Compositional non-interference for fine-grained concurrent programs

Frumin, Dan; Krebbers, Robbert; Birkedal, Lars

DOI
10.1109/SP40001.2021.00003

Publication date
2021

Document Version
Accepted author manuscript

Published in
Proceedings - 2021 IEEE Symposium on Security and Privacy, SP 2021

Compositional Non-Interference for Fine-Grained Concurrent Programs

Dan Frumin
Radboud University

Robbert Krebbers
Delft University of Technology

Lars Birkedal
Aarhus University

***Abstract*—Non-interference is a program property that ensures the absence of information leaks. In the context of programming languages, there exist two common approaches for establishing non-interference: type systems and program logics. Type systems provide strong automation (by means of type checking), but they are inherently restrictive in the kind of programs they support. Program logics support challenging programs, but they typically require significant human assistance, and cannot handle modules or higher-order programs.**

To connect these two approaches, we present SeLoC—a separation logic for non-interference, on top of which we build a type system using the technique of logical relations. By building a type system on top of separation logic, we can compositionally verify programs that consist of typed and untyped parts. The former parts are verified through type checking, while the latter parts are verified through manual proof.

The core technical contribution of SeLoC is a relational form of weakest preconditions that can track information flow using separation logic resources. SeLoC is fully machine-checked, and built on top of the Iris framework for concurrent separation logic in Coq. The integration with Iris provides seamless support for fine-grained concurrency, which was beyond the reach of prior type systems and program logics for non-interference.

***Index Terms*—non-interference, logical relations, separation logic, fine-grained concurrency, Coq, Iris**

I. Introduction

Non-interference is a form of *information flow control* (IFC) used to express that confidential information cannot leak to attackers. To establish non-interference of modern programs, it is crucial to develop verification techniques that support challenging programming paradigms and programming constructs such as concurrency. Furthermore, to scale up these techniques to larger programs, it is important that they are compositional. That is, they should make it possible to establish non-interference of program modules in isolation, without having to consider all possible interference from the environment and other program modules.

Much effort has been put into developing these verification techniques. In terms of expressivity, techniques have been developed that support dynamically allocated references and higher-order functions [1]–[3], and concurrency [4]–[10]. Despite recent advancements, the expressivity of available techniques for non-interference still lags behind the expressivity of techniques for functional correctness, which have seen major breakthroughs since the seminal development of concurrent separation logic [11], [12]. There are several reasons for this.

First, a lot of prior work on non-interference focused on type systems and type system-like logics, *e.g.,* [1], [4], [6], [9], [10]. Such systems provide strong automation (by means of type checking), but lack capabilities to reason about functional correctness, and are thus inherently restrictive in the kind of programs they can verify. For example, it may be the case that the confidentiality of the contents of a reference depends on runtime information instead of solely static information (this is called *value-dependent classification* [7], [13]–[16]).

Second, proving non-interference is harder than proving functional correctness. While functional correctness is a property about each single run of a program, non-interference is stated in terms of multiple runs of the same program. One has to show that for different values of confidential inputs, the attacker cannot observe a different behavior.

To overcome the aforementioned shortcomings, we take a new approach that combines program logics and type systems: we present a concurrent separation logic for non-interference on top of which we build a type system for non-interference. Program modules whose non-interference relies on functional correctness (and thus cannot be type checked) can be assigned a type through a manual proof in our separation logic. This combination of separation logic and type checking makes it possible compositionally to establish non-interference of programs that consist of untyped and typed parts.

Although ideas from concurrent separation logic have been employed in the context of non-interference before [9], [10], we believe that in the context of non-interference the combination of typing and separation logic is new. Moreover, our approach provides a number of other advantages compared to prior work:

- Our separation logic supports *fine-grained concurrency*. That is, it can verify programs that use low-level atomic operations like compare-and-set to implement lock-free concurrent data structures and high-level synchronization mechanisms such as locks/mutexes. In prior work, such mechanisms were taken to be language primitives.
- Our separation logic is *higher-order*, making it possible to assign very general specifications to program modules.
- Our separation logic is *relational*, making it possible to reason about multiple runs of a program with different values for confidential inputs.
- Our separation logic provides a powerful *invariant* mechanism to describe protocols on the shared state, making it possible to reason about sophisticated forms of sharing, as in value-dependent classifications.

In order to build our logic we make use of the Iris framework for concurrent separation logic [17]–[20], which provides basic

building blocks, including the invariant mechanism. To combine typing and separation logic, we follow recent work on *logical relations* in Iris [21]–[26], but apply it to non-interference instead of functional correctness or contextual refinement.

Contributions. We introduce **SeLoC**, the first separation logic for non-interference that combines typing and manual proof.

- We present a number of challenging examples that can be verified using SeLoC (§ II).
- SeLoC supports a language with fine-grained concurrency, higher-order functions, and dynamic (higher-order) references (§ III-A). SeLoC is sound w.r.t. a standard timing-sensitive notion of non-interference—*strong low-bisimulations*—by Sabelfeld and Sands [5] (§ III-B).
- To verify challenging programs, SeLoC features a relational version of weakest preconditions, which integrates seamlessly with the powerful mechanism for invariants and protocols of the Iris framework (§ IV).
- Using the technique of logical relations, we build a type system on top of SeLoC. By building a type system on top of separation logic, we can establish non-interference of programs that consist of typed and untyped parts (§ V).
- To compose proofs of program modules that cannot be type checked (because their interface relies on functional correctness), we show how to express modular separation logic specifications for non-interfence in SeLoC (§ VI).
- We prove soundness of SeLoC by constructing a bisimulation out of a separation logic proof (§ VII).
- We have mechanized SeLoC, its type system, its soundness proof, and all examples in the paper and appendix, in Coq (§ VIII). The mechanization can be found online at [27].

II. Motivating Examples

Before we proceed with the formal development of the paper in § III, we present a number of challenging programs to demonstrate the expressivity of SeLoC.

A. Modularity and data structures

To guarantee non-interference, one should prove that *high-sensitivity* (*i.e.,* confidential) information cannot leak via *low-sensitivity* (*i.e.,* publicly observable) outputs. Apart from such explicit leaks, one has to prove the absence of implicit leaks that arise from the timing behavior of the program. To avoid timing leaks, Agat and Sands [28] outlined the "worst-case principle": a non-interfering algorithm operating on high-sensitivity data should have the same best-case and worst-case execution time. We apply this design principle to a set data structure that stores high-sensitivity elements. The implementation can be type checked using our approach, automatically providing a proof of timing-sensitive non-interference.

