type rule *var* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} y : \tau$ holds, that is, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e[v/x] : \tau$ holds, the requisite result.
2. *int*. So for some i and ℓ', $e = i$ and $\tau = int^{\ell'}$ Now $i[v/x] = i$; then directly by *int*, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e[v/x] : \tau$ holds, the requisite result.
3. *bool*, *location*. Analogous to the case 2 above.
4. *func*. So for some y, p, τ_y, τ_p and ℓ', $e = \lambda y.\, p$, $\tau = (\tau_y \to \tau_p)^{\ell'}$ $pc \leq \ell'$ and $\ell', \Gamma[x \mapsto \tau_x][y \mapsto \tau_y], \mathcal{H} \vdash_{\Phi\ell} p : \tau_p$. There are two possible cases depending on whether x is equal to y or not:
   (a) $x = y$. So $(\lambda y.\, p)[v/x] = \lambda y.\, p$ and $\ell', \Gamma[y \mapsto \tau_y], \mathcal{H} \vdash_{\Phi\ell} p : \tau_p$. Then by *func*, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e[v/x] : \tau$ holds, the requisite result.

(b) $x \neq y$. So $(\lambda y.\, p)[v/x] = \lambda y.\, \big(p[v/x]\big)$, and $\ell', \Gamma[y \mapsto \tau_y][x \mapsto \tau_x], \mathcal{H} \vdash_{\Phi\ell} p : \tau_p$ holds. Now given $p[v/x]$ denotes capture-avoiding substitution, $y \notin \mathit{free}(v)$. Then by premise and Lemma 3.5[5] $pc_{sub}, \Gamma[y \mapsto \tau_y], \mathcal{H} \vdash_{\Phi\ell} v : \tau_x$ holds. Also $pc_{sub} \leq \ell'$. Then by induction hypothesis $\ell', \Gamma[y \mapsto \tau_y], \mathcal{H} \vdash_{\Phi\ell} p[v/x] : \tau_p$ holds. Finally by *func*, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e[v/x] : \tau$ holds, the requisite result.

5. *binop*, *app*, *ref*, *deref*, *assign*. Directly by induction.
6. *let*. Similar to case 4 above.
7. *if-sync*, *if*, *app-sync*. Directly by induction given Lemma 3.5[1a].

□

We define the type judgement $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (p, d) : \tau$ to hold iff $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau,\ pc \leq \ell \iff d = \infty$ and $pc \not\leq \ell \iff d \neq \infty$; note the correlation between the security level of the context pc and the down-counter d – a high context bi-implies synchronized computation of the program p, while a low context bi-implies non-synchronized computation. Also, the type judgement $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (\mathrm{R}[s], n) : \tau$ is defined to hold iff $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[s] : \tau$.

Main Lemma 3.7 (Type Preservation)**.** *If* $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}$, $pc_{sub} \leq pc,\ pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e, d) : \tau$ *and* $\big(\mathrm{H}, e\big)_d \longrightarrow_\Phi \big(\mathrm{H}', e'\big)_{d'}$ *then there exists a* $\mathcal{H}'$ *such that* $\mathcal{H} \subseteq \mathcal{H}',\ pc_{sub}, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} \mathrm{H}'$ *and* $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} (e', d') : \tau$.

Proof. By induction on the derivation of $\big(\mathrm{H}, e\big)_d \longrightarrow_\Phi \big(\mathrm{H}', e'\big)_{d'}$. Following are the semantics rules from Figure 8 applicable at the root of this derivation.

1. BINOP. So for some i_1 and i_2, $e = i_1 \oplus i_2$, $\mathrm{H}' = \mathrm{H}$, for $i_1 \oplus i_2 = v$, $e' = v$ and $d' = d - 1$.
   Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds. By premise and Lemma 3.5[1a], $pc \lhd \tau$. Then by premise with *binop* and *int/bool*, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$, the second requisite result, holds as well.
2. LET. So for some x, v and p, $e = (\mathsf{let}\ x = v\ \mathsf{in}\ p)$, $\mathrm{H}' = \mathrm{H}$, $e' = p[v/x]$ and $d' = d - 1$.
   Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds. By premise, *let* and Lemma 3.6 (Value Substitution in a Program) the second requisite result, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$, holds as well.
3. IF-SYNC. So for some p, $\overline{i_k}$, v, b, p_1, p_2 and n, $e = \mathsf{if}^v_{\mathsf{p},\overline{i_k}}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$, $d = \infty$, $\Phi(\mathsf{p})(\overline{i_k}) = n$, $\mathrm{H}' = \mathrm{H}$ and $d' = n$. Also, $d = \infty$ implies $pc \leq \ell$.
   Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds.
   Now to prove the second requisite result we need to consider two possible cases depending on the value of b:
   (a) $b = \mathsf{true}$. So $e' = \langle p_1 \rangle^v$. By premise, *if-sync* and *sync* $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$ holds, the second requisite result.
   (b) $b = \mathsf{false}$. Analogous to the subcase 3a above.
4. IF. So for some p, $\overline{i_k}$, v, b, p_1 and p_2, $e = \mathsf{if}^v_{\mathsf{p},\overline{i_k}}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$, $\mathrm{H}' = \mathrm{H}$ and $d' = d - 1$.
   Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds.
   Now to prove the second requisite result we need to consider two possible cases depending on the value of b:
   (a) $b = \mathsf{true}$. So $e' = p_1$. There again two possible cases depending on whether d is ∞ or not:
      i. $d = \infty$. So $pc \leq \ell$ and as per the premise of IF, $\Phi(\mathsf{p}) = \varnothing$. Then by premise, *if* and Lemma 3.5[1a,1(d)i] $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$ holds, the second requisite result.
      ii. $d \neq \infty$. So $pc \not\leq \ell$. Then by premise, *if* and Lemma 3.5[1a,1(d)ii] $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$ holds, the second requisite result.
   (b) $b = \mathsf{false}$. Analogous to the subcase 4a above.
5. APP-SYNC. So for some p, $\overline{i_k}$, v, x, p, v' and n, $e = (\lambda x.\, p)\,(v')^v_{\mathsf{p},\overline{i_k}}$, $d = \infty$, $\Phi(\mathsf{p})(\overline{i_k}) = n$, $\mathrm{H}' = \mathrm{H}$, $e' = \big\langle p[v'/x] \big\rangle^v$ and $d' = n$. Also, $d = \infty$ implies $pc \leq \ell$.
   Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds.
   By *app-sync* for some τ_x and ℓ', $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \lambda x.\, p : (\tau_x \to \tau)^{\ell'}$, $\ell' \not\leq \ell$, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v' : \tau_x$ and $\ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau$. Then by *func*, $pc \leq \ell'$ and $\ell', \Gamma[x \mapsto \tau_x], \mathcal{H} \vdash_{\Phi\ell} p : \tau$. By Lemma 3.6 (Value Substitution in a Program) $\ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} p[v'/x] : \tau$. Then by *sync*, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$, the second requisite result, holds.
6. APP. So for some p, $\overline{i_k}$, v, x, p and v', $e = (\lambda x.\, p)\,(v')^v_{\mathsf{p},\overline{i_k}}$, $\mathrm{H}' = \mathrm{H}$, $e' = p[v'/x]$ and $d' = d - 1$.
   Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds.
   Now to prove the second requisite result we need to consider two possible cases depending on the value of d:
   (a) $d = \infty$. So $pc \leq \ell$, and as per APP $\Phi(\mathsf{p}) = \varnothing$. Then by *app*, *func*, Lemma 3.6 (Value Substitution in a Program) and Lemma 3.5[1a,1(d)i] $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$ holds, the second requisite result.
   (b) $d \neq \infty$. So $pc \not\leq \ell$. Then by *app*, *func*, Lemma 3.6 (Value Substitution in a Program) and Lemma 3.5[1a,1(d)ii] $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$ holds, the second requisite result.
7. REF. So for some v and fresh *loc*, $e = \mathsf{ref}\ v$, $\mathrm{H}' = \mathrm{H} \cup \{loc \mapsto v\}$, $e' = loc$ and $d' = d - 1$. By *ref* for some τ_v and ℓ', $\tau = \tau_v\ \mathit{ref}^{\ell'}$, $pc \leq \ell'$, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau_v$ and $\ell' \lhd \tau_v$. By Lemma 3.5[1b] $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau_v$; then by premise, Lemma 3.5[4] and *heap*, for $\mathcal{H}' = \mathcal{H} \cup \{loc \mapsto \tau_v\}$, $pc_{sub}, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds, and by *location*, $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} (e', d') : \tau$, the second requisite result, holds.
8. DEREF. So for some *loc* and v, $e = !loc$, $\mathrm{H}(loc) = v$, $\mathrm{H}' = \mathrm{H}$, $e' = v$ and $d' = d - 1$.
   Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds.
   By *deref* and *location* for some τ_v and ℓ', $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} loc : \tau_v\ \mathit{ref}^{\ell'}$, $pc \leq \ell'$, $\mathcal{H}(loc) = \tau_v$, $\ell' \lhd \tau_v$ and $\tau_v \leq \tau$. By *heap* $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau_v$. Then by Lemma 3.5[1b,2b] $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} (e', d') : \tau$, the second requisite result, holds.
9. ASSIGN. So for some *loc* and v, $e = loc := v$, $loc \in \mathit{dom}(\mathrm{H})$, $\mathrm{H}' = \mathrm{H}[loc \mapsto v]$, $e' = v$ and $d' = d - 1$.
   By premise and *assign* and *location* for some τ_{loc}, ℓ' and τ_v, $\mathcal{H}(loc) = \tau_{loc}$, $pc \leq \ell'$, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} loc : \tau_{loc}\ \mathit{ref}^{\ell'}$, $\ell' \lhd \tau_{loc}$, $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau_v$ and $\tau_v \leq \tau_{loc}, \tau$. Then by premise and Lemma 3.5[1b,2b] $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} v : \tau_{loc}$ holds.
   Then by Lemma 3.5[3] $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds. And by Lemma 3.5[2b] $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$, the second requisite result, holds.
10. SYNC. So for some p, v, n and p', $e = \langle p \rangle^v$, $d = n$, $\big(\mathrm{H}, p\big)_n \longrightarrow_\Phi \big(\mathrm{H}', p'\big)_{n-1}$, $e' = \langle p' \rangle^v$ and $d' = n - 1$.
   By *sync* $pc \leq \ell, \ell'$, $\ell' \not\leq \ell$ and $\ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} p, v : \tau$. Then $pc_{sub} \leq \ell'$ and $\ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} (p, n) : \tau$. By induction hypothesis there exists a $\mathcal{H}'$ such that $\mathcal{H} \subseteq \mathcal{H}'$, $pc_{sub}, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, and $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} (p', n - 1) : \tau$. Then by *sync*, $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} (e', d') : \tau$, the second requisite result, holds.

11. SYNC-PAD. So for some v, v' and n, $e = \langle v \rangle^{v'}$, $d = n$, $\mathrm{H}' = \mathrm{H}$, $e' = e$ and $d' = n - 1$. Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$ and $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$, the requisite results, hold.
12. SYNC-DONE. So for some v and v', $e = \langle v \rangle^{v'}$, $d = 0$, $\mathrm{H}' = \mathrm{H}$, $e' = v$ and $d' = \infty$.
Directly by premise $pc_{sub}, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}'$, the first requisite result, holds.
By *sync* $pc \leq \ell, \ell'$, $\ell' \not\leq \ell$ and $\ell', \Gamma, \mathcal{H} \vdash_{\Phi\ell} v, v' : \tau$. By Lemma 3.5[1a] $\ell' \lhd \tau$, implying $pc \lhd \tau$. Then by Lemma 3.5[1b] $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (e', d') : \tau$, the second requisite result, holds.
13. SYNC-ABORT. Similar to case 12 above.
14. CONTEXT. So for some R and e_{sub}, $e = \mathrm{R}[e_{sub}]$, $\big(\mathrm{H}, e_{sub}\big)_d \longrightarrow_\Phi \big(\mathrm{H}', e'_{sub}\big)_{d'}$ and $e' = \mathrm{R}[e'_{sub}]$.
By Lemma 3.5[6] there exists a τ_{sub} such that there exists a τ_{sub} such that $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} e_{sub} : \tau_{sub}$. By induction hypothesis, there exists a $\mathcal{H}'$ such that $\mathcal{H} \subseteq \mathcal{H}'$, $pc_{sub}, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} \mathrm{H}'$ and $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} (e'_{sub}, d') : \tau_{sub}$. Then by Lemma 3.5[4,7] $pc, \Gamma, \mathcal{H}' \vdash_{\Phi\ell} (e', d') : \tau$ holds, the second requisite result.

□

Main Lemma 3.8 (Progress)**.** *If $pc_{sub}, \emptyset, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}$, $pc_{sub} \leq pc$ and $pc, \emptyset, \mathcal{H} \vdash_{\Phi\ell} (e, d) : \tau$ then either e is a value, or e is a program and $d = 0$, or there exists a H', e' and d' such that $\big(\mathrm{H}, e\big)_d \longrightarrow_\Phi \big(\mathrm{H}', e'\big)_{d'}$.*

Proof. By induction on the derivation of $pc, \emptyset, \mathcal{H} \vdash_{\Phi\ell} (e, d) : \tau$. □

Definition 3.9 (Convergence)**.** *A configuration C given a table of sync timer functions Φ converges to a value v in n steps iff $\mathsf{C} \longrightarrow_\Phi^n \big(\mathrm{H}, v\big)_{d'}$, for some H and d.*

Definition 3.10 (Divergence)**.** *A configuration C given a table of sync timer functions Φ diverges iff for some C', $\mathsf{C} \longrightarrow_\Phi \mathsf{C}'$ and C', given Φ, in turn diverges.*

Definition 3.11. *A program p given a table of sync timer functions Φ converges to value v in n steps (diverges) iff the configuration $\big(\emptyset, p\big)_\infty$ given Φ converges to the value v in n steps (diverges).*

Definition 3.12 (Well-Formed Program)**.** *A program p given a table of sync timer functions Φ is said to be well-formed iff $\bot, \emptyset, \emptyset \vdash_{\Phi\ell} p : \tau$ holds, for some ℓ and τ.*

Theorem 3.13 (Type Safety)**.** *A well-formed program either converges to a value or diverges.*

Proof. By Lemma 3.4[2] and Main Lemmas 3.7 (Type Preservation) and 3.8 (Progress). □

4. Security Properties of λ_{seq}^{sync}

We now formally establish the security of λ_{seq}^{sync}. We show λ_{seq}^{sync} is secure with respect to external timing channels by proving a strong noninterference result [9] between the high data and the running time of a program; the term *strong noninterference* is used to indicate the robustness of the result in presence of termination channels. We prove this result by showing how executions of two programs, which differ only in high values, are strongly bisimilar. Strong bisimilarity is an isomorphism of expressions at all intermediate steps of execution; the term *strong bisimilarity* indicates the lock-step nature of such executions: they must match up step-for-step. The bisimulation relation, to be defined, requires all low values to be identical, while allowing high values to differ. This is a much tighter alignment than is used in standard noninterference results, as the prevention of timing leaks requires a tighter correspondence.

Let $\mu : \{\overline{loc}\} \rightarrow \{\overline{loc}\}$ be an injective function from heap locations (of one run) to the heap locations (of the other run), denoting the correspondence between the supposed bisimilar locations of the respective heaps. Note the fresh heap locations generated by the REF rule at intermediate steps during bisimulation need not be identical; hence the need for the following binary relation, $\doteq_\mu$, on programs which establishes equality of low values in bisimilar programs, except for the heap locations which are required to be isomorphic as per μ. A program p_1 is μ-equal to a program p_2, that is, $p_1 \doteq_\mu p_2$, iff for $p_1 = loc$ we have $p_2 = \mu(loc)$, for some loc; for all other cases of p_1 the relation is homomorphic.

Definition 4.1 (μ-Equality Relation)**.**

1. *(Values).* $v_1 \doteq_\mu v_2$ *iff either,*
   (a) $v_1 = v_2$*; or*
   (b) $v_1 = \lambda x.\, p_1$, $v_2 = \lambda x.\, p_2$ *and* $p_1 \doteq_\mu p_2$*, for some x; or*
   (c) $v_1 = loc$ *and* $v_2 = \mu(loc)$*, for some loc.*
2. *(Programs).* $p_1 \doteq_\mu p_2$ *iff either,*
   (a) *(Binary Operation).* $p_1 = p_1' \oplus p_1''$, $p_2 = p_2' \oplus p_2''$, $p_1' \doteq_\mu p_2'$ *and* $p_1'' \doteq_\mu p_2''$*; or*
   (b) *(Let).* $p_1 = (\mathsf{let}\ x = p_1'\ \mathsf{in}\ p_1'')$, $p_2 = (\mathsf{let}\ x = p_2'\ \mathsf{in}\ p_2'')$, $p_1' \doteq_\mu p_2'$ *and* $p_1'' \doteq_\mu p_2''$*; or*
   (c) *(If).* $p_1 = \mathsf{if}^{v_1'}_{\mathsf{p}, \overline{v_k}}\ p_1'\ \mathsf{then}\ p_1''\ \mathsf{else}\ p_1'''$, $p_2 = \mathsf{if}^{v_2'}_{\mathsf{p}, \overline{v_k'}}\ p_2'\ \mathsf{then}\ p_2''\ \mathsf{else}\ p_2'''$, $p_1' \doteq_\mu p_2'$, $p_1'' \doteq_\mu p_2''$, $p_1''' \doteq_\mu p_2'''$, $v_1' \doteq_\mu v_2'$ *and* $\overline{v_k \doteq_\mu v_k'}$*; or*
   (d) *(App).* $p_1 = p_1'\,(p_1'')^{v_1'}_{\mathsf{p}, \overline{v_k}}$, $p_2 = p_2'\,(p_2'')^{v_2'}_{\mathsf{p}, \overline{v_k'}}$, $p_1' \doteq_\mu p_2'$, $p_1'' \doteq_\mu p_2''$, $v_1' \doteq_\mu v_2'$ *and* $\overline{v_k \doteq_\mu v_k'}$*; or*
   (e) *(Ref).* $p_1 = \mathsf{ref}\ p_1'$, $p_2 = \mathsf{ref}\ p_2'$ *and* $p_1' \doteq_\mu p_2'$*; or*
   (f) *(Deref).* $p_1 = !p_1'$, $p_2 = !p_2'$ *and* $p_1' \doteq_\mu p_2'$*; or*
   (g) *(Assign).* $p_1 = (p_1' := p_1'')$, $p_2 = (p_2' := p_2'')$, $p_1' \doteq_\mu p_2'$ *and* $p_1'' \doteq_\mu p_2''$.

The notation $\mu \circ \mu'$ (read as "μ composed with μ'") denotes function composition.

Lemma 4.2 (Properties of μ-Equality Relation)**.**

1. *(Symmetry). If* $p_1 \doteq_\mu p_2$ *then* $p_2 \doteq_{\mu^{-1}} p_1$.
2. *(Reflexivity). For all* p *and* μ, $p \doteq_\mu p$.
3. *(Transitivity). If* $p_1 \doteq_\mu p_2$ *and* $p_2 \doteq_{\mu'} p_3$ *then* $p_1 \doteq_{\mu' \circ \mu} p_3$.
4. *(Structure of Programs). If* $p_1 \doteq_\mu p_2$ *then* p_1 *and* p_2 *have the same outermost structure.*
5. *(Substitution). If* $p_1 \doteq_\mu p_2$ *and* $v_1 \doteq_\mu v_2$ *then* $p_1[v_1/x] \doteq_\mu p_2[v_2/x]$.
6. *(Extension of* μ*). If* $p_1 \doteq_\mu p_2$ *and* $\mu \subseteq \mu'$ *then* $p_1 \doteq_{\mu'} p_2$ *holds.*

Proof. 1. *(Symmetry).* By induction on the derivation of $p_1 \doteq_\mu p_2$, noting μ is an injection.
2. *(Reflexivity).* By induction on the structure of p, given Definition 4.1, in particular Condition 1a.
3. *(Transitivity).* By induction on the pair of the derivations of $p_1 \doteq_\mu p_2$ and $p_2 \doteq_{\mu'} p_3$, given Definition 4.1, in particular Condition 1c.
4. *(Structure of Programs).* By induction on the derivation of $p_1 \doteq_\mu p_2$.
5. *(Substitution).* By induction on the derivation of $p_1 \doteq_\mu p_2$.
6. *(Extension of* μ*).* By induction on the derivation of $p_1 \doteq_\mu p_2$.

□

We now define the bisimulation relation. We write the type judgement '$pc, \Gamma, \mathcal{H} \vdash^{low}_{\Phi\ell} p : \tau$' to denote $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau$ and $seclevel(\tau) \leq \ell$, and '$pc, \Gamma, \mathcal{H} \vdash^{high}_{\Phi\ell} p : \tau$' for $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau$ and $seclevel(\tau) \not\leq \ell$.