To encapsulate the internal set representation, we first present the interface of our data structure. This interface is given using *closures* (*i.e.,* higher-order functions):[1]

$$\textbf{val}\ \mathit{new\_set} : \texttt{unit} \to \left\{ \begin{array}{l} \mathit{lookup} : \texttt{int}^{\mathbf{H}} \to \texttt{bool}^{\mathbf{H}}; \\ \mathit{insert} : \texttt{int}^{\mathbf{H}} \to \texttt{unit} \end{array} \right\}$$

[1]When using modules or classes, the same kind of considerations apply.

$$\begin{array}{l} \textbf{let}\ \mathit{new\_set}\ () = \\ \quad \textbf{let}\ k = \textbf{ref}(1)\ \textbf{in} \\ \quad \textbf{let}\ \mathit{arr} = \textbf{ref}(\mathit{new\_array}\ 1\ \textbf{None})\ \textbf{in} \\ \quad \left\{ \begin{array}{l} \mathit{lookup}\ x = \\ \quad \mathit{lookup\_loop}\ (!\,\mathit{arr})\ (!\,k)\ 0\ (\mathit{cap}\ (!\,k))\ x\ \textbf{false} \\ \mathit{insert}\ x = \mathit{insert\_loop}\ \mathit{arr}\ k\ 0\ x \end{array} \right\} \\ \textbf{let rec}\ \ \mathit{lookup\_loop}\ a\ k\ l\ r\ x\ \mathit{is\_found} = \\ \quad \textbf{if}\ k = 0\ \textbf{then}\ \mathit{is\_found}\ \textbf{else} \\ \quad \textbf{let}\ i = (l + r)/2\ \textbf{in} \\ \quad \textbf{let}\ e = \mathit{array\_get}\ a\ i\ \textbf{in} \\ \quad \textbf{let}\ \mathit{lr}_1 = (i + 1, r)\ \textbf{in}\ \ \textbf{let}\ \mathit{lr}_2 = (l, i - 1)\ \textbf{in} \\ \quad \textbf{let}\ (l, r) = \textbf{if}\ (e < x)\ \textbf{then}\ \mathit{lr}_1\ \textbf{else}\ \mathit{lr}_2\ \textbf{in} \\ \quad \mathit{lookup\_loop}\ a\ (k - 1)\ l\ r\ x\ (\mathit{is\_found} \vee (e = x)) \\ \textbf{let rec}\ \ \mathit{insert\_loop}\ \mathit{arr}\ k\ i\ x = \ldots \end{array}$$

Figure 1. Implementation of a set using the "worst-case principle".

The function *new_set* allocates an empty set, and returns a record with functions that operate on the set. The function *lookup* takes a high-sensitivity integer—typed as $\texttt{int}^{\mathbf{H}}$, where $\mathbf{H}$ refers to the *high*-sensitivity of the data— and returns a high-sensitivity Boolean—typed as $\texttt{bool}^{\mathbf{H}}$—that signifies whether the argument is in the set or not. The function *insert* takes a high-sensitivity integer, and adds it to the set.

Figure 1 shows an implementation of our set interface using a sorted dynamic array.[2] To implement the function *lookup*, we make use of binary search—but with a twist to avoid timing leaks. An ordinary version of binary search would terminate once it has found the element, making it possible to observe if the element is in the set via timing. Our implementation ensures that *lookup* takes the same time regardless of whether the element is in the set. To achieve that, we represent the set using an array whose size n satisfies $\mathit{cap}(k) = n$, for some k:

$$\mathit{cap}(0) = 0 \qquad \mathit{cap}(k+1) = 1 + 2 \cdot \mathit{cap}(k)$$

This guarantees that the array can be recursively partitioned into two sub-arrays of the same size and a pivot element in the middle. If the number of actual elements in the set is less than $\mathit{cap}(k)$, the array is padded with a dummy element.[3]

If at some iteration of *lookup_loop* we find that the element x is present in the array, we make note of that fact but still continue with the recursion until the array is no longer splittable. Thus, the function *lookup* is always executed with k levels of recursion for an array of size $\mathit{cap}(k)$. In the implementation of *lookup_loop* we pass the parameter k and decrease it on every recursive call.

[2]The full implementation can be found in the Coq mechanization. The full implementation moreover makes use of locks to obtain thread-safety.

[3]For comparisons $<$ and equality $=$ checks we assume that the dummy element **None** is the greatest element and that it is not equal to any actual element in the array, which are of the form $\textbf{Some}(x)$.

The function *insert* traverses the whole array and is thus always executed with $cap(k)$ levels of recursion. If the array is full, then it is dynamically resized to the size $cap(k+1)$. In summary, both *lookup* and *insert* operations employ a low-sensitivity termination condition.

We use our type system (described in § V) to type check the implementation against the interface. Of special note here is the type checking of the **if** branching. In the implementation of *lookup_loop* and *insert_loop* we branch on high-sensitivity data. Notably, in *lookup_loop* we compare the argument x with the pivot e (both are high-sensitivity integers), and descend into one of the partitions of the array depending on this comparison. Branching on high-sensitivity data is not secure in general, but in this case the branching is secure. This is because both branches simply return variables (lr_1 and lr_2), *i.e.,* they do not perform any computations, and thus do not leak information about the high-sensitivity condition via timing.[4]

B. Typing via manual proof

The example in the previous section made use of various operations on arrays: *array_make*, *array_get*, and *array_set*. When reasoning about the set data structure, we assumed that these array operations are safe and secure, *i.e.,* when one tries to access an out-of-bounds index, *array_get* returns a dummy element, instead of reading arbitrary memory.

The programming language that we consider does not have safe arrays as a primitive construct. Instead, safe arrays are implemented as a library: an array is stored together with its length, and the unsafe operations are protected by dynamic checks. Naturally, such operations cannot be type checked in an ML-style type system [29], because their safety and security depends on functional correctness. However, one of the core features of our approach is that such functions can be assigned types through a manual separation logic proof in SeLoC. Such a manual proof takes functional properties (e.g., that the index is within the array bounds) into account. Once we manually verify that the array library satisfies the desired typing, we can compose it with the type checked example from the previous section to obtain a library that guarantees safety and non-interference for its clients.[5]

The combination of typing and manual proof is important for compositionality and scalability: challenging library code whose security relies on functional correctness (such as the library for safe arrays) can be manually verified using separation logic, and then used to automatically type check other libraries (such as the set data structure).

C. Fine-grained concurrency

As shown in § II-B, the ability to fall back to a manual proof is useful to assign types to code that uses operations such as array indexing whose safety and security relies on functional correctness. This ability becomes even more pertinent for (fine-grained) concurrent programs, where the safety and security

[4]In a low-level language like C the branching can be written using arithmetic.

[5]The proof of the array library and its integration in the type checking of the set data structure can be found in the Coq mechanization.

$$
\begin{aligned}
&\textbf{let rec } \mathit{thread1}\ \mathit{out}\ r = (\textbf{if } \neg\, !\, r.\mathit{is\_classified}\\
&\qquad\qquad\qquad\qquad \textbf{then } \mathit{out} \leftarrow !\, r.\mathit{data} \textbf{ else } ());\\
&\qquad\qquad\qquad\qquad \mathit{thread1}\ \mathit{out}\ r\\
&\qquad\quad \textbf{let } \mathit{thread2}\ r = r.\mathit{data} \leftarrow 0;\\
&\qquad\qquad\qquad\qquad r.\mathit{is\_classified} \leftarrow \textbf{false}\\
&\textbf{let } \mathit{prog}\ \mathit{out}\ \mathit{secret} = \textbf{let } r = \left\{\begin{array}{l} \mathit{data} = \textbf{ref}(\mathit{secret});\\ \mathit{is\_classified} = \textbf{ref}(\textbf{true}) \end{array}\right\}\\
&\qquad\qquad\qquad\qquad \textbf{in } \mathit{thread1}\ \mathit{out}\ r \mid\mid \mathit{thread2}\ r
\end{aligned}
$$

Figure 2. Lock-free value-dependent classification.

can depend on specific protocols on data that is shared between threads.

To demonstrate the application to concurrency, we consider the program *prog* in Figure 2, which is a lock-free version of a similar lock-based program in [10]. The program runs two threads in parallel, both of which operate on a reference $r.data$. The data in this reference has a *value-dependent classification*: the value of the flag $r.is\_classified$ determines the sensitivity of $r.data$. If the flag $r.is\_classified$ is set to **false**, then the data stored in $r.data$ is classified with low-sensitivity, and if it is set to **true**, the the data is classified with high-sensitivity. The record r initially contains high-sensitivity data from the integer variable *secret*. The first thread *thread1* checks if the record r is classified (*i.e.,* the flag $r.is\_classified$ is **true**), and if it is not, it leaks the data $r.data$ to an attacker-observable channel *out*. The second thread *thread2* overwrites the data stored in r and resets the classification flag.

Due to the precise interplay of the two threads, the program *prog* is secure, in the sense that it does not leak the data *secret* onto the public channel *out*. Since our example does not use locks, there are more possible interleavings than in the original example in [10], and consequently there are more things that could potentially go wrong in *thread1*:

1) the data $r.data$ can still be classified even if the bit $r.is\_classified$ is set to **false**;
2) the classification of the data stored in r might change between reading the field *is_classified* and reading the actual data from the field *data*.

Notice that if we replace the second thread by the expression below, where the two operations in *thread2* have been swapped, then we would violate the first condition:

$$\textbf{let } \mathit{thread2}_{bad}\ r = r.\mathit{is\_classified} \leftarrow \textbf{false};\ \ r.\mathit{data} \leftarrow 0$$

To verify that both of these situations cannot occur, we have to establish a *protocol* on accessing the record r. The protocol should ensure that at the moment of reading $r.is\_classified$ the data $r.data$ has the correct classification (ruling out situation 1). The protocol should also ensure a form of *monotonicity*: whenever the classification becomes low (*i.e.,* $r.is\_classified$ becomes **false**), $r.data$ is not going to contain high-sensitivity data for the rest of the program (ruling out situation 2).

The security of *thread1*, and the whole program, depends on the specific protocol attached to the record r and that the protocol is followed by all the components that operate on it. In particular, for this example the security depends on the fact that classification only changes in a *monotone* way. We outline the proof of safety and security of this example in § IV-D.

D. Higher-order functions and dynamic references

As shown in this section, higher-order functions are useful for modularity—they can be used to model interfaces. However, since they can operate on encapsulated state, they are difficult to reason about. Fortunately, SeLoC's protocol mechanism is also applicable to proving non-interference of functions with encapsulated state. Consider the program awk, a variation of the "awkward example" of Pitts and Stark [30]:

$$\textbf{let}\ awk\ v = \textbf{let}\ x = \textbf{ref}(v)\ \textbf{in}\ \lambda\, f.\, x \leftarrow 1; f(); !\, x$$

When applied to a value v, the program awk returns a closure that, when invoked, always returns low-sensitivity data from the reference x, even if the original value v has high-sensitivity. Intuitively, $awk\ v$ returns a closure that does not leak any data, even if the original value v passed to awk had high-sensitivity. The lack of leaks crucially relies on the following facts:

- the reference x is allocated in, and remains local to, the closure, it cannot be accessed without invoking the closure;
- the reference x can be updated only in a monotone way: once the original value v gets overwritten with 1, the reference x never holds a high-sensitivity value again.

To see why second condition is important, consider awk_{bad}, which violates the monotonicity, and is thus not secure:

$$\textbf{let}\ awk_{bad}\ v = \textbf{let}\ x = \textbf{ref}(v)\ \textbf{in}\ \lambda\, f.\, x \leftarrow v; x \leftarrow 1; f(); !\, x$$

Let $h = awk_{bad}\ v$ for a high-sensitivity value v. Now, when running $h\ (\lambda\, x.\, \textbf{fork}\ \{h(id)\})$, an attacker could influence the scheduler so that the first dereference $!\, x$ happens just after the assignment $x \leftarrow v$ in the forked-off thread, causing v to leak.

Pitts and Stark studied the "awkward example" to motivate the difficulties of reasoning about higher-order functions and state. They were interested in contextual equivalence, but as we can see, similar considerations apply to non-interference.

III. Preliminaries

In this section we describe the programming language that we consider in this paper (§ III-A), and the non-interference property that SeLoC establishes (§ III-B).

A. Object language and scheduler semantics

SeLoC is defined over an ML-like programming language [29], called HeapLang, with higher-order mutable references, recursion, and **fork**-based concurrency. HeapLang is the default programming language that is shipped with Iris [31]. Its values and expressions are:

$$\begin{aligned} v \in \mathsf{Val} &::= \textbf{rec}\ f\ x = e \mid (v_1, v_2) \mid \textbf{true} \mid \textbf{false} \mid \ldots \\ e, s, t \in \mathsf{Expr} &::= x \mid \textbf{rec}\ f\ x = e \mid e_1(e_2) \mid \textbf{fork}\ \{e\} \\ &\quad \mid \textbf{ref}(e) \mid\ !\, e \mid e_1 \leftarrow e_2 \mid \textbf{CAS}(e_1, e_2, e_3) \mid \ldots \end{aligned}$$

We omit the usual operations on pairs, sums, and integers. The *atomic* compare-and-set operation $\textbf{CAS}(e_1, e_2, e_3)$ checks if the value stored at the location e_1 is equal to e_2, and, if so, sets the value at e_1 to e_3. The **fork** $\{e\}$ construct creates a new thread, which will execute the expression e. The construct **rec** $f\ x = e$ is a recursive λ-function, whose body e can refer to the function f itself and the argument x.

We use the following syntactic sugar: $(\lambda\, x.\, e) \triangleq (\textbf{rec}\ \_\ x = e)$, $(\textbf{let}\ x = e_1\ \textbf{in}\ \ e_2) \triangleq ((\lambda\, x.\, e_2)\ e_1)$, and $(e_1; e_2) \triangleq (\textbf{let}\ \_ = e_1\ \textbf{in}\ \ e_2)$, where we use _ as an anonymous binder, in place of a variable name. HeapLang has no primitive syntax for records, so they are modeled using pairs. Arrays are omitted in the paper, but they are present in the Coq mechanization.