Definition 4.3 (λ^{sync}_{seq}: Bisimulation Relation)**.**

1. *(Programs).* $\big(\mathcal{H}_1, p_1\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, p_2\big)_\infty$ *iff there exists a* p'_1, p'_2, $\overline{x_k}$, $\overline{v_k}$, $\overline{v'_k}$ *and* $\overline{\tau_k}$ *such that,*
   (a) $p_1 = p'_1[\overline{v_k/x_k}]$ *and* $p_2 = p'_2[\overline{v'_k/x_k}]$*; and*
   (b) $p'_1 \doteq_\mu p'_2$*; and*
   (c) $\bot, \Gamma, \mathcal{H}_1 \vdash^{high}_{\Phi\ell} \overline{v_k : \tau_k}$ *and* $\bot, \Gamma, \mathcal{H}_2 \vdash^{high}_{\Phi\ell} \overline{v'_k : \tau_k}$*; and*
   (d) $\Gamma' = \Gamma \cup \{\overline{x_k \mapsto \tau_k}\}$*; and*
   (e) $\bot, \Gamma', \mathcal{H}_1 \vdash_{\Phi\ell} p'_1 : \tau$ *and* $\bot, \Gamma', \mathcal{H}_2 \vdash_{\Phi\ell} p'_2 : \tau$.
2. *(Expressions with Synchronization Constructs).* $\big(\mathcal{H}_1, \mathrm{R}_1[s_1]\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathrm{R}_2[s_2]\big)_n$ *iff there exists a* x *and* τ' *such that,*
   (a) $\mathrm{R}_1[s_1] = (\mathrm{R}_1[x])[s_1/x]$ *and* $\mathrm{R}_2[s_2] = (\mathrm{R}_2[x])[s_2/x]$*; and*
   (b) $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} s_1 : \tau'$ *and* $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} s_2 : \tau'$*; and*
   (c) $\big(\mathcal{H}_1, \mathrm{R}_1[x]\big)_\infty \sim^{\Gamma\cup\{x\mapsto\tau'\}\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathrm{R}_2[x]\big)_\infty$.
3. *(Heaps).* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ *iff*
   (a) $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} \mathrm{H}_1$ *and* $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} \mathrm{H}_2$*; and*
   (b) *For all* $loc \in dom(\mu)$*, there exists a* τ *such that,* $\mathcal{H}_1(loc) = \tau = \mathcal{H}_2\big(\mu(loc)\big)$ *and* $\big(\mathcal{H}_1, \mathrm{H}_1(loc)\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \Big(\mathcal{H}_2, \mathrm{H}_2\big(\mu(loc)\big)\Big)_\infty$.
4. *(Heaps, Expressions).* $\big(\mathcal{H}_1, \mathrm{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathrm{H}_2, e_2\big)_{d_2}$ *iff* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ *and* $\big(\mathcal{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$.

The values $\overline{v_k}$ and $\overline{v'_k}$ in case 1 above denote the high values inside programs p_1 and p_2, respectively, which may be different; hence they are elided in case 1a before establishing μ-equality of the residual programs in case 1b. Analogously, as the code in a synchronization construct is high, the synchronization code is removed in case 2a, before relating the enclosing programs for bisimilarity in case 2c. Furthermore, the down-counters of bisimilar configurations are required to be identical.

Lemma 4.4 (λ^{sync}_{seq}: Properties of Bisimulation Relation)**.**

1. *(Symmetry). If* $\big(\mathcal{H}_1, \mathrm{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathrm{H}_2, e_2\big)_{d_2}$ *then* $\big(\mathcal{H}_2, \mathrm{H}_2, e_2\big)_{d_2} \sim^{\Gamma\tau}_{\Phi\ell\mu-1} \big(\mathcal{H}_1, \mathrm{H}_1, e_1\big)_{d_1}$.
2. *(Reflexivity).*
   (a) *(Programs). If* $\bot, \Gamma, \mathcal{H} \vdash_{\Phi\ell} p : \tau$ *then for all* μ*,* $\big(\mathcal{H}, p\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}, p\big)_\infty$.
   (b) *(Expressions with Synchronization Constructs). If* $\bot, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{R}[s] : \tau$ *then for all* n *and* μ*,* $\big(\mathcal{H}, \mathrm{R}[s]\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}, \mathrm{R}[s]\big)_n$.
   (c) *(Heaps). If* $\bot, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}$ *and* $dom(\mathcal{H}) = \{\overline{loc_k}\}$ *then for all* μ*, such that* $\mu \subseteq \{\overline{loc_k \mapsto loc_k}\}$*,* $(\mathcal{H}, \mathrm{H}) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}, \mathrm{H})$ *holds.*
   (d) *(Heaps, Expressions). If* $\big(\mathcal{H}, e\big)_d \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}, e\big)_d$ *and* $(\mathcal{H}, \mathrm{H}) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}, \mathrm{H})$ *then* $\big(\mathcal{H}, \mathrm{H}, e\big)_d \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}, \mathrm{H}, e\big)_d$.
3. *(Transitivity). If* $\big(\mathcal{H}_1, \mathrm{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathrm{H}_2, e_2\big)_{d_2}$ *and* $\big(\mathcal{H}_2, \mathrm{H}_2, e_2\big)_{d_2} \sim^{\Gamma\tau}_{\Phi\ell\mu'} \big(\mathcal{H}_3, \mathrm{H}_3, e_3\big)_{d_3}$ *then* $\big(\mathcal{H}_1, \mathrm{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell(\mu'\circ\mu)} \big(\mathcal{H}_3, \mathrm{H}_3, e_3\big)_{d_3}$.
4. *(Isomorphism). If* $\big(\mathcal{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$ *then* e_1 *and* e_2 *are either both programs, or both have a sync construct.*
5. *(Programs). If* $\big(\mathcal{H}_1, p_1\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, p_2\big)_\infty$ *then* $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} p_1 : \tau$ *and* $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} p_2 : \tau$.
6. *(Expressions with Synchronization Constructs). If* $\big(\mathcal{H}_1, \mathrm{R}_1[s_1]\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathrm{R}_2[s_2]\big)_n$ *then* $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} \mathrm{R}_1[s_1] : \tau$ *and* $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} \mathrm{R}_2[s_2] : \tau$.
7. *(Synchronization Constructs). If* $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} s_1 : \tau$ *and* $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} s_2 : \tau$ *then for all* n *and* μ*,* $\big(\mathcal{H}_1, s_1\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, s_2\big)_n$.
8. *(Heaps). If* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ *then* $dom(\mathcal{H}_1) = dom(\mathrm{H}_1)$*,* $dom(\mathcal{H}_2) = dom(\mathrm{H}_2)$*,* $dom(\mu) \subseteq dom(\mathrm{H}_1)$ *and* $range(\mu) \subseteq dom(\mathrm{H}_2)$.
9. *(Extension of* μ*). If* $\big(\mathcal{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$ *and* $\mu \subseteq \mu'$ *then* $\big(\mathcal{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu'} \big(\mathcal{H}_2, e_2\big)_{d_2}$.
10. *(Extension of Heaps).*
    (a) *(Expressions). If* $\big(\mathcal{H}_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$*,* $\mathcal{H}_1 \subseteq \mathcal{H}'_1$ *and* $\mathcal{H}_2 \subseteq \mathcal{H}'_2$ *then* $\big(\mathcal{H}'_1, e_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}'_2, e_2\big)_{d_2}$.
    (b) *(Heaps). If* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$*,* $\mathcal{H}_1 \subseteq \mathcal{H}'_1$*,* $\mathrm{H}_1 \subseteq \mathrm{H}'_1$*,* $\mathcal{H}_2 \subseteq \mathcal{H}'_2$*,* $\mathrm{H}_2 \subseteq \mathrm{H}'_2$*,* $\bot, \Gamma, \mathcal{H}'_1 \vdash_{\Phi\ell} \mathrm{H}'_1$ *and* $\bot, \Gamma, \mathcal{H}'_2 \vdash_{\Phi\ell} \mathrm{H}'_2$ *then* $(\mathcal{H}'_1, \mathrm{H}'_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}'_2, \mathrm{H}'_2)$.
11. *(Values).*
    (a) *(Structural Congruence). If* $\big(\mathcal{H}_1, v_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and* e_2 *is a value.*
    (b) *("High" Values). If* $\bot, \Gamma, \mathcal{H}_1 \vdash^{high}_{\Phi\ell} v_1 : \tau$ *and* $\bot, \Gamma, \mathcal{H}_2 \vdash^{high}_{\Phi\ell} v_2 : \tau$ *then for all* μ*,* $\big(\mathcal{H}_1, v_1\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, v_2\big)_\infty$.
    (c) *("Low" Int/Bool Type). If* $\big(\mathcal{H}_1, \hat{v}\big)_\infty \sim^{\Gamma\hat{\tau}}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_\infty$ *and* $seclevel(\hat{\tau}) \leq \ell$ *then* $e_2 = \hat{v}$.
12. *(Heap Update with "High" Value). If* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ *and* $\bot, \Gamma, \mathcal{H}_1 \vdash^{high}_{\Phi\ell} v : \mathcal{H}_1(loc)$ *then* $(\mathcal{H}_1, \mathrm{H}_1[loc \mapsto v]) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$.
13. *(Binary Operation).*
    (a) *(Structural Congruence). If* $\big(\mathcal{H}_1, i_1 \oplus i'_1\big)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and for some* i_2 *and* i'_2*,* $e_2 = i_2 \oplus i'_2$.
    (b) (BINOP)*. If* $\big(\mathcal{H}_1, i_1 \oplus i'_1\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, i_2 \oplus i'_2\big)_\infty$*,* $i_1 \oplus i'_1 = v_1$ *and* $i_2 \oplus i'_2 = v_2$ *then* $\big(\mathcal{H}_1, v_1\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, v_2\big)_\infty$.
14. *(Let).*
    (a) *(Structural Congruence). If* $\big(\mathcal{H}_1, \mathsf{let}\ x = v_1\ \mathsf{in}\ p_1\big)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and for some* v_2 *and* p_2*,* $e_2 = (\mathsf{let}\ x = v_2\ \mathsf{in}\ p_2)$.
    (b) (LET)*. If* $\big(\mathcal{H}_1, \mathsf{let}\ x = v_1\ \mathsf{in}\ p_1\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathsf{let}\ x = v_2\ \mathsf{in}\ p_2\big)_\infty$ *then* $\big(\mathcal{H}_1, p_1[v_1/x]\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, p_2[v_2/x]\big)_\infty$.
15. *(If).*
    (a) *(Structural Congruence). If* $\big(\mathcal{H}_1, \mathsf{if}^{v_1}_{\mathsf{p},\overline{i_k}}\ b_1\ \mathsf{then}\ p_{1_T}\ \mathsf{else}\ p_{1_F}\big)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, e_2\big)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and for some* v_2*,* b_2*,* p_{2_T} *and* p_{2_F}*,* $e_2 = \mathsf{if}^{v_2}_{\mathsf{p},\overline{i_k}}\ b_2\ \mathsf{then}\ p_{2_T}\ \mathsf{else}\ p_{2_F}$.
    (b) (IF-SYNC)*. If* $\big(\mathcal{H}_1, \mathsf{if}^{v_1}_{\mathsf{p},\overline{i_k}}\ b_1\ \mathsf{then}\ p_{1_T}\ \mathsf{else}\ p_{1_F}\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathsf{if}^{v_2}_{\mathsf{p},\overline{i_k}}\ b_2\ \mathsf{then}\ p_{2_T}\ \mathsf{else}\ p_{2_F}\big)_\infty$ *and* $\Phi(\mathsf{p})(\overline{i_k}) = n$ *then* $\big(\mathcal{H}_1, \langle p_{1_T}\rangle^{v_1}\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \langle p_{2_T}\rangle^{v_2}\big)_n$, $\big(\mathcal{H}_1, \langle p_{1_T}\rangle^{v_1}\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \langle p_{2_F}\rangle^{v_2}\big)_n$, $\big(\mathcal{H}_1, \langle p_{1_F}\rangle^{v_1}\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \langle p_{2_T}\rangle^{v_2}\big)_n$, $\big(\mathcal{H}_1, \langle p_{1_F}\rangle^{v_1}\big)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, \langle p_{2_F}\rangle^{v_2}\big)_n$, $\big(\mathcal{H}_1, p_{1_T}\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, p_{2_T}\big)_\infty$ *and* $\big(\mathcal{H}_1, p_{1_F}\big)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \big(\mathcal{H}_2, p_{2_F}\big)_\infty$.

(c) (IF). If $\left(\mathcal{H}_1, \mathsf{if}^{v_1}_{\mathsf{p},\overline{i_k}}\ b_1\ \mathsf{then}\ p_{1_T}\ \mathsf{else}\ p_{1_F}\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathsf{if}^{v_2}_{\mathsf{p},\overline{i_k}}\ b_2\ \mathsf{then}\ p_{2_T}\ \mathsf{else}\ p_{2_F}\right)_\infty$ *and* $\Phi(\mathsf{p}) = \varnothing$ *then* $b_1 = b_2$, $\left(\mathcal{H}_1, p_{1_T}\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, p_{2_T}\right)_\infty$ *and* $\left(\mathcal{H}_1, p_{1_F}\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, p_{2_F}\right)_\infty$.

16. *(App).*
    - *(a) (Structural Congruence). If* $\left(\mathcal{H}_1, (\lambda x_1.\, p_1)\,(v_{1_{arg}})^{v_1}_{\mathsf{p},\overline{i_k}}\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and for some* x_2, p_2, $v_{2_{arg}}$ *and* v_2, $e_2 = (\lambda x_2.\, p_2)\,(v_{2_{arg}})^{v_2}_{\mathsf{p},\overline{i_k}}$.
    - *(b) (APP-SYNC). If* $\left(\mathcal{H}_1, (\lambda x_1.\, p_1)\,(v_{1_{arg}})^{v_1}_{\mathsf{p},\overline{i_k}}\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, (\lambda x_2.\, p_2)\,(v_{2_{arg}})^{v_2}_{\mathsf{p},\overline{i_k}}\right)_\infty$ *and* $\Phi(\mathsf{p})(\overline{i_k}) = n$ *then* $\left(\mathcal{H}_1, \langle p_1[v_{1_{arg}}/x_1]\rangle^{v_1}\right)_n \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \langle p_2[v_{2_{arg}}/x_2]\rangle^{v_2}\right)_n$.
    - *(c) (APP). If* $\left(\mathcal{H}_1, (\lambda x_1.\, p_1)\,(v_{1_{arg}})^{v_1}_{\mathsf{p},\overline{i_k}}\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, (\lambda x_2.\, p_2)\,(v_{2_{arg}})^{v_2}_{\mathsf{p},\overline{i_k}}\right)_\infty$ *and* $\Phi(\mathsf{p}) = \varnothing$ *then* $x_1 = x_2$ *and* $\left(\mathcal{H}_1, p_1[v_{1_{arg}}/x_1]\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, p_2[v_{2_{arg}}/x_2]\right)_\infty$.
17. *(Ref).*
    - *(a) (Structural Congruence). If* $\left(\mathcal{H}_1, \mathsf{ref}\ v_1\right)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and for some* v_2, $e_2 = \mathsf{ref}\ v_2$.
    - *(b) (REF). If* $\left(\mathcal{H}_1, \mathrm{H}_1, \mathsf{ref}\ v_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, \mathsf{ref}\ v_2\right)_\infty$ *then, for some* τ' *and* ℓ', $\tau = \tau'\ ref^{\ell'}$ *and for all* loc_1 *and* loc_2 *such that,* $loc_1 \notin dom(\mathrm{H}_1)$ *and* $loc_2 \notin dom(\mathrm{H}_2)$, $\left(\mathcal{H}_1', \mathrm{H}_1', loc_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', loc_2\right)_\infty$ *holds, such that* $\mathrm{H}_1' = \mathrm{H}_1 \cup \{loc_1 \mapsto v_1\}$, $\mathrm{H}_2' = \mathrm{H}_2 \cup \{loc_2 \mapsto v_2\}$, $\mathcal{H}_1' = \mathcal{H}_1 \cup \{loc_1 \mapsto \tau'\}$, $\mathcal{H}_2' = \mathcal{H}_2 \cup \{loc_2 \mapsto \tau'\}$, *and* $\mu' = \mu \cup \{loc_1 \mapsto loc_2\}$.
18. *(Deref).*
    - *(a) (Structural Congruence). If* $\left(\mathcal{H}_1, \mathrm{H}_1, !loc_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and for some* $loc_2 \in dom(\mathrm{H}_2)$, $e_2 = !loc_2$.
    - *(b) (DEREF). If* $\left(\mathcal{H}_1, \mathrm{H}_1, !loc_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, !loc_2\right)_\infty$ *then* $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{H}_1(loc_1)\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{H}_2(loc_2)\right)_\infty$.
19. *(Assign).*
    - *(a) (Structural Congruence). If* $\left(\mathcal{H}_1, \mathrm{H}_1, loc_1 := v_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$ *then* $d_1 = d_2 = \infty$ *and for some* $loc_2 \in dom(\mathrm{H}_2)$ *and* v_2, $e_2 = (loc_2 := v_2)$.
    - *(b) (ASSIGN). If* $\left(\mathcal{H}_1, \mathrm{H}_1, loc_1 := v_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, loc_2 := v_2\right)_\infty$ *then* $\left(\mathcal{H}_1, \mathrm{H}_1[loc_1 \mapsto v_1], v_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2[loc_2 \mapsto v_2], v_2\right)_\infty$.
20. *(Synchronization Construct).*
    - *(a) (Structural Congruence). If* $\left(\mathcal{H}_1, \langle p_1\rangle^{v_1}\right)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ *then for some* n, p_2 *and* v_2, $d_1 = d_2 = n$, $e_2 = \langle p_2\rangle^{v_2}$ *and* $\left(\mathcal{H}_1, v_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, v_2\right)_\infty$.
    - *(b) (Sub-Structural Bisimilarities). If* $\left(\mathcal{H}_1, \langle v_1\rangle^{v_1'}\right)_{n_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \langle v_2\rangle^{v_2'}\right)_{n_2}$ *then* $\left(\mathcal{H}_1, v_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, v_2\right)_\infty$, $\left(\mathcal{H}_1, v_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, v_2'\right)_\infty$, $\left(\mathcal{H}_1, v_1'\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, v_2\right)_\infty$ *and* $\left(\mathcal{H}_1, v_1'\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, v_2'\right)_\infty$.
21. *(Context).*
    - *(a) (Structural Congruence). If* $\left(\mathcal{H}_1, \mathrm{R}_1[e_{1_{sub}}]\right)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ *then for some* R_2, $e_{2_{sub}}$ *and* τ_{sub}, $e_2 = \mathrm{R}_2[e_{2_{sub}}]$ *and* $\left(\mathcal{H}_1, e_{1_{sub}}\right)_{d_1} \sim^{\Gamma\tau_{sub}}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_{2_{sub}}\right)_{d_2}$.
    - *(b) (CONTEXT). If* $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{R}_1[e_{1_{sub}}]\right)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{R}_2[e_{2_{sub}}]\right)_{d_2}$, $\left(\mathcal{H}_1, \mathrm{H}_1, e_{1_{sub}}\right)_{d_1} \sim^{\Gamma\tau_{sub}}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_{2_{sub}}\right)_{d_2}$, $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\mu \subseteq \mu'$ *and* $\left(\mathcal{H}_1', \mathrm{H}_1', e_{1_{sub}}'\right)_{d_1'} \sim^{\Gamma\tau_{sub}}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', e_{2_{sub}}'\right)_{d_2'}$ *then* $\left(\mathcal{H}_1', \mathrm{H}_1', \mathrm{R}_1[e_{1_{sub}}']\right)_{d_1'} \sim^{\Gamma\tau}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', \mathrm{R}_2[e_{2_{sub}}']\right)_{d_2'}$.

Proof. 1. *(Symmetry)*. By Definition 4.3 given Lemma 4.2[1].

2. *(Reflexivity)*.
   - (a) *(Programs)*. By Definition 4.3[1] and Lemma 4.2[2].
   - (b) *(Expressions with Synchronization Constructs)*. By the type rule *sync-in-context*, Lemma 4.4[2a] and Definition 4.3[2].
   - (c) *(Heaps)*. By Definition 4.3[3], the type rule *heap* and Lemma 4.4[2a].
   - (d) *(Heaps, Expressions)*. By Definition 4.3[4].
3. *(Transitivity)*. By Definition 4.3 and Lemma 4.2[3].
4. *(Isomorphism)*. By Definition 4.3[1,2].
5. *(Programs)*. By Definition 4.3[1c,1e] and Lemma 3.6 (Value Substitution in a Program).
6. *(Expressions with Synchronization Constructs)*. By Definition 4.3[2], Lemma 4.4[5], and the type rule *sync-in-context*.
7. *(Synchronization Constructs)*. By Definition 4.3[2].
8. *(Heaps)*. By Definition 4.3[3] and the type rule *heap*.
9. *(Extension of μ)*. By Definition 4.3[1,2] and Lemma 4.2[6].
10. *(Extension of Heaps)*.
    - (a) *(Expressions)*. By Definition 4.3[1,2] and Lemma 3.5[4].
    - (b) *(Heaps)*. By Definition 4.3[3] and Lemma 4.4[10a].
11. *(Values)*.
    - (a) *(Structural Congruence)*. By Definition 4.3[1], in particular Condition 1b, and Lemma 4.2[4].
    - (b) *("High" Values)*. By Definition 4.3[1], in particular Condition 1c.
    - (c) *("Low" Int/Bool Type)*. By Definition 4.3[1] and Definition 4.1[1a].
12. *(Heap Update with "High" Value)*. The premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, given Definition 4.3[3a], implies $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} \mathrm{H}_1$ and $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} \mathrm{H}_2$. We know $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} v : \mathcal{H}_1(loc)$; then by Lemma 3.5[3] $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} \mathrm{H}_1[loc \mapsto v]$ holds. Now there are two possible cases:
    - (a) $loc \notin dom(\mu)$. By Definition 4.3[3], given the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ and knowing $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} \mathrm{H}_1[loc \mapsto v]$, we get $(\mathcal{H}_1, \mathrm{H}_1[loc \mapsto v]) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, the requisite result.
    - (b) $loc \in dom(\mu)$. By premise we know $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ and $seclevel(\mathcal{H}_1(loc)) \not\leq \ell$. Let $\mathrm{H}_1(loc) = v_1$ and $\mathrm{H}_2\big(\mu(loc)\big) = v_2$. Then as per Definition 4.3[3b], for $\mathcal{H}_1(loc) = \tau$, $\left(\mathcal{H}_1, v_1\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, v_2\right)_\infty$ holds; then as per Lemma 4.4[5] $\bot, \Gamma, \mathcal{H}_2 \vdash^{high}_{\Phi\ell} v_2 : \tau$. Now knowing $\bot, \Gamma, \mathcal{H}_1 \vdash^{high}_{\Phi\ell} v : \tau$, by Lemma 4.4[11b] $\left(\mathcal{H}_1, v\right)_\infty \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, v_2\right)_\infty$ holds. Then by Definition 4.3[3] we get $(\mathcal{H}_1, \mathrm{H}_1[loc \mapsto v]) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, the requisite result.
13. *(Binary Operation)*.
    - (a) *(Structural Congruence)*. By Definition 4.3[1], in particular Condition 1b, and Lemma 4.2[4].
    - (b) *(BINOP)*. By Definition 4.3[1].
14. *(Let)*.
    - (a) *(Structural Congruence)*. By Definition 4.3[1], in particular Condition 1b, and Lemma 4.2[4].
    - (b) *(LET)*. By Definition 4.3[1], in particular Condition 1b, Definition 4.1[2b], Lemma 3.6 (Value Substitution in a Program) and Lemma 4.2[5].

15. *(If)*.
    (a) *(Structural Congruence)*. By Definition 4.3[1], in particular Condition 1b, and Lemma 4.2[4].
    (b) *(*IF-SYNC*)*. By Definition 4.3[1,2], the type rules *if-sync* and *sync* and Definition 4.1[2c].
    (c) *(*IF*)*. By Definition 4.3[1], the type rule *if*, and Definition 4.1[2c,1a].
16. *(App)*.
    (a) *(Structural Congruence)*. By Definition 4.3[1], in particular Condition 1b, and Lemma 4.2[4].
    (b) *(*APP-SYNC*)*. By Definition 4.3[1,2], the type rules *app-sync*, *func* and *sync*, Definition 4.1[2d] and Lemma 3.6 (Value Substitution in a Program).
    (c) *(*APP*)*. By Definition 4.3[1], the type rule *app*, Definition 4.1[2d,1b], Lemma 4.2[5] and Lemma 3.6 (Value Substitution in a Program).
17. *(Ref)*.
    (a) *(Structural Congruence)*. By Definition 4.3[1], in particular Condition 1b, and Lemma 4.2[4].
    (b) *(*REF*)*. By Definition 4.3[1,3] and the type rules *ref* and *heap*.
18. *(Deref)*.
    (a) *(Structural Congruence)*. By Definition 4.3[4,1,3], Lemma 4.2[4] and the premise $dom(\mathcal{H}) = dom(\mathrm{H})$ of the type rule *heap*.
    (b) *(*DEREF*)*. By Definition 4.3[1,3] and the type rules *deref* and *heap*.
19. *(Assign)*.
    (a) *(Structural Congruence)*. By Definition 4.3[4,1,3], Lemma 4.2[4] and the premise $dom(\mathcal{H}) = dom(\mathrm{H})$ of the type rule *heap*.
    (b) *(*ASSIGN*)*. By Definition 4.3[1,3], and the type rules *assign* and *heap*.
20. *(Synchronization Construct)*.
    (a) *(Structural Congruence)*. By Definition 4.3[2], in particular Condition 2b, the type rule *sync*, Lemma 3.5[1a] and Lemma 4.4[11b].
    (b) *(Sub-Structural Bisimilarities)*. By Definition 4.3[2], in particular Condition 2b, the type rule *sync* and Lemma 4.4[11b].
21. *(Context)*.
    (a) *(Structural Congruence)*. By induction on the structure of R_1 given Lemma 4.2[4] and Definition 4.3[1,2].
    (b) *(*CONTEXT*)*. By induction on the structures of R_1 and R_2, given Lemma 4.4[9,10a] and Definition 4.3[1,2].

□

The following lemma states that heap bisimilarity is preserved in computations under high contexts; ℓ' in the following lemma statement denotes such a high context.

Lemma 4.5 (Preservation of Heap Bisimilarity under a High Context, 1-step)**.** *If* $\left(\mathrm{H}_1, p_1\right)_{n_1} \longrightarrow_\Phi \left(\mathrm{H}_1', p_1'\right)_{n_1'}$, $\ell' \not\leq \ell$, $\ell', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} p_1 : \tau_1$ *and* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ *then there exists a* $\mathcal{H}_1'$ *such that* $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_1' : \tau_1$ *and* $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$.

Proof. By induction on the derivation of $\left(\mathrm{H}_1, p_1\right)_{n_1} \longrightarrow_\Phi \left(\mathrm{H}_1', p_1'\right)_{n_1'}$. Following are the possible syntax-directed semantics rules from Figure 8 that are applicable at the root of this derivation.

1. BINOP, LET, IF, APP and DEREF. These do not modify the heap, that is, $\mathrm{H}_1' = \mathrm{H}_1$ for in the case of all these rules.
   By premise and Definition 4.3[3a] $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} \mathrm{H}_1$ and $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} \mathrm{H}_2$ hold; also $\ell', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} (p_1, n_1) : \tau_1$ holds. Then by Main Lemma 3.7 (Type Preservation) there exists a $\mathcal{H}_1'$ such that $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\bot, \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} \mathrm{H}_1'$ and $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} (p_1', n_1') : \tau_1$, that is, $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_1' : \tau_1$, the first requisite result.
   We know $\mathrm{H}_1 = \mathrm{H}_1'$. Then by premise and Lemma 4.4[10b] we get $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, the second requisite result.
2. REF. So for some v_1, $p_1 =$ ref v_1 and for some fresh loc_1, $\left(\mathrm{H}_1, p_1\right)_{n_1} \longrightarrow_\Phi \left(\mathrm{H}_1', p_1'\right)_{n_1'}$ such that $\mathrm{H}_1' = \mathrm{H}_1 \cup \{loc_1 \mapsto v_1\}$, $p_1' = loc_1$ and $n_1' = n_1 - 1$.
   As in case 1 the first requisite result, that is, there exists a $\mathcal{H}_1'$ such that $\mathcal{H}_1 \subseteq \mathcal{H}_1'$ and $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_1' : \tau_1$, and $\bot, \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} \mathrm{H}_1'$ hold.
   We know $\mathrm{H}_1 \subseteq \mathrm{H}_1'$. Then by premise and Lemma 4.4[10b] we get $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, the second requisite result.
3. ASSIGN. So for some loc_1 and v_1, $p_1 = (loc_1 := v_1)$, $loc_1 \in dom(\mathrm{H}_1)$ and $\left(\mathrm{H}_1, p_1\right)_{n_1} \longrightarrow_\Phi \left(\mathrm{H}_1', p_1'\right)_{n_1'}$ such that $\mathrm{H}_1' = \mathrm{H}_1[loc_1 \mapsto v_1]$, $p_1' = v_1$ and $n_1' = n_1 - 1$.
   As in case 1 the first requisite result, that is, there exists a $\mathcal{H}_1'$ such that $\mathcal{H}_1 \subseteq \mathcal{H}_1'$ and $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_1' : \tau_1$, and $\bot, \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} \mathrm{H}_1'$ hold.
   By assumption, premise, Lemma 3.5[1a] and the type rules *assign* and *location* for some $\tau_{1_{sub}}$, $\ell' \lhd \mathcal{H}_1(loc_1)$, $\ell', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} v_1 : \tau_{1_{sub}}$ and $\tau_{1_{sub}} \leq \mathcal{H}_1(loc_1), \tau_1$. Then by Lemma 3.5[2b,1a,1b] $\bot, \Gamma, \mathcal{H}_1 \vdash^{high}_{\Phi\ell} v_1 : \mathcal{H}_1(loc_1)$ holds. By Lemma 4.4[12] we get $(\mathcal{H}_1, \mathrm{H}_1[loc_1 \mapsto v_1]) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, that is, $(\mathcal{H}_1, \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$.
   We know $\mathcal{H}_1 \subseteq \mathcal{H}_1'$ and $\bot, \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} \mathrm{H}_1'$. Then by Lemma 4.4[10b] we get $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, the second requisite result.
4. CONTEXT. So for some R_1, $p_{1_{sub}}$ and $p_{1_{sub}}'$, $p_1 = \mathrm{R}_1[p_{1_{sub}}]$, and for some $p_{1_{sub}}'$, $\left(\mathrm{H}_1, p_{1_{sub}}\right)_{n_1} \longrightarrow_\Phi \left(\mathrm{H}_1', p_{1_{sub}}'\right)_{n_1'}$ and $\left(\mathrm{H}_1, p_1\right)_{n_1} \longrightarrow_\Phi \left(\mathrm{H}_1', p_1'\right)_{n_1'}$ such that $p_1' = \mathrm{R}_1[p_{1_{sub}}']$. Further by premise we know $\ell', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} p_1 : \tau_1$; then by Lemma 3.5[6] there exists a $\tau_{1_{sub}}$ such that $\ell', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} p_{1_{sub}} : \tau_{1_{sub}}$.
   By induction hypothesis there exists a $\mathcal{H}_1'$ such that $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_{1_{sub}}' : \tau_{1_{sub}}$ and $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, the second requisite result.
   Now we know $\ell', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} \mathrm{R}_1[p_{1_{sub}}] : \tau_1$, $\ell', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} p_{1_{sub}} : \tau_{1_{sub}}$, $\mathcal{H}_1 \subseteq \mathcal{H}_1'$ and $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_{1_{sub}}' : \tau_{1_{sub}}$; then by Lemma 3.5[4,7] $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} \mathrm{R}_1[p_{1_{sub}}'] : \tau_1$, that is, $\ell', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_1' : \tau_1$ holds, the first requisite result.

□

Lemma 4.6 (Preservation of Heap Bisimilarity under High Context, n-steps)**.** *If* $\left(\mathrm{H}_1, p_1\right)_{n_1} \longrightarrow^n_\Phi \left(\mathrm{H}_1', p_1'\right)_{n_1'}$, $\left(\mathrm{H}_2, p_2\right)_{n_2} \longrightarrow^{n'}_\Phi \left(\mathrm{H}_2', p_2'\right)_{n_2'}$, $\ell_1', \ell_2' \not\leq \ell$, $\ell_1', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} p_1 : \tau_1$, $\ell_2', \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} p_2 : \tau_2$ *and* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, *then there exists a* $\mathcal{H}_1'$ *and* $\mathcal{H}_2'$ *such that* $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\ell_1', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_1' : \tau_1$, $\ell_2', \Gamma, \mathcal{H}_2' \vdash_{\Phi\ell} p_2' : \tau_2$ *and* $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2', \mathrm{H}_2')$.

Proof. By premise we know $\left(\mathrm{H}_1, p_1\right)_{n_1} \longrightarrow^n_\Phi \left(\mathrm{H}_1', p_1'\right)_{n_1'}$, $\ell_1' \not\leq \ell$, $\ell_1', \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} p_1 : \tau_1$ and $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$; by induction on the length of the above computation given Lemma 4.5 there exists a $\mathcal{H}_1'$ such that $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\ell_1', \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} p_1' : \tau_1$, the first requisite result, and $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$. By Lemma 4.4[1] for $\mu' = \mu^{-1}$, $(\mathcal{H}_2, \mathrm{H}_2) \sim^{\Gamma}_{\Phi\ell\mu'} (\mathcal{H}_1', \mathrm{H}_1')$ holds.

Further by premise we know $\left(\mathrm{H}_2, p_2\right)_{n_2} \longrightarrow_{\Phi}^{n'} \left(\mathrm{H}_2', p_2'\right)_{n_2'}$, $\ell_2' \not\leq \ell$ and $\ell_2', \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} p_2 : \tau_2$, and from above we know $(\mathcal{H}_2, \mathrm{H}_2) \sim^{\Gamma}_{\Phi\ell\mu'} (\mathcal{H}_1', \mathrm{H}_1')$; then by induction on the length of the above computation given Lemma 4.5 there exists a $\mathcal{H}_2'$ such that $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\ell_2', \Gamma, \mathcal{H}_2' \vdash_{\Phi\ell} p_2' : \tau_2$, the second requisite result, and $(\mathcal{H}_2', \mathrm{H}_2') \sim^{\Gamma}_{\Phi\ell\mu'} (\mathcal{H}_1', \mathrm{H}_1')$ hold. By Lemma 4.4[1] $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2', \mathrm{H}_2')$ holds, the third requisite result. □

As discussed above, only high subcomputations under high contexts are ever synchronized in our system; hence the following corollary to the above lemma states the preservation of heap bisimilarity in synchronized computations.

Corollary 4.7 (Preservation of Heap Bisimilarity in Synchronized Computations)**.** *If* $\left(\mathrm{H}_1, s_1\right)_{n_1} \longrightarrow_{\Phi} \left(\mathrm{H}_1', s_1'\right)_{n_1'}$, $\left(\mathrm{H}_2, s_2\right)_{n_2} \longrightarrow_{\Phi} \left(\mathrm{H}_2', s_2'\right)_{n_2'}$, $\bot, \Gamma, \mathcal{H}_1 \vdash_{\Phi\ell} s_1 : \tau_1$, $\bot, \Gamma, \mathcal{H}_2 \vdash_{\Phi\ell} s_2 : \tau_2$ *and* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$*, then there exists a* $\mathcal{H}_1'$ *and* $\mathcal{H}_2'$ *such that* $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\bot, \Gamma, \mathcal{H}_1' \vdash_{\Phi\ell} s_1' : \tau_1$, $\bot, \Gamma, \mathcal{H}_2' \vdash_{\Phi\ell} s_2' : \tau_2$ *and* $(\mathcal{H}_1', \mathrm{H}_1') \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2', \mathrm{H}_2')$.

Proof. By *sync*, SYNC and Lemma 4.6.

□

The following single-step strong bisimulation lemma states: given a pair of bisimilar configurations, if the first configuration takes a step, then the second configuration also takes a step, and bisimilarity is preserved in the resulting configurations.

Lemma 4.8 (λ^{sync}_{seq}: Strong Bisimulation, 1-step)**.** *If* $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_{\Phi} \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ *and* $\left(\mathcal{H}_1, \mathrm{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$ *then for some* H_2', e_2' *and* d_2', $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_{\Phi} \left(\mathrm{H}_2', e_2'\right)_{d_2'}$ *holds, and there exists a* $\mathcal{H}_1'$, $\mathcal{H}_2'$ *and* μ' *such that* $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\mu \subseteq \mu'$ *and* $\left(\mathcal{H}_1', \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', e_2'\right)_{d_2'}$.

Proof. By induction on the derivation of $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_{\Phi} \left(\mathrm{H}_1', e_1'\right)_{d_1'}$. Following are the possible syntax-directed semantics rules from Figure 8 that are applicable at the root of this derivation.

1. BINOP. So for some i_1, i_1' and v_1', $e_1 = i_1 \oplus i_1'$, $i_1 \oplus i_1' = v_1'$ and $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_{\Phi} \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ such that $\mathrm{H}_1' = \mathrm{H}_1$, $e_1' = v_1'$ and $d_1' = d_1 - 1$.
   By premise $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[13a] $d_1 = d_2 = \infty$ and for some i_2 and i_2', $e_2 = i_2 \oplus i_2'$. By BINOP and letting $i_2 \oplus i_2' = v_2'$, we get $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_{\Phi} \left(\mathrm{H}_2', e_2'\right)_{d_2'}$, the first requisite result, such that $\mathrm{H}_2' = \mathrm{H}_2$, $e_2' = v_2'$ and $d_2' = d_2 - 1 = \infty$. Then by Lemma 4.4[13b] $\left(\mathcal{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{d_2'}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2'\right)_{d_2'}$, that is, $\left(\mathcal{H}_1, \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2', e_2'\right)_{d_2'}$, the second requisite result.
2. LET. So for some x, v_1 and p_1, $e_1 = (\mathsf{let}\ x = v_1\ \mathsf{in}\ p_1)$ and $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_{\Phi} \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ such that $\mathrm{H}_1' = \mathrm{H}_1$, $e_1' = p_1[v_1/x]$ and $d_1' = d_1 - 1$.
   By premise $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[14a] $d_1 = d_2 = \infty$ and for some v_2 and p_2, $e_2 = (\mathsf{let}\ x = v_2\ \mathsf{in}\ p_2)$. By LET $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_{\Phi} \left(\mathrm{H}_2', e_2'\right)_{d_2'}$, the first requisite result, such that $\mathrm{H}_2' = \mathrm{H}_2$, $e_2' = e_{2_{next}}[v_2/x]$, and $d_2' = d_2 - 1 = \infty$. Then by Lemma 4.4[14b] $\left(\mathcal{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{d_2'}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2'\right)_{d_2'}$, that is, $\left(\mathcal{H}_1, \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2', e_2'\right)_{d_2'}$, the second requisite result.
3. IF-SYNC. So for some p, $\overline{i_k}$, v_1, b_1, p_{1_T} and p_{1_F}, $e_1 = \mathsf{if}^{v_1}_{\mathsf{p},i}\ b_1\ \mathsf{then}\ p_{1_T}\ \mathsf{else}\ p_{1_F}$ and for some n, $\Phi(\mathsf{p})(\overline{i_k}) = n$.
   By premise $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[15a] $d_1 = d_2 = \infty$ and for some v_2, b_2, p_{2_T} and p_{2_F}, $e_2 = \mathsf{if}^{v_2}_{\mathsf{p},\overline{i_k}}\ b_2\ \mathsf{then}\ p_{2_T}\ \mathsf{else}\ p_{2_F}$. Now there are two possible cases depending on the value of b_1:
   (a) $b_1 = \mathsf{true}$. As per IF-SYNC $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_{\Phi} \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ such that $\mathrm{H}_1' = \mathrm{H}_1$, $e_1' = \langle p_{1_T} \rangle^{v_1}$ and $d_1' = n$. There are again two possible cases depending on the value of b_2:
      i. $b_2 = \mathsf{true}$. By IF-SYNC $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_{\Phi} \left(\mathrm{H}_2', e_2'\right)_{d_2'}$, the first requisite result, such that $\mathrm{H}_2' = \mathrm{H}_2$, $e_2' = \langle p_{2_T} \rangle^{v_2}$, and $d_2' = n$.
      We know $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$; then by Lemma 4.4[15b] $\left(\mathcal{H}_1, e_1'\right)_{n} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{n}$, that is, $\left(\mathcal{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{d_2'}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2'\right)_{d_2'}$, that is, $\left(\mathcal{H}_1, \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2', e_2'\right)_{d_2'}$, the second requisite result.
      ii. $b_2 = \mathsf{false}$. Analogous to the subsubcase 3(a)i above.
      By IF-SYNC $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_{\Phi} \left(\mathrm{H}_2', e_2'\right)_{d_2'}$, the first requisite result, such that $\mathrm{H}_2' = \mathrm{H}_2$, $e_2' = \langle p_{2_F} \rangle^{v_2}$ and $d_2' = n$.
      We know $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$; then by Lemma 4.4[15b] $\left(\mathcal{H}_1, e_1'\right)_{n} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{n}$, that is, $\left(\mathcal{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{d_2'}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2'\right)_{d_2'}$, that is, $\left(\mathcal{H}_1, \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2', e_2'\right)_{d_2'}$, the second requisite result.
   (b) $b_2 = \mathsf{false}$. Analogous to the subcase 3a above.
4. APP-SYNC. (Similar to case 3 above.) So for some p, $\overline{i_k}$, v_1, x_1, p_1 and $v_{1_{arg}}$, $e_1 = (\lambda x_1.\, p_1)\,(v_{1_{arg}})^{v_1}_{\mathsf{p},\overline{i_k}}$ and for some n, $\Phi(\mathsf{p})(\overline{i_k}) = n$.
   By premise $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[16a] $d_1 = d_2 = \infty$ and for some v_2, x_2, p_2 and $v_{2_{arg}}$, $e_2 = (\lambda x_2.\, p_2)\,(v_{2_{arg}})^{v_2}_{\mathsf{p},\overline{i_k}}$.
   Then as per APP-SYNC $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_{\Phi} \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ such that $\mathrm{H}_1' = \mathrm{H}_1$, $e_1' = \langle p_1[v_{1_{arg}}/x_1] \rangle^{v_1}$ and $d_1' = n$; and $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_{\Phi} \left(\mathrm{H}_2', e_2'\right)_{d_2'}$, the first requisite result, such that $\mathrm{H}_2' = \mathrm{H}_2$, $e_2' = \langle p_2[v_{2_{arg}}/x_2] \rangle^{v_2}$, and $d_2' = n$.
   We know $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$; then by Lemma 4.4[16b] $\left(\mathcal{H}_1, e_1'\right)_{n} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{n}$, that is, $\left(\mathcal{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{d_2'}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2'\right)_{d_2'}$, that is, $\left(\mathcal{H}_1, \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2', e_2'\right)_{d_2'}$, the second requisite result.