HeapLang features dynamic thread creation, so we can implement the parallel composition operation using **fork**:

$$\begin{aligned} \textbf{let rec}\ join\ x = \textbf{match}\ !\, x\ \textbf{with}\ \textbf{Some}(v) &\rightarrow v \\ \mid \textbf{None} &\rightarrow join\ x \\ \textbf{let}\ par(f_1, f_2) = \textbf{let}\ x = \textbf{ref}(\textbf{None})\ &\textbf{in} \\ \textbf{fork}\ \{x &\leftarrow \textbf{Some}(f_1())\} \\ \textbf{let}\ v_2 = f_2()\ \textbf{in}\ \ (join\ x, v_2) & \\ e_1 \mid\mid e_2 \triangleq par(\lambda\, \_.\, e_1, \lambda\, \_.\, e_2) & \end{aligned}$$

The operational semantics of HeapLang is split into three reduction relations: thread-local head reduction $\rightarrow_{\mathsf{h}}$, thread-local reduction $\rightarrow_{\mathsf{t}}$, and thread-pool reduction $\rightarrow_{\mathsf{tp}}$. The thread-local head reduction is of the form $(e_1, \sigma_1) \rightarrow_{\mathsf{h}} (e_2, \sigma_2)$, where e_i is an expression, and σ_i is a heap, *i.e.,* a finite map from locations to values ($\mathsf{State} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val}$). Since the security condition that we consider (§ III-B) is tailored towards a deterministic thread-local semantics, we parameterize the operational semantics by an *allocation oracle* $A : \mathsf{State} \rightarrow \mathsf{Loc}$: a function from heaps to locations satisfying $A(\sigma) \notin \sigma$. With the allocation oracle, the allocation head reduction is as follows:

$$(\textbf{ref}(v), \sigma) \rightarrow_{\mathsf{h}} (A(\sigma), \sigma\,[A(\sigma) \leftarrow v])$$

The other rules for the head reduction relation are standard and can be found in the Coq mechanization.

The thread-local head reduction is lifted to the thread-local reduction using *call-by-value evaluation contexts*:

$$K \in \mathit{ECtx} ::= [\bullet] \mid K(v_2) \mid e_1(K) \mid \textbf{if}\ K\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \ldots$$

The thread-local reduction is of the form $(e_1, \sigma_1) \rightarrow_{\mathsf{t}} (\vec{e}_2, \sigma_2)$. The second component contains a list $\vec{e}_2$ of expressions to accommodate forked-off threads as in STEP-FORK:

$$\text{STEP-LIFT}\ \frac{(e_1, \sigma_1) \rightarrow_{\mathsf{h}} (e_2, \sigma_2)}{(K[\,e_1\,], \sigma_1) \rightarrow_{\mathsf{t}} (K[\,e_2\,], \sigma_2)} \qquad \text{STEP-FORK}\ \frac{\vec{e} = K[\,()\,]\ e}{(K[\,\textbf{fork}\ \{e\}\,], \sigma) \rightarrow_{\mathsf{t}} (\vec{e}, \sigma)}$$

The thread-pool reduction $\rightarrow_{\mathsf{tp}}$ is defined by lifting the thread-local reduction to *configurations* $(\vec{e}, \sigma)$. Here, $\vec{e}$ contains all threads, including values for the threads that have terminated. In the definition of $\rightarrow_{\mathsf{tp}}$ we non-deterministically select an expression to take a thread-local step:

$$\frac{(e_i, \sigma_1) \rightarrow_{\mathsf{t}} (e_i' \vec{e}, \sigma_2)}{(e_0 \ldots e_i \ldots e_n, \sigma_1) \rightarrow_{\mathsf{tp}} (e_0 \ldots e_i' \ldots e_n \vec{e}, \sigma_2)}$$

B. Strong low-bisimulations

To state the soundness theorem of SeLoC in § IV-C, we adapt a timing-sensitive notion of non-interference for concurrent programs known as *strong low-simulations* on configurations by Sabelfeld and Sands [5]. To define this notion, we first fix a set $\mathfrak{L} \subseteq \mathsf{Loc}$ of *output locations*, which we assume to be low-sensitivity observable locations. For simplicity, we require these locations to contain integers.

Definition 1. Heaps σ_1 and σ_2 are *low-equivalent for output locations* $\mathfrak{L} \subseteq \mathsf{Loc}$, notation $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$, if they agree on all the $\mathfrak{L}$-locations, *i.e.,* $\forall \ell \in \mathfrak{L}.\, \sigma_1(\ell) = \sigma_2(\ell) \neq \bot \wedge \sigma_1(\ell) \in \mathbb{Z}$.

Definition 2. A *a strong low-bisimulation* is a partial equivalence (*i.e.,* symmetric and transitive) relation $\mathcal{R}$ on configurations such that:

1) If $(v\vec{e}, \sigma_1)\ \mathcal{R}\ (w\vec{s}, \sigma_2)$, then $v = w$;
2) If $(\vec{e}, \sigma_1)\ \mathcal{R}\ (\vec{s}, \sigma_2)$, then $|\vec{e}| = |\vec{s}|$ and $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$;
3) If $(e_0 \dots e_i \dots e_n, \sigma_1)\ \mathcal{R}\ (s_0 \dots s_i \dots s_n, \sigma_2)$ and $(e_i, \sigma_1) \to_{\mathsf{t}} (e_i'\vec{e}, \sigma_1')$, then there exist an s_i', $\vec{s}$ and σ_2' such that:
   - $(s_i, \sigma_2) \to_{\mathsf{t}} (s_i'\vec{s}, \sigma_2')$;
   - $(e_0 \dots e_i' \dots e_n\vec{e}, \sigma_1')\ \mathcal{R}\ (s_0 \dots s_i' \dots s_n\vec{s}, \sigma_2')$.

Notice that the first expression in the thread-pool is the main thread. The first condition in Definition 2 thus states that the return values of the main-thread should agree.

To model the input/high-sensitivity data we use free variables. For simplicity we assume that the input data consists of integers. We then arrive at the following top-level definition of security.

Definition 3 (Security)**.** An expression e with free variables $\vec{x}$ is *secure* if for any heap σ with $\sigma \sim_{\mathfrak{L}} \sigma$, and any sequences of integers $\vec{i}$, $\vec{j}$ with $|\vec{i}| = |\vec{j}| = |\vec{x}|$, there exists a strong low-bisimulation $\mathcal{R}$ such that $(e[\vec{i}/\vec{x}], \sigma)\ \mathcal{R}\ (e[\vec{j}/\vec{x}], \sigma)$.

C. Non-determinism and non-interference

The semantics presented in § III-A is deterministic on the thread-local level, but we can still account for non-determinism arising from a scheduler. Consider the program *rand*, which uses intrinsic non-determinism of the thread-pool semantics to return either **true** or **false**:

$$\textbf{let}\ \mathit{rand}\ () = \textbf{let}\ x = \textbf{ref}(\textbf{true})\ \textbf{in}\ \textbf{fork}\ \{x \leftarrow \textbf{false}\}\,;!\,x$$

This program is secure w.r.t. Definition 3 (we will prove this in § IV using SeLoC).

It is worth pointing out that if we modify the program and insert an additional assignment of a high-sensitivity value h to x, then the resulting program is *not* secure:

$$\textbf{let}\ \mathit{rand}_{bad}\ () = \textbf{let}\ x = \textbf{ref}(\textbf{true})\ \textbf{in}\ \textbf{fork}\ \{x \leftarrow h\}\,;\textbf{fork}\ \{x \leftarrow \textbf{false}\}\,;!\,x$$

The program is not secure because an attacker can pick a scheduler that always executes the leaking assignment, or, even simpler, can run the program many times under the uniform scheduler. Because the program is not secure, we cannot prove it in SeLoC. In SeLoC, we would verify each thread separately, and we would not be able to verify the forked-off thread $x \leftarrow h$ (precisely because it makes the non-determinism of assignments to the reference x dangerous).