5. IF. So for some p, $\overline{i_k}$, v_1, b_1, p_{1_T} and p_{1_F}, $e_1 = \mathsf{if}^{v_1}_{\mathsf{p},\overline{i_k}}\ b_1\ \mathsf{then}\ p_{1_T}\ \mathsf{else}\ p_{1_F}$.
   By premise $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[15a] $d_1 = d_2 = \infty$ and for some v_2, b_2, p_{2_T} and p_{2_F}, $e_2 = \mathsf{if}^{v_2}_{\mathsf{p},\overline{i_k}}\ b_2\ \mathsf{then}\ p_{2_T}\ \mathsf{else}\ p_{2_F}$. Now there are two possible cases depending on the value of b_1:
   (a) $b_1 = \mathsf{true}$. As per IF $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}'_1, e'_1\right)_{d'_1}$ such that $\mathrm{H}'_1 = \mathrm{H}_1$, $e'_1 = p_{1_T}$, $d'_1 = d_1 - 1 = \infty$ and $\Phi(\mathsf{p}) = \varnothing$. Then by Lemma 4.4[15c] $b_1 = b_2 = \mathsf{true}$.
   By IF $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, the first requisite result, such that $\mathrm{H}'_2 = \mathrm{H}_2$, $e'_2 = p_{2_T}$ and $d'_2 = d_2 - 1 = \infty$. Then by Lemma 4.4[15c] $\left(\mathcal{H}_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e'_2\right)_{d'_2}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e'_2\right)_{d'_2}$, that is, $\left(\mathcal{H}_1, \mathrm{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}'_2, e'_2\right)_{d'_2}$, the second requisite result.
   (b) $b_1 = \mathsf{false}$. Analogous to the subcase 5a.
6. APP. (Similar to case 5 above.) So for some p, $\overline{i_k}$, v_1, x_1, p_1 and $v_{1_{arg}}$, $e_1 = (\lambda x_1.\, p_1)\,(v_{1_{arg}})^{v_1}_{\mathsf{p},\overline{i_k}}$.
   By premise $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[16a] $d_1 = d_2 = \infty$ and for some v_2, x_2, p_2 and $v_{2_{arg}}$, $e_2 = (\lambda x_2.\, p_2)\,(v_{2_{arg}})^{v_2}_{\mathsf{p},\overline{i_k}}$.
   As per APP $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}'_1, e'_1\right)_{d'_1}$ such that $\mathrm{H}'_1 = \mathrm{H}_1$, $e'_1 = p_1[v_{1_{arg}}/x_1]$, $d'_1 = d_1 - 1 = \infty$ and $\Phi(\mathsf{p}) = \varnothing$; and $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, the first requisite result, such that $\mathrm{H}'_2 = \mathrm{H}_2$, $e'_2 = p_2[v_{2_{arg}}/x_2]$ and $d'_2 = d_2 - 1 = \infty$. Then by Lemma 4.4[16c] $\left(\mathcal{H}_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e'_2\right)_{d'_2}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e'_2\right)_{d'_2}$, that is, $\left(\mathcal{H}_1, \mathrm{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}'_2, e'_2\right)_{d'_2}$, the second requisite result.
7. REF. So for some v_1, $e_1 = \mathsf{ref}\ v_1$, and for some fresh loc_1, $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}'_1, e'_1\right)_{d'}$ such that $\mathrm{H}'_1 = \mathrm{H}_1 \cup \{loc_1 \mapsto v_1\}$, $e'_1 = loc_1$ and $d'_1 = d_1 - 1$.
   By premise $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[17a] $d_1 = d_2 = \infty$ and for some v_2, $e_2 = \mathsf{ref}\ v_2$. By REF for some fresh loc_2, $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, the first requisite result, such that $\mathrm{H}'_2 = \mathrm{H}_2 \cup \{loc_2 \mapsto v_2\}$, $e'_2 = loc_2$ and $d'_2 = d_2 - 1 = \infty$. Now loc_1 and loc_2 are fresh implying $loc_1 \notin dom(\mathrm{H}_1)$ and $loc_2 \notin dom(\mathrm{H}_2)$. We know by premise $\left(\mathcal{H}_1, \mathrm{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$; then by Lemma 4.4[17b] for some τ' and ℓ', $\tau = \tau'\ ref^{\ell'}$ and $\left(\mathcal{H}'_1, \mathrm{H}'_1, loc_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}'_2, \mathrm{H}'_2, loc_2\right)_{d'_2}$, that is, $\left(\mathcal{H}'_1, \mathrm{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}'_2, \mathrm{H}'_2, e'_2\right)_{d'_2}$, the second requisite result, where $\mathcal{H}'_1 = \mathcal{H}_1 \cup \{loc_1 \mapsto \tau'\}$, $\mathcal{H}'_2 = \mathcal{H}_2 \cup \{loc_2 \mapsto \tau'\}$ and $\mu' = \mu \cup \{loc_1 \mapsto loc_2\}$.
8. DEREF. So for some loc_1, $e_1 = !loc_1$, and for $\mathrm{H}_1(loc_1) = v_1$, $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}'_1, e'_1\right)_{d'_1}$ such that $\mathrm{H}'_1 = \mathrm{H}_1$, $e'_1 = v_1$, and $d'_1 = d_1 - 1$.
   By premise $\left(\mathcal{H}_1, \mathrm{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[18a] $d_1 = d_2 = \infty$ and for some $loc_2 \in dom(\mathrm{H}_2)$, $e_2 = !loc_2$. By DEREF for $\mathrm{H}_2(loc_2) = v_2$, $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, the first requisite result, such that $\mathrm{H}'_2 = \mathrm{H}_2$, $e'_2 = v_2$ and $d'_2 = d_2 - 1 = \infty$. Then by Lemma 4.4[18b] $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{H}_1(loc_1)\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{H}_2(loc_2)\right)_{d'_2}$, that is, $\left(\mathcal{H}_1, \mathrm{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}'_2, e'_2\right)_{d'_2}$ holds, the second requisite result.
9. ASSIGN. So for some loc_1 and v_1, $e_1 = (loc_1 := v_1)$, $loc_1 \in dom(\mathrm{H}_1)$ and $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}'_1, e'_1\right)_{d'_1}$ such that $\mathrm{H}'_1 = \mathrm{H}_1[loc_1 \mapsto v_1]$, $e'_1 = v_1$ and $d'_1 = d_1 - 1$.
   By premise $\left(\mathcal{H}_1, \mathrm{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$ holds; then by Lemma 4.4[19a] $d_1 = d_2 = \infty$ and for some $loc_2 \in dom(\mathrm{H}_2)$, and v_2, $e_2 = (loc_2 := v_2)$. By ASSIGN $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, the first requisite result, such that $\mathrm{H}'_2 = \mathrm{H}_2[loc_2 \mapsto v_2]$, $e'_2 = v_2$, and $d'_2 = d_2 - 1 = \infty$. Then by Lemma 4.4[19b] $\left(\mathcal{H}_1, \mathrm{H}_1[loc_1 \mapsto v_1], v_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2[loc_2 \mapsto v_2], v_2\right)_{d'_2}$, that is, $\left(\mathcal{H}_1, \mathrm{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}'_2, e'_2\right)_{d'_2}$ holds, the second requisite result.
10. SYNC. So for some s_1, p_1, v_1, e'_1, s'_1, p'_1 and n, $e_1 = s_1 = \langle p_1 \rangle^{v_1}$, $d_1 = n$, $\left(\mathrm{H}_1, p_1\right)_n \longrightarrow_\Phi \left(\mathrm{H}'_1, p'_1\right)_{n-1}$ and $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}'_1, e'_1\right)_{d'_1}$ such that $e'_1 = s'_1 = \langle p'_1 \rangle^{v_1}$ and $d'_1 = d_1 - 1$; hence $d_1 > 0$, that is, $n > 0$.
    By premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ and $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ hold; then by Lemma 4.4[20a] for some p_2, v_2 and s_2, $e_2 = s_2 = \langle p_2 \rangle^{v_2}$ and $d_1 = d_2 = n$, and by Definition 4.3[3a] $\bot, \emptyset, \mathcal{H}_2 \vdash_{\Phi\ell} \mathrm{H}_2$.
    We know $\left(\mathcal{H}_1, s_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, s_2\right)_{d_2}$; then by Lemma 4.4[6] $\bot, \emptyset, \mathcal{H}_1 \vdash_{\Phi\ell} s_1 : \tau$ and $\bot, \emptyset, \mathcal{H}_2 \vdash_{\Phi\ell} s_2 : \tau$; hence $\bot, \emptyset, \mathcal{H}_2 \vdash_{\Phi\ell} (\mathrm{H}_2, s_2)_n : \tau$ holds. By Main Lemma 3.8 (Progress) there exists a H'_2, e'_2 and d'_2 such that $(\mathrm{H}_2, s_2)_n \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, that is, $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, the first requisite result.
    By SYNC $d'_2 = n - 1$, that is, $d'_2 = d_2 - 1$, and for some s'_2 and p'_2, $e'_2 = s'_2 = \langle p'_2 \rangle^{v_2}$. By Corollary 4.7 (Preservation of Heap Bisimilarity under Synchronized Computation) there exists a $\mathcal{H}'_1$ and $\mathcal{H}'_2$ such that $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$, $\bot, \emptyset, \mathcal{H}'_1 \vdash_{\Phi\ell} e'_1 : \tau$, $\bot, \emptyset, \mathcal{H}'_2 \vdash_{\Phi\ell} e'_2 : \tau$ and $(\mathcal{H}'_1, \mathrm{H}'_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}'_2, \mathrm{H}'_2)$.
    By Lemma 4.4[7] $\left(\mathcal{H}'_1, s'_1\right)_{n-1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}'_2, s'_2\right)_{n-1}$, that is, $\left(\mathcal{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}'_2, e'_2\right)_{d'_2}$. Knowing $(\mathcal{H}'_1, \mathrm{H}'_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}'_2, \mathrm{H}'_2)$ by Definition 4.3[4] we get $\left(\mathcal{H}_1, \mathrm{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}'_2, e'_2\right)_{d'_2}$, the second requisite result.
11. SYNC-PAD. Similar to the case 10 above.
12. SYNC-DONE. So for some v'_1 and v_1, $e_1 = \langle v'_1 \rangle^{v_1}$, $d_1 = 0$ and $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}'_1, e'_1\right)_{d'_1}$ such that $\mathrm{H}'_1 = \mathrm{H}_1$, $e'_1 = v'_1$ and $d'_1 = \infty$.
    By premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ and $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ hold; then by Lemma 4.4[20a] for some p_2, and v_2, $e_2 = \langle p_2 \rangle^{v_2}$ and $d_1 = d_2 = 0$. Now there are two possible cases depending on whether p_2 is a value or not:
    (a) $p_2 = v'_2$, for some v'_2. By SYNC-DONE $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, the first requisite result, such that $\mathrm{H}'_2 = \mathrm{H}_2$, $e'_2 = v'_2$ and $d'_2 = \infty$. Then by Lemma 4.4[20b] $\left(\mathcal{H}_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e'_2\right)_{d'_2}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e'_2\right)_{d'_2}$, that is,

$\left(\mathcal{H}_1, \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2', e_2'\right)_{d_2'}$, the second requisite result.

(b) p_2 is not a value. By SYNC-ABORT $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}_2', e_2'\right)_{d_2'}$, the first requisite result, such that $\mathrm{H}_2' = \mathrm{H}_2$, $e_2' = v_2$ and $d_2' = \infty$. Then by Lemma 4.4[20b] $\left(\mathcal{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2'\right)_{d_2'}$ holds, which in conjunction with the premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ implies $\left(\mathcal{H}_1, \mathrm{H}_1, e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2'\right)_{d_2'}$, that is, $\left(\mathcal{H}_1, \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2', e_2'\right)_{d_2'}$, the second requisite result.

13. SYNC-ABORT. Analogous to the case 12 above given Lemma 4.4[20b].

14. CONTEXT. So for some R_1, $e_{1_{sub}}$ and $e_{1_{sub}}'$, $e_1 = \mathrm{R}_1[e_{1_{sub}}]$ and for some $e_{1_{sub}}'$, $\left(\mathrm{H}_1, e_{1_{sub}}\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}_1', e_{1_{sub}}'\right)_{d_1'}$ and $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ such that $e_1' = \mathrm{R}_1[e_{1_{sub}}']$.

By premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ and $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ hold; then by Lemma 4.4[21a] for some R_2, $e_{2_{sub}}$ and τ_{sub}, $e_2 = \mathrm{R}_2[e_{2_{sub}}]$ and $\left(\mathcal{H}_1, e_{1_{sub}}\right)_{d_1} \sim^{\emptyset\tau_{sub}}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_{2_{sub}}\right)_{d_2}$.

Knowing $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$, by Definition 4.3[4] we get $\left(\mathcal{H}_1, \mathrm{H}_1, e_{1_{sub}}\right)_{d_1} \sim^{\emptyset\tau_{sub}}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_{2_{sub}}\right)_{d_2}$. By assumption we know $\left(\mathrm{H}_1, e_{1_{sub}}\right)_{d_1} \longrightarrow_\Phi \left(\mathrm{H}_1', e_{1_{sub}}'\right)_{d_1'}$. Then by induction hypothesis for some H_2', $e_{2_{sub}}'$ and d_2', $\left(\mathrm{H}_2, e_{2_{sub}}\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}_2', e_{2_{sub}}'\right)_{d_2'}$, and there exists a $\mathcal{H}_1'$, $\mathcal{H}_2'$ and μ', such that $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\mu \subseteq \mu'$ and $\left(\mathcal{H}_1', \mathrm{H}_1', e_{1_{sub}}'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', e_{2_{sub}}'\right)_{d_2'}$.

Then by CONTEXT $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_\Phi \left(\mathrm{H}_2', e_2'\right)_{d_2'}$, the first requisite result, such that $e_2' = \mathrm{R}_2[e_{2_{sub}}']$. Finally by Lemma 4.4[21b] $\left(\mathcal{H}_1', \mathrm{H}_1', \mathrm{R}_1[e_{1_{sub}}']\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', \mathrm{R}_2[e_{2_{sub}}']\right)_{d_2'}$, that is, $\left(\mathcal{H}_1', \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', e_2'\right)_{d_2'}$ holds, the second requisite result.

□

The following n-step strong bisimulation lemma states: given a pair of bisimilar configurations, if the first configuration takes n steps, then the second configuration also take n steps, and bisimilarity is preserved in the resulting configurations.

Lemma 4.9 (λ^{sync}_{seq}: Strong Bisimulation, n-steps)**.** *If* $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow^n_\Phi \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ *and* $\left(\mathcal{H}_1, \mathrm{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$ *then for some* H_2', e_2' *and* d_2', $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow^n_\Phi \left(\mathrm{H}_2', e_2'\right)_{d_2'}$ *holds, and there exists a* $\mathcal{H}_1'$, $\mathcal{H}_2'$ *and* μ' *such that* $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\mu \subseteq \mu'$ *and* $\left(\mathcal{H}_1', \mathrm{H}_1', e_1'\right)_{d_1'} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}_2', \mathrm{H}_2', e_2'\right)_{d_2'}$.

Proof. By induction on the derivation of $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow^n_\Phi \left(\mathrm{H}_1', e_1'\right)_{d_1'}$ given Lemma 4.8 (λ^{sync}_{seq}: Strong Bisimulation, 1-step). □

The strong bisimulation lemma then entails the following strong noninterference property. It states: if one run of a program converges to a low integral or boolean value in a certain number of steps, then a second run of that program, but possibly differing in high values from the first run, also converges to the same value and in the same number of steps as the first one.

Theorem 4.10 (λ^{sync}_{seq}: Strong Noninterference)**.** *If* $\bot, \{\overline{x_k \mapsto \tau_k}\}, \emptyset \vdash^{low}_{\Phi\ell} p : \hat{\tau}$, $\bot, \emptyset, \emptyset \vdash^{high}_{\Phi\ell} \overline{v_k, v_k' : \tau_k}$, $p_1 = p[\overline{v_k/x_k}]$, $p_2 = p[\overline{v_k'/x_k}]$, *and* p_1 *given* Φ *converges to a value* $\hat{v}$ *in* n *steps, then* p_2 *given* Φ *converges to the same value* $\hat{v}$ *in* n *steps.*

Proof. By premise, for some H_1 and d_1, $\left(\emptyset, p_1\right)_\infty \longrightarrow^n_\Phi \left(\mathrm{H}_1, \hat{v}\right)_{d_1}$ and Definition 4.3[1,3,4] $\left(\emptyset, \emptyset, p_1\right)_\infty \sim^{\emptyset\hat{\tau}}_{\Phi\ell\emptyset} \left(\emptyset, \emptyset, p_2\right)_\infty$. Then by Lemma 4.9 ($\lambda^{sync}_{seq}$: Strong Bisimulation, n-steps) for some H_2, e_2 and d_2, $\left(\emptyset, p_2\right)_\infty \longrightarrow^n_\Phi \left(\mathrm{H}_2, e_2\right)_{d_2}$, and there exists a $\mathcal{H}_1$, $\mathcal{H}_2$ and μ such that $\emptyset \subseteq \mathcal{H}_1$, $\emptyset \subseteq \mathcal{H}_2$, $\emptyset \subseteq \mu$ and $\left(\mathcal{H}_1, \mathrm{H}_1, \hat{v}\right)_{d_1} \sim^{\emptyset\hat{\tau}}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$. By Definition 4.3[4,1] $\left(\mathcal{H}_1, \hat{v}\right)_{d_1} \sim^{\emptyset\hat{\tau}}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$ and $d_1 = d_2 = \infty$; then by Lemma 4.4[11c] $e_2 = \hat{v}$. That is, p_2 given Φ converges to the value $\hat{v}$ in n steps, the requisite result. □

The following corollary to the above lemma states the strong noninterference property in presence of divergence.

Corollary 4.11 (λ^{sync}_{seq}: Strong Noninterference under Divergence)**.** *If* $\bot, \{\overline{x_k \mapsto \tau_k}\}, \emptyset \vdash^{low}_{\Phi\ell} p : \hat{\tau}$, $\bot, \emptyset, \emptyset \vdash^{high}_{\Phi\ell} \overline{v_k, v_k' : \tau_k}$, $p_1 = p[\overline{v_k/x_k}]$, $p_2 = p[\overline{v_k'/x_k}]$, *and* p_1 *given* Φ *diverges, then* p_2 *given* Φ *diverges as well.*

Proof. By Lemma 3.6 (Value Substitution in a Program), Theorem 3.13 (Type Safety) and Theorem 4.10 (λ^{sync}_{seq}: Strong Noninterference). □

The combination of Theorem 4.10 and Corollary 4.11 imply the security of timing and termination channels in λ^{sync}_{seq}.

Definition 4.12 (Secure Program)**.** *A program* p *given a table of sync timer functions* Φ *is secure iff for* $free(p) = \{\overline{x_k}\}$ *and for all* $\overline{i_k}$ *and* $\overline{i_k'}$, *either both* $p[\overline{i_k/x_k}]$ *and* $p[\overline{i_k'/x_k}]$ *given* Φ *converge to the same value in the same number of steps, or both diverge.*

Definition 4.13 (Secure-Typed Program)**.** *A program* p *given a table of sync timer functions* Φ *is secure-typed iff* $\bot, \{\overline{x_k \mapsto int^{\ell_k}}\}, \emptyset \vdash^{low}_{\Phi\ell} p : \hat{\tau}$, *where* $free(p) = \{\overline{x_k}\}$, $\overline{\ell_k} \not\leq \ell$ *and some* $\hat{\tau}$.

The following corollary to the above noninterference lemmas directly states the security guarantee of λ^{sync}_{seq} – λ^{sync}_{seq} is secure with respect to external timing channels.

Theorem 4.14 (Security of λ^{sync}_{seq})**.** *Secure-typed programs are secure.*

Proof. By Theorem 4.10 (λ^{sync}_{seq}: Strong Noninterference) and Corollary 4.11 (λ^{sync}_{seq}: Strong Noninterference under Divergence). □

5. The $\lambda^{sync}_{conc_{\mathcal{D}}}$ Runtime System

We now show how the results of the previous section can directly generalize to the case of concurrent execution, with a deterministic scheduler in this section, and with a nondeterministic scheduler in the following section. Here we formalize the $\lambda^{sync}_{conc_{\mathcal{D}}}$ system as an extension to λ^{sync}_{seq}. The syntax for $\lambda^{sync}_{conc_{\mathcal{D}}}$ appears in Figure 11 and its small-step operational semantics in Figure 12, as addendums to Figures 7 and 8 respectively. The trace $\mathtt{t}$ denotes the scheduling history – the interleavings thus far – of a pool of threads; it represents the internal timing behavior of concurrent programs, as discussed in Section 2. The scheduler $\mathcal{D}$ denotes an arbitrary deterministic

$$
\begin{array}{lcll}
\mathtt{t} & ::= & \epsilon \mid \mathtt{t}, n & \textit{trace} \\
\hat{\mathrm{H}} & ::= & \{\overline{loc}\} \to \{\overline{\hat{v}}\} & \textit{heap with only integer/boolean values} \\
\mathcal{D} & : & \mathbb{N} \times \{\overline{\mathtt{t}}\} \times \{\overline{\hat{\mathrm{H}}}\} \to \mathbb{N} & \textit{deterministic scheduler} \\
\zeta & ::= & (e, d) & \textit{thread} \\
\mathrm{T} & ::= & \overline{\zeta} & \textit{pool of threads} \\
\mathrm{C} & ::= & \left(\mathrm{H}, \mathrm{T}\right)_{\mathtt{t}} & \textit{concurrent runtime configuration} \\
\mathbb{L} & ::= & \{\overline{loc}\} & \textit{set of "low integer/boolean" heap locations} \\
\Upsilon & ::= & \overline{\tau} & \textit{pool of types} \\
\hat{\mathcal{H}} & ::= & \{\overline{loc}\} \to \{\overline{\hat{\tau}}\} & \textit{type of heap with only int/bool types}
\end{array}
$$

Figure 11. $\lambda^{sync}_{conc_{\mathcal{D}}}$: Syntax and Type Grammar

$$
\textsc{thread}\quad \frac{\zeta = (e, d) \qquad \left(\mathrm{H}, e\right)_d \longrightarrow_{\Phi} \left(\mathrm{H}', e'\right)_{d'} \qquad \zeta' = (e', d')}{(\mathrm{H}, \zeta) \longrightarrow_{\Phi} (\mathrm{H}', \zeta')}
$$

$$
\textsc{thread-pool\_next-schedule}\quad \frac{\mathrm{T}[i] = \zeta_i \qquad (\mathrm{H}, \zeta_i) \longrightarrow_{\Phi} (\mathrm{H}', \zeta_i')}{\left(\mathrm{H}, \mathrm{T}\right)_{\mathtt{t}} \longrightarrow_{\Phi i} \left(\mathrm{H}', \mathrm{T}[i \mapsto \zeta_i']\right)_{\mathtt{t}, i}}
$$

$$
\textsc{thread-pool\_}\mathcal{D}\quad \frac{|\,\mathrm{T}\,| = k \qquad \hat{\mathrm{H}}_{low} = \mathrm{H}\lceil\mathbb{L} \qquad \mathcal{D}(k, \mathtt{t}, \hat{\mathrm{H}}_{low}) = i \qquad \left(\mathrm{H}, \mathrm{T}\right)_{\mathtt{t}} \longrightarrow_{\Phi i} \left(\mathrm{H}', \mathrm{T}'\right)_{\mathtt{t}'}}{\left(\mathrm{H}, \mathrm{T}\right)_{\mathtt{t}} \longrightarrow_{\Phi\mathbb{L}\mathcal{D}} \left(\mathrm{H}', \mathrm{T}'\right)_{\mathtt{t}'}}
$$

Figure 12. $\lambda^{sync}_{conc_{\mathcal{D}}}$: Syntax-Directed Small-Step Operational Semantics

scheduler (possibly chosen by an attacker), and is parameterized over a 3-tuple: the number of threads in the thread pool, the trace so far, and a low portion of the heap with only integer or boolean values; it returns an index into the thread pool for the next scheduled thread. Each thread ζ is a pair of an expression e and a down-counter d.

The small-step reduction relations $\longrightarrow_{\Phi i}$ and $\longrightarrow_{\Phi\mathbb{L}\mathcal{D}}$, for a table of sync timer functions Φ, the next scheduled thread i, a set of low integer/boolean heap locations $\mathbb{L}$ (to be fed to the scheduler), and a deterministic scheduler $\mathcal{D}$, are each defined over configurations $\left(\mathrm{H}, \mathrm{T}\right)_{\mathtt{t}}$, where the heap H is shared amongst all threads in the thread pool T; the n-step reflexive and transitive closure of $\longrightarrow_{\Phi\mathbb{L}\mathcal{D}}$ is denoted as $\longrightarrow^{n}_{\Phi\mathbb{L}\mathcal{D}}$. The down-counter in a thread is decremented via the THREAD rule only when the thread is under execution, meaning only the cumulative scheduled time of a thread, and not the wall-clock time, is counted towards the synchronized execution time of a branching statement in that thread. The restricted-to operator $\lceil$ is defined as: $\mathrm{H}\lceil\mathbb{L} = \mathrm{H}'$ iff $\mathrm{H}' \subseteq \mathrm{H}$ and $dom(\mathrm{H}) = \mathbb{L}$, and the indexing operation on a thread pool T is defined as: $\mathrm{T}[i] = \zeta_i$ iff $\mathrm{T} = \overline{\zeta_k}$, for some $\overline{\zeta_k}$, and $1 \leq i \leq k$. The thread update operation $\mathrm{T}[i \mapsto \zeta_i']$, where i denotes the index of the thread to be updated to ζ_i', is defined to be equal to T' iff $\mathrm{T} = \zeta_1, \ldots, \zeta_i, \ldots, \zeta_k$, for some $\overline{\zeta_k}$, and $\mathrm{T}' = \zeta_1, \ldots, \zeta_i', \ldots, \zeta_k$; while the number of threads in a thread pool is defined as: $|\,\mathrm{T}\,| = k$ iff $\mathrm{T} = \overline{\zeta_k}$, for some $\overline{\zeta_k}$. For technical ease the scheduler is invoked after each step, via the rule THREAD-POOL_$\mathcal{D}$; more liberal scheduling policies are subsumed by this approach.

$\lambda^{sync}_{conc_{\mathcal{D}}}$ does not support dynamic thread creation, or explicit thread yielding. The former can be incorporated using the thread pool partitioning technique of [34]; while the latter is directly encodable in our system. A thread ζ wanting to yield control can do so by simply setting a specific location in the portion of the heap passed to the scheduler – $\hat{\mathrm{H}}_{low}$ in the rule THREAD-POOL_$\mathcal{D}$ – thus informing the scheduler of its intent to yield. As only low heap locations are observable to the scheduler, yielding is disallowed under high guards – yielding has direct impact on the interleavings of threads, and hence must be prohibited in high contexts to prevent internal timing leaks; [34] enforces the same restriction.

5.1 Security Properties of $\lambda^{sync}_{conc_{\mathcal{D}}}$

We now formally establish the security guarantee of $\lambda^{sync}_{conc_{\mathcal{D}}}$. We show $\lambda^{sync}_{conc_{\mathcal{D}}}$ is secure with respect to both external and internal timing channels by proving a strong noninterference result between high data and the external and internal timing behavior of concurrent programs. We demonstrate how executions of two pools of threads, which differ only in high values, are strongly bisimilar. Strong bisimilarity implies isomorphism of the pools of threads at all intermediate steps of execution; as for λ^{sync}_{seq}, the term *strong bisimilarity* indicates the lock-stepness of bisimilar executions. The bisimulation relation, defined below, essentially requires all low values in corresponding threads as well as the interleavings of the concurrent executions to be identical, while allowing high values to differ; it is an addendum to λ^{sync}_{seq}'s bisimulation relation (Definition 4.3). We define the predicate $seclevel(\mathcal{H}) \leq \ell$ to hold iff $\forall \tau \in range(\mathcal{H}).\ seclevel(\tau) \leq \ell$.

Definition 5.1 ($\lambda^{sync}_{conc_{\mathcal{D}}}$: Bisimulation Relation)**.**

1. *(Threads).* $(\mathcal{H}_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \zeta_2)$ *iff* $\zeta_1 = (e_1, d_1)$, $\zeta_2 = (e_2, d_2)$ *and* $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$.
2. *(Pools of Threads).* $\left(\mathcal{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$ *iff* $\mathtt{t}_1 = \mathtt{t}_2$ *and for some* $\overline{\zeta_k}$, $\overline{\zeta_k'}$ *and* $\overline{\tau_k}$,
   (a) $\mathrm{T}_1 = \overline{\zeta_k}$, $\mathrm{T}_2 = \overline{\zeta_k'}$, *and* $\Upsilon = \overline{\tau_k}$; *and*
   (b) $\forall 1 \leq i \leq k.\ (\mathcal{H}_1, \zeta_i) \sim^{\Gamma\tau_i}_{\Phi\ell\mu} (\mathcal{H}_2, \zeta_i')$.
3. *(Heaps).* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{H}_2)$ *iff for some* $\hat{\mathcal{H}}$,
   (a) $\mathcal{H}_1\lceil\mathbb{L} = \mathcal{H}_2\lceil\mathbb{L} = \hat{\mathcal{H}}$, *and* $seclevel(\hat{\mathcal{H}}) \leq \ell$; *and*
   (b) $\forall loc \in \mathbb{L}.\ loc \mapsto loc \in \mu$, *and* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$.
4. *(Heaps, Threads).* $(\mathcal{H}_1, \mathrm{H}_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2, \zeta_2)$ *iff* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ *and* $(\mathcal{H}_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{T}_2)$.
5. *(Heaps, Pools of Threads).* $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$ *iff* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{H}_2)$ *and* $\left(\mathcal{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$.

As discussed in Section 2, the trace of a concurrent execution constitutes an internal timing channel, and hence must exhibit low behavior; accordingly the case 2 above requires the traces of bisimilar configurations, $\mathtt{t}_1$ and $\mathtt{t}_2$, to be identical. Further the corresponding threads in bisimilar thread pools are required to be bisimilar. The portions of bisimilar heaps to be passed to the scheduler, as indicated by $\mathbb{L}$, are restricted to contain only low integral or boolean values by the condition 3a – low function values may have high values wrapped inside them as per Definition 4.3[1], while low heap locations may store high values as implied by the type rule *location*; hence neither is guaranteed to be independent of high data, and thus cannot be passed to the scheduler without potentially violating noninterference. The other two parameters to the scheduler, the number of threads in the thread pool and the trace, are both low, thus guaranteeing the scheduler behavior is low as well. Further the above bisimulation relation is scheduler-independent, akin to the one in [38].

Lemma 5.2 ($\lambda^{sync}_{conc_{\mathcal{D}}}$: Properties of Bisimulation Relation)**.**

1. *(Symmetry).*
   (a) *(Heaps, Threads). If* $(\mathcal{H}_1, \mathrm{H}_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2, \zeta_2)$ *then* $(\mathcal{H}_2, \mathrm{H}_2, \zeta_2) \sim^{\Gamma\tau}_{\Phi\ell\mu^{-1}} (\mathcal{H}_1, \mathrm{H}_1, \zeta_1)$.

(b) (Heaps, Pools of Threads). If $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$ *then* $\left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu^{-1}\mathbb{L}} \left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1}$.

2. *(Reflexivity).*
   (a) (Threads). If $\zeta = (e, d)$ *and* $\left(\mathcal{H}, e\right)_d \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}, e\right)_d$ *then* $(\mathcal{H}, \zeta) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}, \zeta)$.
   (b) (Pool of Threads). If $\mathrm{T} = \overline{\zeta_k}$, $\Upsilon = \overline{\tau_k}$ *and* $\forall 1 \leq i \leq k$. $(\mathcal{H}, \zeta_i) \sim^{\Gamma\tau_i}_{\Phi\ell\mu} (\mathcal{H}, \zeta_i)$ *then for any* $\mathtt{t}$, $\left(\mathcal{H}, \mathrm{T}\right)_{\mathtt{t}} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu} \left(\mathcal{H}, \mathrm{T}\right)_{\mathtt{t}}$.
   (c) (Heaps). If $(\mathcal{H}, \mathrm{H}) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}, \mathrm{H})$, $\mathcal{H}\lceil\mathbb{L} = \hat{\mathcal{H}}$, *seclevel*$(\hat{\mathcal{H}}) \leq \ell$ *and* $\forall loc \in \mathbb{L}$. $loc \mapsto loc \in \mu$, *then* $(\mathcal{H}, \mathrm{H}) \sim^{\Gamma}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}, \mathrm{H})$.
   (d) (Heaps, Threads). If $\zeta = (e, d)$, $\left(\mathcal{H}, e\right)_d \sim^{\Gamma\tau}_{\Phi\ell\mu} \left(\mathcal{H}, e\right)_d$ *and* $(\mathcal{H}, \mathrm{H}) \sim^{\Gamma}_{\Phi\ell\mu} (\mathcal{H}, \mathrm{H})$ *then* $(\mathcal{H}, \mathrm{H}, \zeta) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}, \mathrm{H}, \zeta)$.
   (e) (Heaps, Pools of Threads). If $\left(\mathcal{H}, \mathrm{T}\right)_{\mathtt{t}} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu} \left(\mathcal{H}, \mathrm{T}\right)_{\mathtt{t}}$ *and* $(\mathcal{H}, \mathrm{H}) \sim^{\Gamma}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}, \mathrm{H})$ *then* $\left(\mathcal{H}, \mathrm{H}, \mathrm{T}\right)_{\mathtt{t}} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} \left(\mathcal{H}, \mathrm{H}, \mathrm{T}\right)_{\mathtt{t}}$.
3. *(Transitivity).*
   (a) (Heaps, Threads). If $(\mathcal{H}_1, \mathrm{H}_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2, \zeta_2)$ *and* $(\mathcal{H}_2, \mathrm{H}_2, \zeta_2) \sim^{\Gamma\tau}_{\Phi\ell\mu'} (\mathcal{H}_3, \mathrm{H}_3, \zeta_3)$ *then for* $\mu'' = \mu' \circ \mu$, $(\mathcal{H}_1, \mathrm{H}_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu''} (\mathcal{H}_3, \mathrm{H}_3, \zeta_3)$.
   (b) (Heaps, Pools of Threads). If $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$ *and* $\left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \left(\mathcal{H}_3, \mathrm{H}_3, \mathrm{T}_3\right)_{\mathtt{t}_3}$ *then for* $\mu'' = \mu' \circ \mu$, $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu''\mathbb{L}} \left(\mathcal{H}_3, \mathrm{H}_3, \mathrm{T}_3\right)_{\mathtt{t}_3}$.
4. *(Extension of Types of Heaps and μ).*
   (a) (Threads). If $(\mathcal{H}_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \zeta_2)$, $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$ *and* $\mu \subseteq \mu'$ *then* $(\mathcal{H}'_1, \zeta_1) \sim^{\Gamma\tau}_{\Phi\ell\mu'} (\mathcal{H}'_2, \zeta_2)$.
   (b) (Pools of Threads). If $\left(\mathcal{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$, $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$ *and* $\mu \subseteq \mu'$ *then* $\left(\mathcal{H}'_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu'} \left(\mathcal{H}'_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$.
5. *(Equality of Int/Bool Typed "Low" Sub-Heaps). If* $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\Gamma}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{H}_2)$ *then* $\mathrm{H}_1\lceil\mathbb{L} = \mathrm{H}_2\lceil\mathbb{L}$.

Proof. 1. *(Symmetry).*
   (a) *(Heaps, Threads).* By Definition 5.1[4,1] and Lemma 4.4[1].
   (b) *(Heaps, Pools of Threads).* By Definition 5.1[5,2] and Lemma 5.2[1a].
2. *(Reflexivity).* By Definition 5.1.
3. *(Transitivity).*
   (a) *(Heaps, Threads).* By Definition 5.1[4,1] and Lemma 4.4[3].
   (b) *(Heaps, Pools of Threads).* By Definition 5.1[5,2] and Lemma 5.2[3a].
4. *(Extension of Types of Heaps and μ).*
   (a) *(Threads).* By Definition 5.1[1] and Lemma 4.4[10a,9].
   (b) *(Pools of Threads).* By Definition 5.1[2] and Lemma 5.2[4a].
5. *(Equality of Int/Bool Typed "Low" Sub-Heaps).* By Definition 5.1[3], Definition 4.3[3] and Lemma 4.4[11c].

□

The following single-step strong bisimulation lemma states: given a pair of bisimilar configurations, if the first configuration steps, then the second configuration also steps, and bisimilarity is preserved in the resulting configurations. For the most part the lemma follows directly from Lemma 4.8 (λ^{sync}_{seq}: Strong Bisimulation, 1-step); the only exception being the isomorphism of the traces in case 3 below, which is a direct consequence of low scheduler behavior.

Lemma 5.3 ($\lambda^{sync}_{conc_{\mathcal{D}}}$: Strong Bisimulation, 1-step)**.**

1. *(Threads). If* $(\mathrm{H}_1, \zeta_1) \longrightarrow_{\Phi} (\mathrm{H}'_1, \zeta'_1)$ *and* $(\mathcal{H}_1, \mathrm{H}_1, \zeta_1) \sim^{\emptyset\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2, \zeta_2)$ *then for some* H'_2 *and* ζ'_2, $(\mathrm{H}_2, \zeta_2) \longrightarrow_{\Phi} (\mathrm{H}'_2, \zeta'_2)$ *holds, and there exists a* $\mathcal{H}'_1$, $\mathcal{H}'_2$ *and* μ' *such that* $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$, $\mu \subseteq \mu'$ *and* $(\mathcal{H}'_1, \mathrm{H}'_1, \zeta'_1) \sim^{\emptyset\tau}_{\Phi\ell\mu'} (\mathcal{H}'_2, \mathrm{H}'_2, \zeta'_2)$.
2. *(Pools of Threads with Next Schedule). If* $\left(\mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \longrightarrow_{\Phi i} \left(\mathrm{H}'_1, \mathrm{T}'_1\right)_{\mathtt{t}'_1}$ *and* $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$ *then for some* H'_2, T'_2 *and* $\mathtt{t}'_2$, $\left(\mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2} \longrightarrow_{\Phi i} \left(\mathrm{H}'_2, \mathrm{T}'_2\right)_{\mathtt{t}'_2}$ *holds, and there exists a* $\mathcal{H}'_1$, $\mathcal{H}'_2$ *and* μ' *such that* $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$, $\mu \subseteq \mu'$ *and* $\left(\mathcal{H}'_1, \mathrm{H}'_1, \mathrm{T}'_1\right)_{\mathtt{t}'_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \left(\mathcal{H}'_2, \mathrm{H}'_2, \mathrm{T}'_2\right)_{\mathtt{t}'_2}$.
3. *(Pools of Threads with Deterministic Scheduler). If* $\left(\mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \longrightarrow_{\Phi\mathbb{L}\mathcal{D}} \left(\mathrm{H}'_1, \mathrm{T}'_1\right)_{\mathtt{t}'_1}$ *and* $\left(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} \left(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$ *then for some* H'_2, T'_2 *and* $\mathtt{t}'_2$, $\left(\mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2} \longrightarrow_{\Phi\mathbb{L}\mathcal{D}} \left(\mathrm{H}'_2, \mathrm{T}'_2\right)_{\mathtt{t}'_2}$ *holds, and there exists a* $\mathcal{H}'_1$, $\mathcal{H}'_2$ *and* μ' *such that* $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$, $\mu \subseteq \mu'$ *and* $\left(\mathcal{H}'_1, \mathrm{H}'_1, \mathrm{T}'_1\right)_{\mathtt{t}'_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \left(\mathcal{H}'_2, \mathrm{H}'_2, \mathrm{T}'_2\right)_{\mathtt{t}'_2}$.