IV. Overview of SeLoC

We provide an overview of SeLoC by presenting its proof rules for relational reasoning (§ IV-A), its invariant mechanism (§ IV-B), its soundness theorem (§ IV-C), and finally its protocol mechanism (§ IV-D), which we apply to the verification of the program *prog* from § II-C. The grammar of SeLoC is:

$$\begin{aligned} P, Q \in \mathsf{Prop} ::= {} & \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid P * Q \\ & \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_\theta v \mid \mathsf{awp}_\theta\, e\, \{\Phi\} \qquad (\theta \in \{\mathsf{L}, \mathsf{R}\}) \\ & \mid \mathsf{dwp}_{\mathcal{E}}\, e_1 \,\&\, e_2\, \{\Phi\} \\ & \mid \boxed{P}^{\mathcal{N}} \mid \triangleright P \mid \Box P \mid {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \mid \dots \end{aligned}$$

SeLoC features the standard separation logic connectives like separating conjunction ($*$) and magic wand ($-\!\!*$). Since SeLoC is based on Iris [17]–[20], it incorporates all the Iris connectives and modalities, in particular the *later modality* ($\triangleright$) for dealing with recursion, the *persistence modality* ($\Box$) for dealing with shareable resources, and the *invariant connective* ($\boxed{P}^{\mathcal{N}}$) and the *update modality* (${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$) for establishing and relying on protocols. We will not introduce the Iris connectives in detail, but rather explain them on a by-need basis. An interested reader is referred to [20], [32] for further details. Various connectives are annotated with *name spaces* $\mathcal{N} \in \mathit{InvName}$ and *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ to handle some bookkeeping. When the mask is omitted, it is assumed to be $\top$, the largest mask. We let $\Rightarrow_{\mathcal{E}}$ denote ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}}$. Readers who are unfamiliar with Iris can safely ignore the name spaces and invariant masks.

A selection of proof rules of SeLoC is given in Figure 3. Each inference rule $\dfrac{P_1 \ \dots \ P_n}{Q}$ in this paper should be read as an entailment $P_1 * \dots * P_n \vdash Q$. In the subsequent sections we explain and motivate the rules of SeLoC.

A. Relational reasoning

The quintessential connective of SeLoC is the *double weakest precondition* $\mathsf{dwp}_{\mathcal{E}}\, e_1 \,\&\, e_2\, \{\Phi\}$. It intuitively expresses that any two runs of e_1 and e_2 are related in a lock-step bisimulation-like way, and that the resulting values of any two terminating runs are related by the *postcondition* $\Phi : \mathsf{Val} \to \mathsf{Val} \to \mathsf{Prop}$. We refer to e_1 (resp. e_2) as the *left-hand side* (resp. the *right-hand side*). The double weakest precondition is defined such that if $\forall \vec{i}\, \vec{j} \in \mathbb{Z}.\, \mathsf{dwp}\, e[\vec{i}/\vec{x}] \,\&\, e[\vec{j}/\vec{x}]\, \{v_1\, v_2.\, v_1 = v_2\}$ (with $\vec{x}$ the free variables of e), then e is secure. We defer the precise soundness statement to § IV-C.

A selection of rules for double weakest preconditions[6] are given in Figure 3. Some of these rules are generalizations of the ordinary weakest precondition rules (*e.g.,* DWP-VAL, DWP-WAND, DWP-FUPD, DWP-BIND). The more interesting rules are the *symbolic execution* rules, which allow executing the programs

[6]Some of the SeLoC rules involve the *later modality* $\triangleright$, which is standard for dealing with recursion and impredicative invariants [20, Section 5.5]. The occurrences of $\triangleright$ can be ignored for the purposes of this paper.

$$\textsc{DWP-VAL}\ \frac{\Phi(v_1, v_2)}{\mathsf{dwp}_{\mathcal{E}}\ v_1\ \&\ v_2\ \{\Phi\}} \qquad \textsc{DWP-WAND}\ \frac{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Psi\} \qquad (\forall v_1\ v_2.\ \Psi(v_1, v_2) \mathrel{-\!\!*} \Phi(v_1, v_2))}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

$$\textsc{DWP-FUPD}\ \frac{\Rrightarrow_{\mathcal{E}} \mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{v_1\ v_2.\ \Rrightarrow_{\mathcal{E}} \Phi(v_1, v_2)\}}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}} \qquad \textsc{DWP-BIND}\ \frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{v_1\ v_2.\ \mathsf{dwp}\ K_1[\,v_1\,]\ \&\ K_2[\,v_2\,]\ \{\Phi\}\}}{\mathsf{dwp}\ K_1[\,e_1\,]\ \&\ K_2[\,e_2\,]\ \{\Phi\}}$$

$$\textsc{DWP-PURE}\ \frac{e_1 \to_{\mathsf{pure}} e_1' \qquad e_2 \to_{\mathsf{pure}} e_2' \qquad \triangleright\, \mathsf{dwp}\ e_1'\ \&\ e_2'\ \{\Phi\}}{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}} \qquad \textsc{DWP-FORK}\ \frac{\triangleright\, \mathsf{dwp}\ e_1\ \&\ e_2\ \{\mathsf{True}\} \qquad \triangleright\, \Phi((), ())}{\mathsf{dwp}_{\mathcal{E}}\ (\mathbf{fork}\ \{e_1\})\ \&\ (\mathbf{fork}\ \{e_2\})\ \{\Phi\}}$$

$$\textsc{DWP-AWP}\ \frac{\mathsf{awp}_{\mathsf{L}}\ e_1\ \{\Psi_1\} \qquad \mathsf{awp}_{\mathsf{R}}\ e_2\ \{\Psi_2\} \qquad (\forall v_1, v_2.\ (\Psi_1(v_1) * \Psi_2(v_2)) \mathrel{-\!\!*} \triangleright\, \Phi(v_1, v_2))}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}} \qquad \textsc{AWP-STORE}\ \frac{\ell \mapsto_\theta v_1 \qquad (\ell \mapsto_\theta v_2 \mathrel{-\!\!*} \Phi())}{\mathsf{awp}_\theta\ \ell \leftarrow v_2\ \{\Phi\}}$$

$$\textsc{AWP-LOAD}\ \frac{\ell \mapsto_\theta v \qquad (\ell \mapsto_\theta v \mathrel{-\!\!*} \Phi(v))}{\mathsf{awp}_\theta\ !\ell\ \{\Phi\}} \qquad \textsc{AWP-ALLOC}\ \frac{\forall \ell.\ \ell \mapsto_\theta v \mathrel{-\!\!*} \Phi(\ell)}{\mathsf{awp}_\theta\ \mathbf{ref}(v)\ \{\Phi\}} \qquad \textsc{DWP-INV-ALLOC}\ \frac{P \qquad (\boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\})}{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}} \qquad \textsc{INV-DUP}\ \frac{\boxed{P}^{\mathcal{N}}}{\boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}}$$

$$\textsc{DWP-INV}\ \frac{\boxed{P}^{\mathcal{N}} \qquad (\triangleright\, P \mathrel{-\!\!*} \mathsf{dwp}_{\mathcal{E}-\mathcal{N}}\ e_1\ \&\ e_2\ \{v_1\ v_2.\ P * \Phi(v_1, v_2)\}) \qquad \mathrm{atomic}(e_1) \qquad \mathrm{atomic}(e_2) \qquad \mathcal{N} \in \mathcal{E}}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