Proof. 1. *(Threads).* By premise $(\mathrm{H}_1, \zeta_1) \longrightarrow_{\Phi} (\mathrm{H}'_1, \zeta'_1)$; as per THREAD for some e_1, d_1, e'_1 and d'_1, $\zeta_1 = (e_1, d_1)$, $\left(\mathrm{H}_1, e_1\right)_{d_1} \longrightarrow_{\Phi} \left(\mathrm{H}'_1, e'_1\right)_{d'_1}$ and $\zeta'_1 = (e'_1, d'_1)$.
By premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2)$ and $(\mathcal{H}_1, \zeta_1) \sim^{\emptyset\tau}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{T}_2)$ hold, and by Definition 5.1[1] for some e_2 and d_2, $\zeta_2 = (e_2, d_2)$ and $\left(\mathcal{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, e_2\right)_{d_2}$. Then by Definition 4.3[4] $\left(\mathcal{H}_1, \mathrm{H}_1, e_1\right)_{d_1} \sim^{\emptyset\tau}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{H}_2, e_2\right)_{d_2}$.
By Lemma 4.8 (λ^{sync}_{seq}: Strong Bisimulation, 1-step) for some H'_2, e'_2 and d'_2, $\left(\mathrm{H}_2, e_2\right)_{d_2} \longrightarrow_{\Phi} \left(\mathrm{H}'_2, e'_2\right)_{d'_2}$, and there exists a $\mathcal{H}'_1$, $\mathcal{H}'_2$ and μ', such that $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$, $\mu \subseteq \mu'$ and $\left(\mathcal{H}'_1, \mathrm{H}'_1, e'_1\right)_{d'_1} \sim^{\emptyset\tau}_{\Phi\ell\mu'} \left(\mathcal{H}'_2, \mathrm{H}'_2, e'_2\right)_{d'_2}$. By THREAD for $\zeta'_2 = (e'_2, d'_2)$, $(\mathrm{H}_2, \zeta_2) \longrightarrow_{\Phi} (\mathrm{H}'_2, \zeta'_2)$, the first requisite result. Then by Definition 4.3[4] and Definition 5.1[1,4] we get $(\mathcal{H}'_1, \mathrm{H}'_1, \zeta'_1) \sim^{\emptyset\tau}_{\Phi\ell\mu'} (\mathcal{H}'_2, \mathrm{H}'_2, \zeta'_2)$, the second requisite result.
2. *(Pools of Threads with Next Schedule).* By premise $\left(\mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \longrightarrow_{\Phi i} \left(\mathrm{H}'_1, \mathrm{T}'_1\right)_{\mathtt{t}'_1}$; as per THREAD-POOL_NEXT-SCHEDULE for some $\overline{\zeta_{1_k}}$ and ζ'_{1_i}, $\mathrm{T}_1 = \overline{\zeta_{1_k}}$, $(\mathrm{H}, \zeta_{1_i}) \longrightarrow_{\Phi} (\mathrm{H}'_1, \zeta'_{1_i})$, $\mathrm{T}'_1 = \mathrm{T}_1[i \mapsto \zeta'_{1_i}]$ and $\mathtt{t}'_1 = \mathtt{t}_1, i$.
By premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{H}_2)$ and $\left(\mathcal{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu} \left(\mathcal{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2}$ hold. Then by Definition 5.1[2] $\mathtt{t}_1 = \mathtt{t}_2$ and for some $\overline{\zeta_{2_k}}$ and $\overline{\tau_k}$, $\mathrm{T}_2 = \overline{\zeta_{2_k}}$, $\Upsilon = \overline{\tau_k}$ and $(\mathcal{H}_1, \zeta_{1_i}) \sim^{\emptyset\tau_i}_{\Phi\ell\mu} (\mathcal{H}_2, \zeta_{2_i})$. By Definition 5.1[4] $(\mathcal{H}_1, \mathrm{H}_1, \zeta_{1_i}) \sim^{\emptyset\tau_i}_{\Phi\ell\mu} (\mathcal{H}_2, \mathrm{H}_2, \zeta_{2_i})$.
By Lemma 5.3[1] for some H'_2 and ζ'_{2_i}, $(\mathrm{H}_2, \zeta_{2_i}) \longrightarrow_{\Phi} (\mathrm{H}'_2, \zeta'_{2_i})$, and there exists a $\mathcal{H}'_1$, $\mathcal{H}'_2$ and μ' such that $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$, $\mu \subseteq \mu'$ and $(\mathcal{H}'_1, \mathrm{H}'_1, \zeta'_{1_i}) \sim^{\emptyset\tau_i}_{\Phi\ell\mu'} (\mathcal{H}'_2, \mathrm{H}'_2, \zeta'_{2_i})$. By THREAD-POOL_NEXT-SCHEDULE for $\mathrm{T}'_2 = \mathrm{T}_2[i \mapsto \zeta'_{2_i}]$ and $\mathtt{t}'_2 = \mathtt{t}_2, i$, $\left(\mathrm{H}_2, \mathrm{T}_2\right)_{\mathtt{t}_2} \longrightarrow_{\Phi i} \left(\mathrm{H}'_2, \mathrm{T}'_2\right)_{\mathtt{t}'_2}$ holds, the first requisite result. Further knowing $\mathtt{t}_1 = \mathtt{t}_2$ and $\mathtt{t}'_1 = \mathtt{t}_1, i$ implies $\mathtt{t}'_1 = \mathtt{t}'_2$. Finally by Definition 5.1[4,2,5] and Lemma 5.2[4] we get $\left(\mathcal{H}'_1, \mathrm{H}'_1, \mathrm{T}'_1\right)_{\mathtt{t}'_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \left(\mathcal{H}'_2, \mathrm{H}'_2, \mathrm{T}'_2\right)_{\mathtt{t}'_2}$, the second requisite result.
3. *(Pools of Threads with Deterministic Scheduler).* By premise $\left(\mathrm{H}_1, \mathrm{T}_1\right)_{\mathtt{t}_1} \longrightarrow_{\Phi\mathbb{L}\mathcal{D}} \left(\mathrm{H}'_1, \mathrm{T}'_1\right)_{\mathtt{t}'_1}$; as per THREAD-POOL_$\mathcal{D}$

for some $\overline{\zeta_{1_k}}$, $\hat{\mathrm{H}}_{low}$ and i, $\mathrm{T}_1 = \overline{\zeta_{1_k}}$, $\hat{\mathrm{H}}_{low} = \mathrm{H}_1\lceil\mathbb{L}$, $\mathcal{D}(k, \mathtt{t}_1, \hat{\mathrm{H}}_{low}) = i$ and $\big(\mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \longrightarrow_{\Phi i} \big(\mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'}$.
By premise $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{H}_2)$ and $\big(\mathcal{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu} \big(\mathcal{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2}$ hold. Then by Lemma 5.2[5] $\mathrm{H}_1\lceil\mathbb{L} = \mathrm{H}_2\lceil\mathbb{L} = \hat{\mathrm{H}}_{low}$, and by Definition 5.1[2] $\mathtt{t}_1 = \mathtt{t}_2$ and for some $\overline{\zeta_{2_k}}$, $\mathrm{T}_2 = \overline{\zeta_{2_k}}$. So $\mathcal{D}(k, \mathtt{t}_2, \hat{\mathrm{H}}_{low}) = \mathcal{D}(k, \mathtt{t}_1, \hat{\mathrm{H}}_{low}) = i$.
Now we know $\big(\mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \longrightarrow_{\Phi i} \big(\mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'}$ and $\big(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} \big(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2}$; by Lemma 5.3[2] for some H_2', T_2' and $\mathtt{t}_2'$, $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2} \longrightarrow_{\Phi i} \big(\mathrm{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$, and there exists a $\mathcal{H}_1'$, $\mathcal{H}_2'$ and μ' such that $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\mu \subseteq \mu'$ and $\big(\mathcal{H}_1', \mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \big(\mathcal{H}_2', \mathrm{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$, the second requisite result. Then by THREAD-POOL_$\mathcal{D}$ $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2} \longrightarrow_{\Phi\mathbb{L}\mathcal{D}} \big(\mathrm{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$ holds, the first requisite result.

□

The following n-step strong bisimulation lemma then states: given a pair of bisimilar configurations, if the first pool of threads takes n steps with a certain sequence of interleavings, then the second thread pool also exhibits the same external and internal timing behavior, that is, it also takes n steps, and with the same sequence of interleavings, respectively, as the first one; further bisimilarity is preserved in the resulting configurations.

Lemma 5.4 ($\lambda^{sync}_{conc_{\mathcal{D}}}$: Strong Bisimulation, n-steps)**.** *If* $\big(\mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \longrightarrow^n_{\Phi\mathbb{L}\mathcal{D}} \big(\mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'}$ *and* $\big(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} \big(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2}$ *then for some* H_2', T_2' *and* $\mathtt{t}_2'$, $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2} \longrightarrow^n_{\Phi\mathbb{L}\mathcal{D}} \big(\mathrm{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$ *holds, and there exists a* $\mathcal{H}_1'$, $\mathcal{H}_2'$ *and* μ' *such that* $\mathcal{H}_1 \subseteq \mathcal{H}_1'$, $\mathcal{H}_2 \subseteq \mathcal{H}_2'$, $\mu \subseteq \mu'$ *and* $\big(\mathcal{H}_1', \mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \big(\mathcal{H}_2', \mathrm{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$.

Proof. By induction on the derivation of $\big(\mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \longrightarrow^n_{\Phi\mathbb{L}\mathcal{D}} \big(\mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'}$ given Lemma 5.3[3].

□

Definition 5.5 (Convergence under a Deterministic Scheduler)**.** *A thread* i *in a configuration* C*, given a table of sync timer functions* Φ*, a set of heap locations* $\mathbb{L}$*, and a deterministic scheduler* $\mathcal{D}$*, converges to a value* v *in* n *steps iff* $\mathrm{C} \longrightarrow^n_{\Phi\mathbb{L}\mathcal{D}} \big(\mathrm{H}, \mathrm{T}\big)_{\mathtt{t}}$ *for some* H*,* T *and* $\mathtt{t}$*, and* $\mathrm{T}[i] = (v, d)$*, for some* d*.*

The following lemma, entailed by the strong bisimulation lemma, then formalizes the property of strong noninterference exhibited by $\lambda^{sync}_{conc_{\mathcal{D}}}$. It states: if a thread in one run of a concurrent program converges to a low integral or boolean value in a certain number of steps, then the same thread in a second run of that concurrent program, but possibly differing in high values from the first run, also converges to the same value and in the same number of steps as the first one.

We define the type judgement $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} (\mathrm{H}, \mathrm{T}) : \Upsilon$, where $\mathrm{T} = \overline{\zeta_k}$ and $\Upsilon = \overline{\tau_k}$, to hold iff $pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \mathrm{H}$ and $\forall 1 \leq i \leq k.\ pc, \Gamma, \mathcal{H} \vdash_{\Phi\ell} \zeta_i : \tau_i$. The indexing operation on a type pool is defined as on a thread pool. The multi-substitution $\zeta[\overline{v_k/x_k}]$, where $\zeta = (e, d)$, is shorthand for ζ' such that $\zeta' = \big(e[\overline{v_k/x_k}], d\big)$, and the multi-substitution $\mathrm{T}[\overline{v_k/x_k}]$, where $\mathrm{T} = \overline{\zeta_g}$, is shorthand for T' such that $\mathrm{T}' = \overline{\zeta_g'}$ and $\forall 1 \leq i \leq g.\ \zeta_i' = \zeta_i[\overline{v_k/x_k}]$; while $(\mathrm{H}, \mathrm{T})[\overline{v_k/x_k}]$ abbreviates $\big(\mathrm{H}[\overline{v_k/x_k}], \mathrm{T}[\overline{v_k/x_k}]\big)$.

Theorem 5.6 ($\lambda^{sync}_{conc_{\mathcal{D}}}$: Strong Noninterference)**.** *If* $\bot, \{\overline{x_k \mapsto \tau_k}\}, \mathcal{H} \vdash_{\Phi\ell} (\mathrm{H}, \mathrm{T}) : \Upsilon$, $\Upsilon[i] = \hat{\tau}$, $\mathcal{H}\lceil\mathbb{L} = \hat{\mathcal{H}}$,

δ ::= $\overline{n}$ *discrete distribution of thread indices*
$\mathcal{N}$: $\mathbb{N} \times \{\overline{\mathtt{t}}\} \times \{\overline{\hat{\mathrm{H}}}\} \to \{\overline{\delta}\}$ *nondeterministic scheduler*

Figure 13. $\lambda^{sync}_{conc_{\mathcal{N}}}$: Syntax Grammar

$$\text{THREAD-POOL\_}\mathcal{N}\ \frac{|\mathrm{T}| = k \qquad \hat{\mathrm{H}}_{low} = \mathrm{H}\lceil\mathbb{L} \qquad \mathcal{N}(k, \mathtt{t}, \hat{\mathrm{H}}_{low}) = \delta \qquad i \in \delta \qquad \big(\mathrm{H}, \mathrm{T}\big)_{\mathtt{t}} \longrightarrow_{\Phi i} \big(\mathrm{H}', \mathrm{T}'\big)_{\mathtt{t}'}}{\big(\mathrm{H}, \mathrm{T}\big)_{\mathtt{t}} \longrightarrow_{\Phi\mathbb{L}\mathcal{N}} \big(\mathrm{H}', \mathrm{T}'\big)_{\mathtt{t}'}}$$

Figure 14. $\lambda^{sync}_{conc_{\mathcal{N}}}$: Syntax-Directed Small-Step Operational Semantics

seclevel$(\hat{\tau})$, *seclevel*$(\hat{\mathcal{H}}) \leq \ell$, $\bot, \emptyset, \emptyset \vdash^{high}_{\Phi\ell} \overline{v_k, v_k' : \tau_k}$, $(\mathrm{H}_1, \mathrm{T}_1) = (\mathrm{H}, \mathrm{T})[\overline{v_k/x_k}]$, $(\mathrm{H}_2, \mathrm{T}_2) = (\mathrm{H}, \mathrm{T})[\overline{v_k'/x_k}]$ *and the thread* i *in* $\big(\mathrm{H}_1, \mathrm{T}_1\big)_{\epsilon}$ *given* Φ*,* $\mathbb{L}$ *and a deterministic scheduler* $\mathcal{D}$ *converges to a value* $\hat{v}$ *in* n *steps, then the thread* i *in* $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon}$ *given* Φ*,* $\mathbb{L}$ *and* $\mathcal{D}$ *converges to the same value* $\hat{v}$ *in* n *steps.*

Proof. By premise for some H_1', T_1', $\mathtt{t}_1'$ and d_1', $\big(\mathrm{H}_1, \mathrm{T}_1\big)_{\epsilon} \longrightarrow^n_{\Phi\mathbb{L}\mathcal{D}} \big(\mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'}$ and $\mathrm{T}_1'[i] = (\hat{v}, d_1')$. Let $\mathbb{L} = \{\overline{loc_k}\}$, for some $\overline{loc_k}$, and $\mu = \{\overline{loc_k \mapsto loc_k}\}$; then by Definition 5.1 and 4.3[1,3] $\big(\mathcal{H}, \mathrm{H}_1, \mathrm{T}_1\big)_{\epsilon} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} \big(\mathcal{H}, \mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon}$. By Lemma 5.4 ($\lambda^{sync}_{conc_{\mathcal{D}}}$: Strong Bisimulation, n-steps) for some H_2', T_2' and $\mathtt{t}_2'$, $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon} \longrightarrow^n_{\Phi\mathbb{L}\mathcal{D}} \big(\mathrm{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$ holds, and there exists a $\mathcal{H}_1'$, $\mathcal{H}_2'$ and μ' such that $\mathcal{H} \subseteq \mathcal{H}_1'$, $\mathcal{H} \subseteq \mathcal{H}_2'$, $\mu \subseteq \mu'$ and $\big(\mathcal{H}_1', \mathrm{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \big(\mathcal{H}_2', \mathrm{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$. By Definition 5.1[5] $\big(\mathcal{H}_1', \mathrm{T}_1'\big)_{\mathtt{t}_1'} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} \big(\mathcal{H}_2', \mathrm{T}_2'\big)_{\mathtt{t}_2'}$ holds. Then by Definition 5.1[2,1] and Definition 4.3[1], for some e_2' and d_2' such that $\mathrm{T}_2'[i] = (e_2', d_2')$, $\big(\mathcal{H}_1', \hat{v}\big)_{d_1'} \sim^{\emptyset\hat{\tau}}_{\Phi\ell\mu'} \big(\mathcal{H}_2', e_2'\big)_{d_2'}$ and $d_1' = d_2' = \infty$; by Lemma 4.4[11c] $e_2' = \hat{v}$. That is, the thread i in $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon}$ given Φ, $\mathbb{L}$ and $\mathcal{D}$ converges to the value $\hat{v}$ in n steps, the requisite result.

□

6. The $\lambda^{sync}_{conc_{\mathcal{N}}}$ Runtime System

We now extend $\lambda^{sync}_{conc_{\mathcal{D}}}$ to $\lambda^{sync}_{conc_{\mathcal{N}}}$, which incorporates nondeterminism into the scheduler. The language syntax for $\lambda^{sync}_{conc_{\mathcal{N}}}$ appears in Figure 13, and its small-step operational semantics in Figure 14, as addendums to Figures 11 and 12 respectively. The scheduler $\mathcal{N}$ denotes an arbitrary nondeterministic scheduler (possibly chosen by an attacker), and is parameterized over the same 3-tuple as the deterministic scheduler $\mathcal{D}$ in $\lambda^{sync}_{conc_{\mathcal{D}}}$; however, unlike $\mathcal{D}$, it returns a discrete distribution δ of indices into the thread pool denoting the probability of each thread to be scheduled next – an index can occur multiple times in the distribution δ, and the more times an index occurs, the greater the likelihood of the corresponding thread to be chosen. The operational semantics rule THREAD-POOL_$\mathcal{N}$ randomly picks an index from δ choosing the next scheduled thread; we write $i \in \delta$ iff $\delta = \ldots i \ldots$. The small-step reduction relation $\longrightarrow_{\Phi\mathbb{L}\mathcal{N}}$ and its n-step version $\longrightarrow^n_{\Phi\mathbb{L}\mathcal{N}}$ are defined analogous to the reduction relations $\longrightarrow_{\Phi\mathbb{L}\mathcal{D}}$ and $\longrightarrow^n_{\Phi\mathbb{L}\mathcal{D}}$, respectively, for $\lambda^{sync}_{conc_{\mathcal{D}}}$.

Our nondeterministic scheduler $\mathcal{N}$ is similar to the one in [38], but the latter assigns continuous (real-number) likelihoods to the threads being scheduled, while $\mathcal{N}$ employs discrete probabilities; our discrete probabilities can however be made arbitrarily precise as the distribution δ may be arbitrarily large. We take a discrete probability approach as it allows for a simpler formalism.

$$
\begin{array}{lcll}
\mathbf{C} & ::= & \mathrm{C} \mid \overline{\mathbf{C}} & \textit{discrete distribution of runtime configurations} \\
\mathbb{H} & ::= & \mathcal{H} \mid \overline{\mathbb{H}} & \textit{discrete distribution of types of heaps} \\
\mathbf{u} & ::= & \mu \mid \overline{\mathbf{u}} & \textit{discrete distribution of } \mu\textit{'s} \\
\hat{\Delta} & ::= & \overline{\hat{v}} & \textit{discrete distribution of integer/boolean values}
\end{array}
$$

Figure 15. $\lambda^{sync}_{conc_{\mathcal{N}}}$ (All-Paths-In-Parallel): Grammar *(for proof of probabilistic noninterference only)*

$$
\textsc{thread-pool}\_\mathcal{N}\_\textsc{all-paths}\quad
\frac{\begin{array}{c}\mathrm{C} = \big(\mathrm{H}, \mathrm{T}\big)_{\mathtt{t}} \\ \mathrm{T} = \overline{\zeta_k} \qquad \hat{\mathrm{H}}_{low} = \mathrm{H}\lceil\mathbb{L} \qquad \mathcal{N}\big(k, \mathtt{t}, \hat{\mathrm{H}}_{low}\big) = \delta \qquad \delta = i_1, \ldots, i_n \\ \forall i \in i_1, \ldots, i_n.\ \mathrm{C} \longrightarrow_{\Phi i} \mathrm{C}_i \qquad \mathbf{C} = \mathrm{C}_{i_1}, \ldots, \mathrm{C}_{i_n}\end{array}}{\mathrm{C} \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}}
$$

$$
\textsc{configuration-distribution}\quad
\frac{\mathbf{C} = \mathbf{C}_1, \ldots, \mathbf{C}_k \qquad \forall 1 \leq i \leq k.\ \mathbf{C}_i \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_i \qquad \mathbf{C}' = \mathbf{C}'_1, \ldots, \mathbf{C}'_k}{\mathbf{C} \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'}
$$

Figure 16. $\lambda^{sync}_{conc_{\mathcal{N}}}$ (All-Paths-In-Parallel): Operational Semantics *(for proof of probabilistic noninterference only)*

6.1 Security Properties of $\lambda^{sync}_{conc_{\mathcal{N}}}$

The discrete probability distribution permits us to prove the probabilistic noninterference result [38] for $\lambda^{sync}_{conc_{\mathcal{N}}}$ using a technique which aligns better with those used for λ^{sync}_{seq} and $\lambda^{sync}_{conc_{\mathcal{D}}}$. Our proof technique is as follows: execute in parallel all possible paths indicated by the nondeterministic scheduler, and show the final discrete distribution of low values and their timings are independent of high data. To this end we define an all-paths-in-parallel semantics, corresponding the semantics of $\lambda^{sync}_{conc_{\mathcal{N}}}$, in Figure 16, and the corresponding grammar appears in Figure 15. It captures, in parallel, all possible interleavings of a concurrent program under a nondeterministic scheduler. The reduction relation $\rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}}$, for a table of sync timer functions Φ, a set of low integral or boolean heap locations $\mathbb{L}$, and a nondeterministic scheduler $\mathcal{N}$, is defined over discrete distributions of configurations $\mathbf{C}$; the corresponding n-step reflexive and transitive closure is denoted as $\rightsquigarrow^{n}_{\Phi\mathbb{L}\mathcal{N}}$.

The following lemma states the soundness of the all-paths-in-parallel semantics relative to the operational semantics of $\lambda^{sync}_{conc_{\mathcal{N}}}$. We write $\mathrm{C} \in \mathbf{C}$ iff either, $\mathbf{C} = \mathrm{C}$, or for some $\overline{\mathbf{C}_k}$, $\mathbf{C} = \overline{\mathbf{C}_k}$ and $\bigvee_{1 \leq i \leq k} \mathrm{C} \in \mathbf{C}_i$.

Lemma 6.1 (Soundness of All-Paths-In-Parallel Semantics)**.** *If* $\mathrm{C} \longrightarrow^{n}_{\Phi\mathbb{L}\mathcal{N}} \mathrm{C}'$, $\mathrm{C} \in \mathbf{C}$ *and* $\mathbf{C} \rightsquigarrow^{n}_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'$ *then* $\mathrm{C}' \in \mathbf{C}'$.

Proof. By induction on the derivation of $\mathbf{C} \rightsquigarrow^{n}_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'$. □

$\lambda^{sync}_{conc_{\mathcal{N}}}$ is proved to be probabilistically noninterfering by showing that all-paths-in-parallel executions of two pools of threads, which differ only in high values, are strongly bisimilar. Strong bisimilarity implies isomorphism of the distributions of configurations at all intermediate steps of the all-paths-in-parallel execution, with the term *strong bisimilarity* indicating the lock-stepness of bisimilar executions as before. The bisimulation relation, defined below, inductively entails the bisimilarity of corresponding configurations; it is an addendum to $\lambda^{sync}_{conc_{\mathcal{D}}}$'s bisimulation relation (Definition 5.1).

Definition 6.2 ($\lambda^{sync}_{conc_{\mathcal{N}}}$: Bisimulation Relation)**.**

1. *(Configurations).* $(\mathcal{H}_1, \mathrm{C}_1) \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{C}_2)$ *iff* $\mathrm{C}_1 = \big(\mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1}$, $\mathrm{C}_2 = \big(\mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2}$ *and* $\big(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} \big(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2}$.
2. *(Distributions of Configurations).* $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$ *iff for some* $\overline{\mathbb{H}'_k}$, $\overline{\mathbb{H}''_k}$, $\overline{\mathbf{C}'_k}$, $\overline{\mathbf{C}''_k}$, *and* $\overline{\mathbf{u}_k}$, $\mathbb{H}_1 = \overline{\mathbb{H}'_k}$, $\mathbb{H}_2 = \overline{\mathbb{H}''_k}$, $\mathbf{C}_1 = \overline{\mathbf{C}'_k}$, $\mathbf{C}_2 = \overline{\mathbf{C}''_k}$, $\mathbf{u} = \overline{\mathbf{u}_k}$, *and* $\forall 1 \leq i \leq k.\ (\mathbb{H}'_i, \mathbf{C}'_i) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}_i\mathbb{L}} (\mathbb{H}''_i, \mathbf{C}''_i)$.

Note the above bisimulation relation is scheduler-independent, akin to the one in [38].

The inverse of a pool of μ's, $\mathbf{u}^{-1}$ is defined as, $\mathbf{u}^{-1} = \mu^{-1}$ if $\mathbf{u} = \mu$, and $\mathbf{u}^{-1} = \mathbf{u}_1^{-1}, \ldots, \mathbf{u}_k^{-1}$ if $\mathbf{u} = \overline{\mathbf{u}_k}$. Analogously, the composition of distributions of μ's is defined as, $\mathbf{u} \circ \mathbf{u}' = \mu \circ \mu'$ if $\mathbf{u} = \mu$ and $\mathbf{u}' = \mu'$, and $\mathbf{u} \circ \mathbf{u}' = \mathbf{u}_1 \circ \mathbf{u}'_1, \ldots, \mathbf{u}_k \circ \mathbf{u}'_k$ if $\mathbf{u} = \overline{\mathbf{u}_k}$ and $\mathbf{u}' = \overline{\mathbf{u}'_k}$.

Lemma 6.3 ($\lambda^{sync}_{conc_{\mathcal{N}}}$: Properties of Bisimulation Relation)**.**

1. *(Symmetry). If* $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$ *then* $(\mathbb{H}_2, \mathbf{C}_2) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}^{-1}\mathbb{L}} (\mathbb{H}_1, \mathbf{C}_1)$.
2. *(Reflexivity).*
   (a) *(Configurations). If* $\mathrm{C} = \big(\mathrm{H}, \mathrm{T}\big)_{\mathtt{t}}$ *and* $\big(\mathcal{H}, \mathrm{H}, \mathrm{T}\big)_{\mathtt{t}} \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} \big(\mathcal{H}, \mathrm{H}, \mathrm{T}\big)_{\mathtt{t}}$ *then* $(\mathcal{H}, \mathrm{C}) \sim^{\Gamma\Upsilon}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}, \mathrm{C})$.
   (b) *(Distributions of Configurations). If* $\mathbb{H} = \overline{\mathbb{H}_k}$, $\mathbf{C} = \overline{\mathbf{C}_k}$, $\mathbf{u} = \overline{\mathbf{u}_k}$, *and* $\forall i.\ 1 \leq i \leq k \implies (\mathbb{H}_i, \mathbf{C}_i) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}_i\mathbb{L}} (\mathbb{H}_i, \mathbf{C}_i)$, *then* $(\mathbb{H}, \mathbf{C}) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}, \mathbf{C})$.
3. *(Transitivity). If* $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$ *and* $(\mathbb{H}_2, \mathbf{C}_2) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}_3, \mathbf{C}_3)$ *then for* $\mathbf{u}'' = \mathbf{u}' \circ \mathbf{u}$, $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}''\mathbb{L}} (\mathbb{H}_3, \mathbf{C}_3)$.

Proof. 1. *(Symmetry)*. By Definition 6.2 given Lemma 5.2[1b].
2. *(Reflexivity)*. By Definition 6.2.
3. *(Transitivity)*. By Definition 6.2 and Lemma 5.2[3b].

□

The following single-step strong bisimulation lemma states: given a pair of bisimilar (distributions of) configurations, if the first one steps (all-paths-in-parallel), then the second one also steps (all-paths-in-parallel), and bisimilarity is preserved in the resulting (distributions of) configurations. For the most part the lemma follows directly from Lemma 5.3[2]; the only exception being the isomorphism of the traces in case 2 below, which is a direct consequence of low scheduler behavior.

We write $\mathbb{H}_1 \subseteq \mathbb{H}_2$ if either, for some $\mathcal{H}_1$ and $\mathcal{H}_2$, $\mathbb{H}_1 = \mathcal{H}_1$, $\mathbb{H}_2 = \mathcal{H}_2$ and $\mathcal{H}_1 \subseteq \mathcal{H}_2$, or for some $\overline{\mathbb{H}_{1_k}}$ and $\overline{\mathbb{H}_{2_k}}$, $\mathbb{H}_1 = \overline{\mathbb{H}_{1_k}}$, $\mathbb{H}_2 = \overline{\mathbb{H}_{2_k}}$ and $\overline{\mathbb{H}_{1_k} \subseteq \mathbb{H}_{2_k}}$. Analogously we write $\mathbf{u}_1 \subseteq \mathbf{u}_2$ if either, for some μ_1 and μ_2, $\mathbf{u}_1 = \mu_1$, $\mathbf{u}_2 = \mu_2$ and $\mu_1 \subseteq \mu_2$, or for some $\overline{\mathbf{u}_{1_k}}$ and $\overline{\mathbf{u}_{2_k}}$, $\mathbf{u}_1 = \overline{\mathbf{u}_{1_k}}$, $\mathbf{u}_2 = \overline{\mathbf{u}_{2_k}}$ and $\overline{\mathbf{u}_{1_k} \subseteq \mathbf{u}_{2_k}}$.

Lemma 6.4 ($\lambda^{sync}_{conc_{\mathcal{N}}}$: Strong Bisimulation, 1-step)**.**

1. *(Configurations). If* $\mathrm{C}_1 \longrightarrow_{\Phi i} \mathrm{C}'_1$ *and* $(\mathcal{H}_1, \mathrm{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{C}_2)$ *then for some* C'_2, $\mathrm{C}_2 \longrightarrow_{\Phi i} \mathrm{C}'_2$ *holds, and there exists a* $\mathcal{H}'_1$, $\mathcal{H}'_2$ *and* μ' *such that* $\mathcal{H}_1 \subseteq \mathcal{H}'_1$, $\mathcal{H}_2 \subseteq \mathcal{H}'_2$, $\mu \subseteq \mu'$ *and* $(\mathcal{H}'_1, \mathrm{C}'_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mu'\mathbb{L}} (\mathcal{H}'_2, \mathrm{C}'_2)$.
2. *(Distributions of Configurations). If* $\mathbf{C}_1 \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_1$ *and* $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$ *then for some* $\mathbf{C}'_2$, $\mathbf{C}_2 \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_2$ *holds, and there exists a* $\mathbb{H}'_1$, $\mathbb{H}'_2$ *and* $\mathbf{u}'$ *such that* $\mathbb{H}_1 \subseteq \mathbb{H}'_1$, $\mathbb{H}_2 \subseteq \mathbb{H}'_2$, $\mathbf{u} \subseteq \mathbf{u}'$ *and* $(\mathbb{H}'_1, \mathbf{C}'_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}'_2, \mathbf{C}'_2)$.

Proof. 1. *(Configurations)*. By Definition 6.2[1] and Lemma 5.3[2].
2. *(Distributions of Configurations)*. By induction on the derivation of $\mathbf{C}_1 \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_1$. Following are the possible semantics rule from Figure 16 applicable at the root of this derivation.
   (a) THREAD-POOL_$\mathcal{N}$_ALL-PATHS. For some C_1, H_1, T_1, $\mathtt{t}_1$, $\overline{\zeta_{1_k}}$, $\hat{\mathrm{H}}_{low}$, δ, $\overline{i_n}$ and $\mathrm{C}_{1_{i_1}}$, …, $\mathrm{C}_{1_{i_n}}$, $\mathbf{C}_1 = \mathrm{C}_1, \mathrm{C}_1 = \big(\mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1}$, $\mathrm{T}_1 = \overline{\zeta_{1_k}}$, $\hat{\mathrm{H}}_{low} = \mathrm{H}_1\lceil\mathbb{L}, \mathcal{N}(k, \mathtt{t}_1, \hat{\mathrm{H}}_{low}) = \delta$,

$\delta = i_1, \ldots, i_n, \forall i \in i_1, \ldots, i_n.\ \mathrm{C}_1 \longrightarrow_{\Phi i} \mathrm{C}_{1_i}$ and $\mathbf{C}'_1 = \mathrm{C}_{1_{i_1}}, \ldots, \mathrm{C}_{1_{i_n}}$.

By premise we know $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$; then knowing $\mathbf{C}_1 = \mathrm{C}_1$ by Definition 6.2[2] for some C_2, $\mathcal{H}_1$, $\mathcal{H}_2$ and μ, $\mathbf{C}_2 = \mathrm{C}_2$, $\mathbb{H}_1 = \mathcal{H}_1$, $\mathbb{H}_2 = \mathcal{H}_2$, $\mathbf{u} = \mu$ and $(\mathcal{H}_1, \mathrm{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{C}_2)$. We know for all $i \in i_1, \ldots, i_n$, $\mathrm{C}_1 \longrightarrow_{\Phi i} \mathrm{C}_{1_i}$; then by applying Lemma 6.4[1] to each of the aforementioned reductions we get, for all $i \in i_1, \ldots, i_n$, there exists a C_{2_i} such that $\mathrm{C}_2 \longrightarrow_{\Phi i} \mathrm{C}_{2_i}$ and there exists a $\mathcal{H}_{1_i}$, $\mathcal{H}_{2_i}$ and μ_i such that $\mathcal{H}_1 \subseteq \mathcal{H}_{1_i}$, $\mathcal{H}_2 \subseteq \mathcal{H}_{2_i}$, $\mu \subseteq \mu_i$ and $(\mathcal{H}_{1_i}, \mathrm{C}_{1_i}) \sim^{\emptyset\Upsilon}_{\Phi\ell\mu_i\mathbb{L}} (\mathcal{H}_{2_i}, \mathrm{C}_{2_i})$. Letting $\mathbb{H}'_1 = \mathcal{H}_{1_{i_1}}, \ldots, \mathcal{H}_{1_{i_n}}$, $\mathbb{H}'_2 = \mathcal{H}_{2_{i_1}}, \ldots, \mathcal{H}_{2_{i_n}}$ and $\mathbf{u}' = \mu_{i_1}, \ldots, \mu_{i_n}$, by Definition 6.2[2] we get $(\mathbb{H}'_1, \mathrm{C}_{1_{i_1}}, \ldots, \mathrm{C}_{1_{i_n}}) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}'_2, \mathrm{C}_{2_{i_1}}, \ldots, \mathrm{C}_{2_{i_n}})$.

By premise we know $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$, that is, $(\mathcal{H}_1, \mathrm{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{C}_2)$. We also know $\mathrm{C}_1 = \big(\mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1}$. Then by Definition 6.2[1] for some H_2, T_2 and $\mathtt{t}_2$, $\mathrm{C}_2 = \big(\mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2}$ and $\big(\mathcal{H}_1, \mathrm{H}_1, \mathrm{T}_1\big)_{\mathtt{t}_1} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} \big(\mathcal{H}_2, \mathrm{H}_2, \mathrm{T}_2\big)_{\mathtt{t}_2}$; by Definition 5.1[5,2] we get $(\mathcal{H}_1, \mathrm{H}_1) \sim^{\emptyset}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}_2, \mathrm{H}_2)$, $\mathtt{t}_1 = \mathtt{t}_2$ and for some $\overline{\zeta_{2_k}}$, $\mathrm{T}_2 = \overline{\zeta_{2_k}}$. By Lemma 5.2[5] $\mathrm{H}_1\lceil\mathbb{L} = \mathrm{H}_2\lceil\mathbb{L} = \hat{\mathrm{H}}_{low}$. Hence $\mathcal{N}(k, \mathtt{t}_2, \hat{\mathrm{H}}_{low}) = \mathcal{N}(k, \mathtt{t}_1, \hat{\mathrm{H}}_{low}) = \delta$. Then by THREAD-POOL_$\mathcal{N}$_ALL-PATHS for $\mathbf{C}'_2 = \mathrm{C}_{2_{i_1}}, \ldots, \mathrm{C}_{2_{i_n}}$, $\mathrm{C}_2 \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_2$, that is, $\mathbf{C}_2 \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_2$ holds, the first requisite result.

We know $(\mathbb{H}'_1, \mathrm{C}_{1_{i_1}}, \ldots, \mathrm{C}_{1_{i_n}}) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}'_2, \mathrm{C}_{2_{i_1}}, \ldots, \mathrm{C}_{2_{i_n}})$, that is, $(\mathbb{H}'_1, \mathbf{C}'_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}'_2, \mathbf{C}'_2)$, the second requisite result.

(b) CONFIGURATION-DISTRIBUTION. For some $\overline{\mathbf{C}_{1_k}}$, and $\overline{\mathbf{C}'_{1_k}}$, $\mathbf{C}_1 = \mathbf{C}_{1_1}, \ldots, \mathbf{C}_{1_k}, \forall 1 \leq i \leq k.\ \mathbf{C}_{1_i} \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_{1_i}$, and $\mathbf{C}'_1 = \mathbf{C}'_{1_1}, \ldots, \mathbf{C}'_{1_k}$.

By premise $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$; then by Definition 6.2[2] for some $\overline{\mathbf{C}_{2_k}}$, $\overline{\mathbb{H}_{1_k}}$, $\overline{\mathbb{H}_{2_k}}$, and $\overline{\mathbf{u}_k}$, $\mathbf{C}_2 = \overline{\mathbf{C}_{2_k}}$, $\mathbb{H}_1 = \overline{\mathbb{H}_{1_k}}$, $\mathbb{H}_2 = \overline{\mathbb{H}_{2_k}}$, $\mathbf{u} = \overline{\mathbf{u}_k}$, and for all i such that $1 \leq i \leq k$, $(\mathbb{H}_{1_i}, \mathbf{C}_{1_i}) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}_i\mathbb{L}} (\mathbb{H}_{2_i}, \mathbf{C}_{2_i})$ holds.

By induction hypothesis for all i such that $1 \leq i \leq k$, for some $\mathbf{C}'_{2_i}$, $\mathbf{C}_{2_i} \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_{2_i}$, and there exists a $\mathbb{H}'_{1_i}$, $\mathbb{H}'_{2_i}$ and $\mathbf{u}'_i$ such that $\mathbb{H}_{1_i} \subseteq \mathbb{H}'_{1_i}$, $\mathbb{H}_{2_i} \subseteq \mathbb{H}'_{2_i}$, $\mathbf{u}_i \subseteq \mathbf{u}'_i$ and $(\mathbb{H}'_{1_i}, \mathbf{C}'_{1_i}) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'_i\mathbb{L}} (\mathbb{H}'_{2_i}, \mathbf{C}'_{2_i})$.

By CONFIGURATION-DISTRIBUTION, letting $\mathbf{C}'_2 = \mathbf{C}'_{2_1}, \ldots, \mathbf{C}'_{2_k}$, $\mathbf{C}_2 \rightsquigarrow_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_2$, the first requisite result. Then by Definition 6.2[2], for $\mathbb{H}'_1 = \mathbb{H}'_{1_1}, \ldots, \mathbb{H}'_{1_k}$, $\mathbb{H}'_2 = \mathbb{H}'_{2_1}, \ldots, \mathbb{H}'_{2_k}$ and $\mathbf{u}' = \mathbf{u}'_1, \ldots, \mathbf{u}'_k$, $(\mathbb{H}'_1, \mathbf{C}'_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}'_2, \mathbf{C}'_2)$ holds, the second requisite result.

□

The following n-step strong bisimulation lemma then states: given a pair of bisimilar (distributions of) configurations, if the first one takes n steps (all-paths-in-parallel), then the second one also takes n steps (all-paths-in-parallel), and bisimilarity is preserved in the resulting (distributions of) configurations.

Lemma 6.5 ($\lambda^{sync}_{conc_{\mathcal{N}}}$: Strong Bisimulation, n-steps)**.** *If* $\mathbf{C}_1 \rightsquigarrow^n_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_1$ *and* $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$ *then for some* $\mathbf{C}'_2$, $\mathbf{C}_2 \rightsquigarrow^n_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_2$ *holds, and there exists a* $\mathbb{H}'_1$, $\mathbb{H}'_2$ *and* $\mathbf{u}'$ *such that* $\mathbb{H}_1 \subseteq \mathbb{H}'_1$, $\mathbb{H}_2 \subseteq \mathbb{H}'_2$, $\mathbf{u} \subseteq \mathbf{u}'$ *and* $(\mathbb{H}'_1, \mathbf{C}'_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}'_2, \mathbf{C}'_2)$.

Proof. By induction on the derivation of $\mathbf{C}_1 \rightsquigarrow^n_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_1$ given Lemma 6.4[2]. □

The concatenation operator @ on $\hat{\Delta}$'s is defined as, $\hat{\Delta}_1@\hat{\Delta}_2 = \hat{\Delta}_3$ iff $\hat{\Delta}_1 = \overline{\hat{v}_j}$, $\hat{\Delta}_2 = \overline{\hat{v}'_k}$ and $\hat{\Delta}_3 = \hat{v}_1, \ldots, \hat{v}_j, \hat{v}'_1, \ldots, \hat{v}'_k$.

Definition 6.6 (Ground to Discrete Distribution of Values)**.**

1. *(Configuration).* $\lfloor \mathrm{C} \rfloor_i = \hat{v}$ *if* $\mathrm{C} = \big(\mathrm{H}, \mathrm{T}\big)_{\mathtt{t}}$ *and* $\mathrm{T}[i] = (\hat{v}, d)$*, for some* H, t *and* d*; otherwise* $\lfloor \mathrm{C} \rfloor_i = \cdot$.
2. *(Distribution of Configurations).* $\lfloor \mathbf{C} \rfloor_i = \hat{\Delta}$ *iff* $\mathbf{C} = \overline{\mathbf{C}_k}$, $\forall 1 \leq j \leq k.\ \lfloor \mathbf{C}_j \rfloor_i = \hat{\Delta}_j$*, and* $\hat{\Delta} = \hat{\Delta}_1@\ldots@\hat{\Delta}_k$.

Lemma 6.7 ("Low" Int/Bool Type)**.** *If* $(\mathbb{H}_1, \mathbf{C}_1) \sim^{\Gamma\Upsilon}_{\Phi\ell\mathbf{u}\mathbb{L}} (\mathbb{H}_2, \mathbf{C}_2)$, $\Upsilon[i] = \hat{\tau}$ *seclevel*$(\hat{\tau}) \leq \ell$ *and* $\lfloor \mathbf{C}_1 \rfloor_i = \hat{\Delta}$ *then* $\lfloor \mathbf{C}_2 \rfloor_i = \hat{\Delta}$.

Proof. By Definition 6.2, Definition 5.1, Definition 4.3[1], Lemma 4.4[11c] and Definition 6.6. □

The following defines the convergence of a thread to a distribution of values under a nondeterministic scheduler.

Definition 6.8 (Convergence under a Nondeterministic Scheduler)**.** *A thread* i *in a configuration* C*, given a table of sync timer functions* Φ*, a set of heap locations* $\mathbb{L}$ *and a nondeterministic scheduler* $\mathcal{N}$*, converges to a distribution of values* $\hat{\Delta}$ *in* n *steps iff* $\mathrm{C} \rightsquigarrow^n_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}$*, for some* $\mathbf{C}$*, and* $\lfloor \mathbf{C} \rfloor_i = \hat{\Delta}$.

The strong bisimulation lemma then entails the following property of probabilistic noninterference for $\lambda^{sync}_{conc_{\mathcal{N}}}$. It states: if a thread in a concurrent configuration converges to a certain distribution of low integral or boolean values after a certain number of steps, then the same thread in that concurrent configuration, but possibly differing in high values from the first one, converges to the same distribution of values after the same number of steps as the first one.