Figure 3. A selection of the proof rules of SeLoC.

on both sides in a lock-step fashion. If both sides involve a pure-redex, we can use DWP-PURE. The premises $e \to_{\mathsf{pure}} e'$ denote that e deterministically reduces to e' without any side-effects (*e.g.,* $(\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ e\ \mathbf{else}\ t) \to_{\mathsf{pure}} e$). If both sides involve a fork, we can use the rule DWP-FORK, which is a generalization of Iris's fork rule to the relational case. To explain SeLoC's rules for symbolic execution of heap-manipulating expressions, we need to introduce some additional machinery:

- Due to SeLoC's relational nature, there are left- and right-hand side versions of the *points-to connectives* $\ell \mapsto_\theta v$, where $\theta \in \{\mathsf{L}, \mathsf{R}\}$, which denote that the value v of location ℓ in the heap associated with the left-hand side program and the right-hand side program, resp.
- To avoid a quadratic explosion in combinations of all possible heap-manipulating expressions on the left- and the right-hand side, SeLoC includes a unary weakest precondition $\mathsf{awp}_\theta\ e\ \{\Phi\}$ for atomic and fork-free expressions. The rules for unary weakest preconditions (*e.g.,* AWP-STORE, AWP-LOAD, AWP-ALLOC) are similar to those of Iris, but each rule is parameterized by a side $\theta \in \{\mathsf{L}, \mathsf{R}\}$.

The rule DWP-AWP connects dwp and awp_θ. For instance, using DWP-AWP, AWP-STORE, and AWP-LOAD, we can derive:

$$\frac{\ell_1 \mapsto_{\mathsf{L}} v_1 \qquad \ell_2 \mapsto_{\mathsf{R}} v_2 \qquad (\ell_1 \mapsto_{\mathsf{L}} v_1 * \ell_2 \mapsto_{\mathsf{R}} v_2') \mathrel{-\!\!*} \mathsf{dwp}\ v_1\ \&\ ()\ \{\Phi\}}{\mathsf{dwp}\ !\ell_1\ \&\ (\ell_2 \leftarrow v_2')\ \{\Phi\}}$$

B. Invariants

Let us demonstrate, by means of an example, how to use the symbolic execution rules together with the powerful invariant mechanism of Iris. Recall the *rand* example from § III-C. We can use invariants to prove the following:

Proposition 4. $\mathsf{dwp}\ \mathit{rand}\ ()\ \&\ \mathit{rand}\ ()\ \{v_1\ v_2.\ v_1 = v_2\}$.

Proof. First we use DWP-PURE to symbolically execute a β-reduction. We then use DWP-BIND to "focus" on the $\mathbf{ref}(\mathbf{true})$ subexpression, leaving us with the goal:

$$\mathsf{dwp}\ \mathbf{ref}(\mathbf{true})\ \&\ \mathbf{ref}(\mathbf{true})\ \{\Phi\}$$
$$\text{where } \Phi(\ell_1, \ell_2) \triangleq \mathsf{dwp}\ \mathbf{let}\ x = \ell_1\ \mathbf{in}\ \ldots\ \&\ \mathbf{let}\ x = \ell_2\ \mathbf{in}\ \ldots\ \{v_1\ v_2.\ v_1 = v_2\}$$

We then symbolically execute the allocation, using DWP-AWP and AWP-ALLOC, obtaining $\ell_1 \mapsto_{\mathsf{L}} \mathbf{true}$ and $\ell_2 \mapsto_{\mathsf{R}} \mathbf{true}$:

$$\ell_1 \mapsto_{\mathsf{L}} \mathbf{true} * \ell_2 \mapsto_{\mathsf{R}} \mathbf{true} \vdash \mathsf{dwp}\ \mathbf{fork}\ \{\ell_1 \leftarrow \mathbf{false}\}; !\,\ell_1\ \&\ \mathbf{fork}\ \{\ell_2 \leftarrow \mathbf{false}\}; !\,\ell_2\ \{v_1\ v_2.\ v_1 = v_2\}$$

It is tempting to use DWP-FORK; but in both the main thread and the forked-off thread we need $\ell_1 \mapsto_{\mathsf{L}} -$ and $\ell_2 \mapsto_{\mathsf{R}} -$ to symbolically execute the dereference and assignment to ℓ_1 and ℓ_2. To share the points-to connectives between both threads, we put them into an Iris-style invariant.

Iris-style invariants are logical propositions denoted as $\boxed{P}^{\mathcal{N}}$, which express that P holds at all times. Unlike in other logics, Iris-style invariants are not attached to locks. Rather, one can

explicitly open an invariant during an atomic step of execution to get access to its contents. To create a new invariant we use the DWP-INV-ALLOC rule, which transfers P into the an invariant $\boxed{P}^{\mathcal{N}}$ with a name space $\mathcal{N} \in \mathit{InvName}$. The transfer of P into an invariant makes it possible to share P between different threads (using INV-DUP). To access an invariant we use the rule DWP-INV. It allow us to *open* an invariant during an atomic symbolic execution step. The *masks* $\mathcal{E} \subseteq \mathit{InvName}$ on dwp are used to keep track of which invariants have been open. This is done to prevent invariant reentrancy.

Returning to our example, we can use DWP-INV-ALLOC to allocate the invariant $I \triangleq \boxed{\exists b \in \mathbb{B}.\, \ell_1 \mapsto_{\mathsf{L}} b * \ell_2 \mapsto_{\mathsf{R}} b}^{\mathcal{N}}$. This invariant not only allows different threads to access ℓ_1 and ℓ_2 (via INV-DUP), but it also ensures that ℓ_1 and ℓ_2 contain the same Boolean value throughout the execution.

The proof then proceeds as follows. We apply DWP-FORK and get two new goals:

1) $I \vdash \mathsf{dwp}\ \ell_1 \leftarrow \textbf{false} \,\&\, \ell_2 \leftarrow \textbf{false}\ \{\mathsf{True}\}$;
2) $I \vdash \mathsf{dwp}\ !\ell_1 \,\&\, !\ell_2\ \{v_1\, v_2.\, v_1 = v_2\}$.