Theorem 6.9 ($\lambda^{sync}_{conc_{\mathcal{N}}}$: Probabilistic Noninterference)**.** *If* $\bot, \{\overline{x_k \mapsto \tau_k}\}, \mathcal{H} \vdash_{\Phi\ell} (\mathrm{H}, \mathrm{T}) : \Upsilon$, $\Upsilon[i] = \hat{\tau}$, $\mathcal{H}\lceil\mathbb{L} = \hat{\mathcal{H}}$*, seclevel*$(\hat{\tau})$*, seclevel*$(\hat{\mathcal{H}}) \leq \ell$*,* $\bot, \emptyset, \emptyset \vdash^{high}_{\Phi\ell} \overline{v_k, v'_k : \tau_k}$*,* $(\mathrm{H}_1, \mathrm{T}_1) = (\mathrm{H}, \mathrm{T})[\overline{v_k/x_k}]$, $(\mathrm{H}_2, \mathrm{T}_2) = (\mathrm{H}, \mathrm{T})[\overline{v'_k/x_k}]$ *and the thread* i *in* $\big(\mathrm{H}_1, \mathrm{T}_1\big)_{\epsilon}$ *given* Φ*,* $\mathbb{L}$ *and a nondeterministic scheduler* $\mathcal{N}$ *converges to a distribution of values* $\hat{\Delta}$ *in* n *steps, then the thread* i *in* $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon}$ *given* Φ*,* $\mathbb{L}$ *and* $\mathcal{N}$ *converges to the same distribution of values* $\hat{\Delta}$ *in* n *steps.*

Proof. By premise for $\mathrm{C}_1 = \big(\mathrm{H}_1, \mathrm{T}_1\big)_{\epsilon}$ and some $\mathbf{C}'_1$, $\mathrm{C}_1 \rightsquigarrow^n_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_1$ and $\lfloor \mathbf{C}'_1 \rfloor_i = \hat{\Delta}$. Let $\mathbb{L} = \{\overline{loc_k}\}$, for some $\overline{loc_k}$, and $\mu = \{\overline{loc_k \mapsto loc_k}\}$; by Definition 5.1 and 4.3[1,3] $\big(\mathcal{H}, \mathrm{H}_1, \mathrm{T}_1\big)_{\epsilon} \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} \big(\mathcal{H}, \mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon}$. Then by Definition 6.2[1] for $\mathrm{C}_2 = \big(\mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon}$, $(\mathcal{H}, \mathrm{C}_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mu\mathbb{L}} (\mathcal{H}, \mathrm{C}_2)$. By Lemma 6.5 ($\lambda^{sync}_{conc_{\mathcal{N}}}$: Strong Bisimulation, n-steps) for some $\mathbf{C}'_2$, $\mathrm{C}_2 \rightsquigarrow^n_{\Phi\mathbb{L}\mathcal{N}} \mathbf{C}'_2$ holds, and there exists a $\mathbb{H}'_1$, $\mathbb{H}'_2$ and $\mathbf{u}'$ such that $\mathcal{H}_1 \subseteq \mathbb{H}'_1$, $\mathcal{H}_2 \subseteq \mathbb{H}'_2$, $\mu \subseteq \mathbf{u}'$ and $(\mathbb{H}'_1, \mathbf{C}'_1) \sim^{\emptyset\Upsilon}_{\Phi\ell\mathbf{u}'\mathbb{L}} (\mathbb{H}'_2, \mathbf{C}'_2)$. Then by Lemma 6.7 $\lfloor \mathbf{C}'_2 \rfloor_i = \hat{\Delta}$. That is, the thread i in $\big(\mathrm{H}_2, \mathrm{T}_2\big)_{\epsilon}$ given Φ, $\mathbb{L}$ and $\mathcal{N}$ converges to the distribution of values $\hat{\Delta}$ in n steps, the requisite result.

□

Note the above result holds for all nondeterministic schedulers $\mathcal{N}$, that is, it is scheduler-independent [38].

7. Related Work

Since the seminal work of Volpano and Smith [44] much work has been done to secure timing channels in programming languages. However, all the efforts focus on static analyses; to our knowledge there is no language-based runtime approach to secure timing channels. Further, most existing works [43, 38, 47, 42, 34, 3] concentrate on closing the internal timing channels, leaving the external

ones vulnerable to attacks. Our dynamic synchronization technique secures both external and internal timing channels.

The program transformation approach of Agat [1] is the only known language-based technique to secure both external and internal timing channels. It equalizes the execution times of high branches by a program transformation which pads each branch with instructions that pretend to execute the other branch as well. Various extensions to Agat's approach have been proposed – [38] employed a minor variant to secure internal timing channels in the context of multi-threaded programs (using skip instructions and dummy forks instead of the "skip-commands", `skipAsn` and `skipIf`, used by Agat), [10] augmented Agat's work towards a bytecode-like language, while [46] adapted it to secure typed assembly languages [27]. Recently a variant based on unification, which generates programs smaller in size, has been proposed [19].

The aforementioned works, however, have limited expressiveness in that they do not appear to generalize to incorporate higher-order functions – the exact target code of a branching statement is assumed to be known statically in these works, whereas higher-order function invocations are dynamic in that the exact code to be invoked is only known at runtime. Furthermore, none except [10] allow looping executions on high guards, that is, executions whose running times may depend on secret data – the execution times of such recursive computations are inherently tied to secret information (the high terminating condition) and it does not seem possible to secure them via static approaches without imposing severe restrictions such as the disallowance of subsequent low outputs [10]. We support both fully higher-order function invocations as well as recursive computations with high terminating conditions.

On the other hand, our dynamic synchronization technique endures an overhead of computing the sync times at runtime, and then handling the setting and expiration of the synchronization timers. In practice a hybrid approach may be best: use Agat's technique wherever applicable, and use our dynamic synchronization technique to secure cases the former cannot handle.

A recent work by Barthe *et al*. [4] proposed a variant of Agat's cross-copying technique based on a transaction mechanism. Each branch is wrapped in a transaction and then sequentially composed with the other corresponding branch. The transaction around the original branch is committed, while the cross-copied transaction is aborted. This technique is applicable to object-oriented languages with exceptions and methods calls. However it suffers from the same limitations as Agat's cross-copying approach, in that neither high-recursive programs nor higher-order function applications (dynamic dispatch in their object-oriented context) are supported. Additionally, unlike Agat's, this work suffers from a runtime overhead of committing and aborting transactions. Neither it is clear if it is applicable in a concurrent setting.

Molnar *et al*. [26] proposed methods to secure C programs against side-channel attacks. It performs a C-to-C program transformation to ensure both branches of conditional branching statements are executed but only results from the branch that would have been executed in the original program are retained through logical masking. The approach has been formalized for a simple imperative language called IncredibL, with only bounded loops (loops that can be fully unrolled statically), if statements and straight-line assignments. It is unclear if the proposed techniques would generalize to a richer language.

Neilson and Schwartzbach [28] presented a domain-specific imperative programming language called SMCL for Secure Multiparty Computation (SMC). To prevent timing leaks its runtime semantics evaluates both branches of if statements with a secret conditionals in sequence and on copies of the local state. After these executions, the results of both branches are merged such that only those of the valid branch are retained. It does not allows loops on secret conditionals nor calls to recursive functions guarded by high conditions.

Kobayashi and Shirane [15] proposed a Java virtual machine language based type system for timing-sensitive noninterference. The system relies on the computation of control dependency for identifying sensitive regions, and closes timing channels in sequential code by inserting a delay linear with respect to the normal execution time. It does not support objects or subroutines.

On a different note, Siveroni *et al*.[40, 31] apply a padding technique, similar to but more efficient than Agat's, to transform two processes to have indistinguishable timing behavior. The work is not programming language specific, but based probabilistic transition systems (PTS) [12]. In principle it is adaptable to any probabilistic language whose semantics can be expressed in terms of a PTS model, as for example our $\lambda^{sync}_{conc_{\mathcal{N}}}$ language.

8. Towards a Realistic System

Language Expressiveness The λ^{sync}_{seq} language does not support exceptions or interactive inputs and outputs. Existing works [7, 4] have studied exceptions in the context of timing channels and have shown disallowing exceptions thrown in high contexts from escaping to low contexts prevents timing leaks. λ^{sync}_{seq} can be extended to incorporate exceptions using the same restriction – the synchronization construct delineates the enclosed high context from the enclosing low context, so exceptions must be disallowed from escaping the synchronization construct; exceptions not handled inside the synchronization context must be ignored. Interactive inputs and outputs can be incorporated using a similar philosophy; low inputs and outputs must both be disallowed in high contexts – requests for low inputs in high contexts must be fulfilled with dummy values instead, and low outputs in high contexts must be omitted, as in [21].

The $\lambda^{sync}_{conc_{\mathcal{D}}}$ and $\lambda^{sync}_{conc_{\mathcal{N}}}$ languages do not support dynamic thread creation. Existing works [38, 34] have studied dynamic thread creation in the context of timing channels – [38] proposed cross-copying the fork instructions in high branching statements as dummy forks, along the lines of Agat's technique, to secure timing channels; while [34] suggested partitioning the thread pool into low and high parts, such that threads created in high contexts are "hidden" in the high part, while the low part contains threads generated in low contexts; the timings of the high threads are hidden, via the scheduler, from those of the low threads. We believe $\lambda^{sync}_{conc_{\mathcal{D}}}$ and $\lambda^{sync}_{conc_{\mathcal{N}}}$ can be extended to incorporate forking following the thread pool partitioning technique [34].

Sabelfeld [35] studied the effects of synchronization primitives such as semaphores on timing channels, and concluded all synchronizations on high semaphores or in high contexts are susceptible to information leaks; $\lambda^{sync}_{conc_{\mathcal{D}}}$ and $\lambda^{sync}_{conc_{\mathcal{N}}}$ can be extended to incorporate synchronization along the same lines. Further we believe our techniques are adaptable to distributed programs as well along the lines of [23, 36].

Our formal model does not support declassification of secure data, a desirable feature in practice. One interesting aspect of our runtime technique is the parameterization of the sync times on low values; when coupled with declassification this parameterization can provide a mechanism to declassify *just* the timing channels. For example, if one or more parameters to a sync timer function (the $\overline{i_k}$ of IF-SYNC and APP-SYNC rules in Figure 8) is declassified, only the associated timing channel has in effect been allowed to disseminate high information, and if there are no attackers observing that timing channel the high data remains secret.

Implementation of Sync Timers Interval timers provided by modern operating systems can be used to realize our dynamic synchronization technique. On Unix the `setitimer` function, included

in <sys/time.h>, provides a mechanism for a process to interrupt itself in the future by setting a timer; the process receives a signal when the timer expires. ITIMER_VIRTUAL is an interval timer which counts the actual execution time of a process (its virtual time), and sends a sigvtalrm signal to the process when it expires. The virtual timer can be used to directly realize our dynamic synchronization technique: a compiler or a preprocessor can instrument the source program such that just before executing the code to be synchronized, ITIMER_VIRTUAL is set to the sync time indicated by the associated sync timer function. The sigvtalrm handler can then realize the SYNC-DONE and SYNC-ABORT rules of Figure 8. To be more precise, consider the following λ^{sync}_{seq} program fragment,

$$\mathsf{let}\ y = (\mathsf{if}^{v}_{\mathsf{p},\overline{v_k}}\ p\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{in}\ \ldots \tag{5}$$

where the guard p is typed high, implying the computation of branching statement p must be synchronized. The following Caml [22] program fragment implements our dynamic synchronization technique for (5):

```
let f = fun _ -> failwith "timeout" in
Sys.set_signal (Sys.sigvtalrm) (Sys.Signal_handle f);
...
let h = p in let x = ref v in let sync_time = Φ(p)(v_k) in
try
  Unix.setitimer Unix.ITIMER_VIRTUAL sync_time;
  x := if h then p1 else p2;
  while true do () done (* analogue of SYNC-PAD *)
with
  Failure "timeout" -> ();
let y = !x in ...
```
(6)

Note we have empirically validated the above program fragment in OCaml 3.10.1 on Linux. Caml libraries provide the Unix.setitimer system call which sets the virtual timer ITIMER_VIRTUAL to *sync_time*, and the Sys.set_signal function which installs a signal handler f for the signal sigvtalrm generated when ITIMER_VIRTUAL expires. The nonterminating loop 'while true do () done' captures the semantics of the SYNC-PAD rule in Figure 8 – if the computation of the preceding branching statement terminates before the timer expires the while-loop pads out the residual sync time. Observe the signal handler f for sigvtalrm simply throws the exception Failure "timeout" when the sync timer expires; the exception is caught and discarded by the try-catch statement, and the control is then transferred to the next statement 'let $y = !x$ in ...'. Also notice the computed value of the branching statement is assigned to the same heap location, referenced by the variable x, as the default placeholder value v; in effect the value of $!x$ flowing into y is the value of the branch computation if the sync timer expired after the assignment '$x :=$ if h then p_1 else p_2' had happened; otherwise the default placeholder value flows into y.

Note the transformation illustrated above, from program fragment (5) to the program fragment (6), is generic and applicable to function application statements as well; further it can be done automatically by a preprocessor, and to target languages other than Caml (in C, for example, siglongjmp() can be used to perform the nonlocal transfer of control via the signal handler, as opposed to an exception in the above illustration). Also the above implementation denotes only one of the possibly many ways to realize our dynamic synchronization technique; the above method has the advantage of using mechanisms readily available on existing systems. It may be possible to devise more efficient techniques, say for example without the overhead of systems calls and/or context-switching; the exploration of such mechanisms is a direction for future investigation.

The resolution of interval timers provided by the setitimer function is in microseconds; if deemed too coarse, the POSIX.1b high resolution timers (HRTs) provide higher resolution timers in nanoseconds, and can be used instead. However a resolution in the order of nanoseconds is unlikely to impart better efficiency given the overhead of setting up the interval timers along with the subsequent handling of the timer interrupts, both of which are system calls involving context-switching, is nonetheless likely to be on the order of microseconds on modern machines. In fact, given the overhead of runtime synchronization it may be too expensive to individually synchronize each branching statement of a high guard, especially if it is in a tight loop. Coarser granularities for synchronization could be used to amortize the associated overhead; for example, it may be more efficient to synchronize the execution of an entire loop as opposed to each individual branching statement inside it each time around. The exploration of such optimizations is an important direction for future research.

In typical multiprocessing environments there is likely to be some delay in the arrival of the timer interrupt after the corresponding timer has expired, depending on the system load, CPU pipelining, and other factors. However such delays are likely to be "random", that is, likely governed by factors extraneous to the process under synchronization, thus independent of the values, both high and low, local to the process. Such potential delays are, therefore, unlikely to leak information; however a closer scrutiny is desirable to confirm the lack of useful information in such delays.

Securing Resource Channels Time is a form of resource. Recall our philosophy behind securing timing channels is to fix the execution times of high computations based on low values, and if a high computation does not finish in the allotted time then it is aborted and replaced with a placeholder value. A similar philosophy can be applied to other forms of resource channels as well.

Memory usage is a form of resource channel – the memory consumption pattern of a program may depend on secret data, and that could be used by the attacker to gain classified information. For example, one branch of a high branching statement may allocate a lot more memory than the other branch; an attacker could then gain information on the secret guard by tracking the memory usage of that program. One potential approach to securing such memory usage channels is to preallocate a fixed amount of memory for use by the high portion of a program computation; if the high computation exhausts the preallocated memory then the bluffing technique takes over, and all further requests for memory by the high computation are satisfied by recycling memory from the preallocated space. Analogous to sync times, the amount of preallocated high memory may be parameterized over low values.

The technique sketched above for securing memory usage channels can be made robust in presence of garbage collection as well – the high memory is clearly demarcated, hence the garbage collector can be restricted to spend a fixed amount of time to clean the high memory regardless of the amount of garbage in it; as above, the bluffing technique provides a fallback in case the high memory runs out of space or the collection time-limit is exceeded.

Other forms of resource channel such as those associated with utilization of disc space, network sockets, etc. can be secured in a similar manner: by fixing the usage of high computation. The bluffing technique is critical to the soundness of this approach, providing a fallback when such resource fixtures are found to be inadequate.

Guarding against Cache Attacks Consider the following program, adapted from [2],

$$\begin{array}{l} z := x; \\ \mathsf{if}\ (h == 1)\ \mathsf{then}\ z := x\ \mathsf{else}\ z := y; \\ \mathsf{output}^{low}(\text{“done”}); \end{array} \tag{7}$$

where h holds a single high bit. Assuming neither x nor y is in the cpu cache before the execution of this code, the running time of the branching statement is likely to reveal the value of

h – the value of x will likely be cached after the first assignment $z := x$; hence the later assignment $z := x$ in the then-branch will probably run a bit faster than the assignment $z := y$ in the else-branch. This variance in the running time can be used by an attacker to gain information on h. Such information leaks induced by variance in cache behavior are referred to as *cache leaks*, and the corresponding attacks are called *cache attacks*. The capacity of covert timing channels implemented through cache leaks is not likely to be very high but it is certainly high enough to warrant attention, as shown in [2] via a simple experiment; since then many attacks exploiting the timing variability due to cache effects have been demonstrated [30, 29].

Our dynamic synchronization technique equalizes the running times of corresponding branches regardless of the state of the cache; hence, it would be robust against cache attacks for program (7). However, our technique does *not* synchronize the caches at the end of different executions of a high branching statement. Consider the following variant of the above program,

$$\begin{array}{l}\mathsf{if}\ (h == 1)\ \mathsf{then}\ z := x\ \mathsf{else}\ z := y;\\ z := x;\\ \mathsf{output}^{low}(\text{“done”});\end{array} \tag{8}$$

Again assuming neither x nor y is in the cache before the execution of (8), if the then-branch is taken, x is likely to be in the cache, and y likely not, at the end the branching statement; the opposite holds if the else-branch were taken. Consequently, the subsequent assignment $z := x$ may run a bit faster in the case of then-branch compared to the else-branch, revealing the value of h via the timing of the low output “done”.

One simple mitigation strategy would be to flush the entire cache at the end of each synchronized run of a branch, wiping out any possible correlation between the high guard and cached values. The cache can be flushed by reading enough garbage into it [2].

9. Conclusion

We have demonstrated how timing channels can be provably eliminated in both sequential and concurrent programs, in presence of deterministic as well as nondeterministic schedulers. Our techniques advance the current state of the art by showing how timing channels can be eliminated in programs with higher-order functions, with recursive computations under high guards, and in the presence of libraries for which the source code is not available. We have illustrated how our dynamic synchronization technique can be realized on standard computing platforms using interval timers. Furthermore, we believe the technique of bluff computation holds promise in securing resource channels as well. Our solution is general and can be incorporated into arbitrary software products.

The paper makes several technical contributions. Our strong bisimulation proof technique is a general technique for proving noninterference in presence of timing channels. Our method of employing discrete probability distributions, and the associated all-paths-in-parallel operational semantics, allows for a more direct proof of probabilistic noninterference in concurrent programs. The notion of bluff computation generalizes the lenient execution model of [7] to be more broadly applicable.

References

[1] J. Agat. Transforming out timing leaks. In *POPL*, 2000. 1, 3, 4, 22

[2] J. Agat and D. Sands. On confidentiality and algorithms. In *SP*, 2001. 1, 4, 23, 24

[3] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *ESORICS*, 2007. 1, 21

[4] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. In *QAPL*, 2005. 22

[5] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP*, 2001. 1, 4

[6] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, 2003. 1

[7] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *CSFW*, 2004. 3, 22, 24

[8] D. E. R. Denning. *Cryptography and Data Security*. 1982. 6

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982. 1, 10

[10] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. In *BYTECODE*, 2005. 1, 22

[11] W.-M. Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium Research in Security and Privacy*, 1991. 1

[12] B. Jonsson, K. Larsen, and W. Yi. Probabilistic extensions of process algebras. In *Handbook of Process Algebra*, 2001. 22

[13] M. H. Kang and I. S. Moskowitz. A pump for rapid, reliable, secure communication. In *CCS*, 1993. 1

[14] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. In *ESORICS*, 1998. 1

[15] N. Kobayashi and K. Shirane. Type-based information flow analysis for a low-level language. In *APLAS*, 2002. 22

[16] P. Kocher, J. Jaffe, and B. Jun. Using unpredictable information to minimize leakage from smartcards and other cryptosystems. In *USA patent, International Publication number WO 99/63696*, 1999. 1

[17] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, 1996. 1

[18] B. Köpf and D. Basin. Timing-sensitive information flow analysis for synchronous systems. In *ESORICS*, 2006. 1

[19] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *FAST*, 2005. 22

[20] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973. 1

[21] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *ASIAN*, 2006. 22

[22] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Caml Language. `http://caml.inria.fr/`. 2, 23

[23] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Comput. Secur.*, 2003. 22

[24] D. May, H. L. Muller, and N. P. Smart. Non-deterministic processors. In *ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy*, pages 115–129, London, UK, 2001. Springer-Verlag. 1

[25] D. May, H. L. Muller, and N. P. Smart. Random register renaming to foil dpa. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, 2001. 1

[26] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC*, 2005. 22

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. In *POPL '98: Symposium on Principles of programming languages*, 1998. 22

[28] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS*, 2007. 22

[29] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes, 2005. 24

[30] D. Page. Defending against cache based side-channel attacks. *Information Security Technical Report*, 2003. 24

[31] A. D. Pierro, C. Hankin, I. Siveroni, and H. Wiklicky. Tempus fugit: How to plug it. *The Journal of Logic and Algebraic Programming 72 (2007) 173190*, 2007. 22

[32] F. Pottier and V. Simonet. Information flow inference for ML. *TOPLAS*, 2003. 6, 7

[33] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*, 1978. 2

[34] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *CSFW*, 2006. 1, 17, 21, 22

[35] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *PSI '02: Perspectives of System Informatics*, 2001. 22

[36] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *SAS'02: Symposium on Static Analysis*, 2002. 22

[37] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003. 1

[38] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, 2000. 1, 17, 19, 20, 21, 22

[39] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, 2007. 6

[40] I. Siveroni. Filling out the gaps: A padding algorithm for transforming out timing leaks. *Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005)*, 2005. 22

[41] G. Smith. A new type system for secure information flow. In *CSFW*, 2001. 1, 4

[42] G. Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 2006. 21

[43] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, 1998. 1, 21

[44] Volpano and Smith. Eliminating covert flows with minimum typings. In *CSFW*, 1997. 21

[45] R. Wilhelm et al. The worst-case execution time problem – overview of methods and survey of tools. *TECS*, 2007. 3

[46] D. Yu and N. Islam. A typed assembly language for confidentiality. In *APLAS*, 2007. 22

[47] S. Zdancewic and A. Myers. Observational determinism for concurrent program security. *CSFW*, 2003. 1, 4, 21

[48] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. In *Cryptology ePrint Archive, Report 2005/388*, 2005. 1