The invariant I can be used for proving both goals (INV-DUP). The first goal involves proving that the assignment of **false** to ℓ_1 and ℓ_2 is secure. We verify this via DWP-INV, and temporarily opening the invariant I to obtain $\ell_1 \mapsto_{\mathsf{L}} b$ and $\ell_2 \mapsto_{\mathsf{R}} b$. We then apply DWP-AWP, and symbolically execute the assignment to obtain $\ell_1 \mapsto_{\mathsf{L}} \textbf{false}$ and $\ell_2 \mapsto_{\mathsf{R}} \textbf{false}$. At the end of this atomic step, we verify that the invariant I still holds.

The second goal is solved in a similar way. When we dereference ℓ_1 and ℓ_2 we know that they contain the same value because of the invariant I. □

C. Soundness

We now state SeLoC's soundness theorem, which guarantees that verified programs are actually secure w.r.t. Definition 3.

As we have described in § III-B, we fix a set $\mathfrak{L}$ of output locations that we assume to be observable by the attacker. We require these locations to always contain the same data in both runs of the program. To reflect this in the logic, we use an invariant that owns the observable locations and forces them to contain the same values in both heaps:

$$I_{\mathfrak{L}} \triangleq \ast_{\ell \in \mathfrak{L}} \boxed{\exists i \in \mathbb{Z}.\, \ell \mapsto_{\mathsf{L}} i * \ell \mapsto_{\mathsf{R}} i}^{\mathcal{N}.(\ell,\ell)}$$

When we verify a program under the invariant $I_{\mathfrak{L}}$, we are forced to interact with the locations in $\mathfrak{L}$ as if they are permanently publicly observable. With this in mind we state the soundness theorem, which we prove in § VII.

Theorem 5 (Soundness)**.** Suppose that:

$$I_{\mathfrak{L}} \vdash \forall \vec{i}, \vec{j} \in \mathbb{Z}.\, \mathsf{dwp}\ e[\vec{i}/\vec{x}] \,\&\, e[\vec{j}/\vec{x}]\ \{v_1\, v_2.\, v_1 = v_2\}$$

is derivable, where $\vec{x}$ are the free variables of e, and $\vec{i}$ and $\vec{j}$ are lists of integers with $|\vec{i}| = |\vec{j}| = |\vec{x}|$, then:

- the expression e is secure, and,
- the configuration $(e[\vec{i}/\vec{x}], \sigma)$ is safe (*i.e.,* cannot get stuck) for any list of integers $\vec{i}$, and any heap σ with $\sigma \sim_{\mathfrak{L}} \sigma$.

D. Protocols

Now that we have seen the basics of Iris-style invariants in SeLoC, let us use the protocol mechanism SeLoC inherits from Iris to verify the example *prog* from Figure 2. We prove the following proposition, which serves as a premise for Theorem 5, and therefore implies the security of *prog*.

Proposition 6. For any integers $i_1, i_2 \in \mathbb{Z}$, we have $I_{\{out\}} \vdash \mathsf{dwp}\ \mathit{prog}\ \mathit{out}\ i_1 \,\&\, \mathit{prog}\ \mathit{out}\ i_2\ \{v_1\, v_2.\, v_1 = v_2 = ((),())\}$.

Proof. We first need a derived rule for parallel composition (which we defined in terms of **fork** in § III-A). The parallel composition operation satisfies a binary version of the standard specification in Concurrent Separation Logic [11]:

$$\text{DWP-PAR}\quad \frac{\mathsf{dwp}\ e_1 \,\&\, s_1\ \{\Psi_1\} \qquad \mathsf{dwp}\ e_2 \,\&\, s_2\ \{\Psi_2\} \qquad (\forall v_1, v_2, w_1, w_2.\, (\Psi_1(v_1, w_1) * \Psi_2(v_2, w_2)) \mathrel{-\!\!*} \Phi((v_1, w_1), (v_2, w_2)))}{\mathsf{dwp}\ (e_1 \mid\mid e_2) \,\&\, (s_1 \mid\mid s_2)\ \{\Phi\}}$$

Second, we need to establish a protocol on the way the values in the record r may evolve. We identify three logical states $\mathsf{State} \triangleq \{\mathsf{Classified}, \mathsf{Intermediate}, \mathsf{Declassified}\}$ the record r can be in; visualized in Figure 4-a:

1) Classified, if the data stored in the record is classified, and $r.\mathit{is\_classified}$ points to **true**;
2) Intermediate, when the data stored in the record is not classified anymore, but $r.\mathit{is\_classified}$ still points to **true**;
3) Declassified, when the data stored in the record is not classified and $r.\mathit{is\_classified}$ points to **false**. This state is final in the sense that once the state of the record becomes Declassified, it forever remains so.

The idea behind the proof is as follows: we use an invariant to track the logical state together with the points-to connectives for the physical state of the record. This way, we ensure that the protocol is followed by both threads.

To model the protocol in SeLoC, we use Iris's mechanism for user-defined ghost state. The exact way this mechanism works is described in [17], [20], but is not important for this paper. What is important is that via this mechanism we can define predicates $\mathsf{in\_state}_\gamma, \mathsf{state\_token}_\gamma : \mathsf{State} \to \mathsf{Prop}$ that satisfy the rules in Figure 4-b. The predicate $\mathsf{in\_state}_\gamma$ will be shared using an invariant, while *thread2* will own the predicate $\mathsf{state\_token}_\gamma$. Rule STATE-AGREE states that the predicates $\mathsf{in\_state}_\gamma$ and $\mathsf{state\_token}_\gamma$ agree on the logical state. If a thread owns both predicates, it can change the logical state using STATE-CHANGE, but only in a way that respects the transition system. Rule DECLASSIFIED-DUP states that once a thread learns that the record is in the final state, *i.e.,* Declassified, this knowledge remains true forever. The predicates are indexed by a *ghost name* γ to allow for different instances of the transition system using STATE-ALLOC. Figure 4-c displays the invariant that ties together the ghost and physical state. It is defined for the records r_1 and r_2 on the left- and the right-hand side, resp. We verify each thread separately with respect to this invariant, which we open every time we access the record.

a) The protocol as a transition system:

b) The rules for ghost state:

STATE-ALLOC
$$\Rrightarrow_{\mathcal{E}} \exists\gamma.\ \mathsf{in\_state}_\gamma(\mathsf{Classified}) * \mathsf{state\_token}_\gamma(\mathsf{Classified})$$

STATE-AGREE
$$\frac{\mathsf{in\_state}_\gamma(s_1) \qquad \mathsf{state\_token}_\gamma(s_2)}{s_1 = s_2}$$

STATE-CHANGE
$$\frac{s_1 \rightarrow s_2 \qquad \mathsf{in\_state}_\gamma(s_1) \qquad \mathsf{state\_token}_\gamma(s_1)}{\Rrightarrow_{\mathcal{E}} \mathsf{in\_state}_\gamma(s_2) * \mathsf{state\_token}_\gamma(s_2)}$$

DECLASSIFIED-DUP
$$\frac{\mathsf{state\_token}_\gamma(\mathsf{Declassified})}{\mathsf{state\_token}_\gamma(\mathsf{Declassified}) * \mathsf{state\_token}_\gamma(\mathsf{Declassified})}$$

c) The invariant:

$$\mathcal{N}\Big(\begin{array}{l}(\mathsf{in\_state}_\gamma(\mathsf{Classified}) * \exists i_1, i_2.\ r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true} * \\ \quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i_1 * r_2.\mathit{data} \mapsto_{\mathsf{R}} i_2) \\ \vee (\mathsf{in\_state}_\gamma(\mathsf{Intermediate}) * \exists i.\ r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true} * \\ \quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i * r_2.\mathit{data} \mapsto_{\mathsf{R}} i) \\ \vee (\mathsf{in\_state}_\gamma(\mathsf{Declassified}) * \exists i.\ r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{false} * \\ \quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{false} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i * r_2.\mathit{data} \mapsto_{\mathsf{R}} i)\end{array}\Big)$$

Figure 4. Value-dependent classification.

Proof of the complete program: We symbolically execute the allocation of the records r_1 and r_2, giving us the resources $r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true}$, $r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true}$, $r_1.\mathit{data} \mapsto_{\mathsf{L}} i_1$, and $r_2.\mathit{data} \mapsto_{\mathsf{R}} i_2$. We then use STATE-ALLOC to obtain $\mathsf{in\_state}_\gamma(\mathsf{Classified})$ and $\mathsf{state\_token}_\gamma(\mathsf{Classified})$. With these resources at hand, we use DWP-INV-ALLOC to establish the invariant in Figure 4-c, which can be shared between both threads. We use DWP-PAR with $\Psi_1(v_1, v_2) \triangleq \Psi_2(v_1, v_2) \triangleq (v_1 = v_2 = ())$, and use the token $\mathsf{state\_token}_\gamma(\mathsf{Classified})$ for the proof of the second thread.

Proof of thread1: We use the symbolic execution rules for dereferencing $r_1.\mathit{is\_classified}$ and $r_2.\mathit{is\_classified}$ until both of them become **false**. At that point, the invariant tells us that we are in the Declassified state. Subsequently, when using the symbolic execution rule for dereferencing $r_1.\mathit{data}$ and $r_2.\mathit{data}$, we use a copy of the predicate $\mathsf{state\_token}_\gamma(\mathsf{Declassified})$ to determine that the last disjunct of the invariant must hold. From that, we know that both $r_1.\mathit{data}$ and $r_2.\mathit{data}$ contain the same value. Using this information we can safely symbolically execute the assignments to the output location out.

Proof of thread2: We start the proof with the initial predicate $\mathsf{state\_token}_\gamma(\mathsf{Classified})$ and update the logical state with each assignment. The complete formalized proof can be found in the Coq mechanization. □

TYPED-IF-LOW
$$\frac{\Gamma \vdash e : \mathtt{bool}^{\mathbf{L}} \qquad \Gamma \vdash t : \tau \qquad \Gamma \vdash u : \tau}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ t\ \textbf{else}\ u : \tau}$$

TYPED-IF-HIGH
$$\frac{\Gamma \vdash e : \mathtt{bool}^{\mathbf{H}} \qquad \Gamma \vdash v : \tau \qquad \Gamma \vdash w : \tau \qquad v,\ w \text{ are values or variables in } \Gamma \qquad \tau \text{ is flat}}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ v\ \textbf{else}\ w : \tau}$$

TYPED-STORE
$$\frac{\Gamma \vdash e : \mathtt{ref}\ \tau \qquad \Gamma \vdash t : \tau}{\Gamma \vdash e \leftarrow t : \mathtt{unit}}$$

TYPED-OUT
$$\frac{\ell \in \mathfrak{L}}{\Gamma \vdash \ell : \mathtt{ref\ int}^{\mathbf{L}}}$$

Figure 5. A selection of the typing rules.

V. Type System and Logical Relations

We show how to define a type system for non-interference as an abstraction on top of SeLoC using the technique of *logical relations*. While logical relations have been used to model type systems and logics for safety and contextual refinement in (variants of) Iris before [21]–[26], we—for the first time—use logical relations in Iris to model a type system for non-interference (§ V-A). We moreover show how we can combine type-checked code with code that has been manually verified using double weakest preconditions in SeLoC (§ V-B).

The types that we consider are as follows:

$$\tau \in \mathsf{Type} ::= \mathtt{unit} \mid \mathtt{int}^{\chi} \mid \mathtt{bool}^{\chi} \mid \tau \times \tau' \mid \mathtt{ref}\ \tau \mid (\tau \to \tau')^{\chi}$$

Here, $\chi, \xi \in \mathsf{Lbl}$ range over the *sensitivity labels* $\{\mathbf{L}, \mathbf{H}\}$ that form a lattice with $\mathbf{L} \sqsubseteq \mathbf{H}$. While any bounded lattice will do, we use the two-element lattice for brevity's sake.

The typing judgment is of the form $\Gamma \vdash e : \tau$, where Γ is an assignment of variables to types, e is an expression, and τ is a type. Some typing rules are given in Figure 5, and the rest can be found in Appendix A. The rule TYPED-OUT shows that every output location $\ell \in \mathfrak{L}$ is typed as a reference to a low-sensitivity integer. By $\tau \sqcup \xi$ we denote the *level stamping*, *e.g.*, $\mathtt{int}^{\chi} \sqcup \xi = \mathtt{int}^{\chi \sqcup \xi}$. See Appendix A for the full definition.

To type check the set data structure from § II, we need to support benign branching on high-sensitivity Booleans. For that purpose we use the rule TYPED-IF-HIGH. In the rule, both branches should either be values or variables, ensuring that they do not perform any computations. In addition, both branches should be of a *flat type*—$\mathtt{int}^{\mathbf{H}}$, $\mathtt{bool}^{\mathbf{H}}$, or a product of two flat types. Function types are not flat because they can leak via timing behavior outside the **if** branch itself.

Notice that our type system has no sensitivity labels on reference types, and no program counter label on the typing judgment. While such labels are common in security type systems for languages with (higher-order) references [1]–[3], [33], a direct adaptation of such type systems is not sound with respect to the termination-sensitive notion of non-interference we consider. A counterexample is provided in Appendix A-B.

$$
\begin{aligned}
[\![\texttt{unit}]\!](v_1, v_2) &\triangleq v_1 = v_2 = () \\
[\![\texttt{int}^{\chi}]\!](v_1, v_2) &\triangleq v_1, v_2 \in \mathbb{Z} * (\chi = \mathbf{L} \to v_1 = v_2) \\
[\![\texttt{bool}^{\chi}]\!](v_1, v_2) &\triangleq v_1, v_2 \in \mathbb{B} * (\chi = \mathbf{L} \to v_1 = v_2) \\
[\![\tau \times \tau']\!](v_1, v_2) &\triangleq \exists w_1, w_2, w_1', w_2'. \\
&\qquad v_1 = (w_1, w_1') * v_2 = (w_2, w_2') * \\
&\qquad [\![\tau]\!](w_1, w_2) * [\![\tau']\!](w_1', w_2') \\
[\![\texttt{ref}\ \tau]\!](v_1, v_2) &\triangleq v_1, v_2 \in \mathsf{Loc} * \\
&\qquad \boxed{\exists w_1\, w_2.\, v_1 \mapsto_{\mathsf{L}} w_1 * \\
&\qquad\quad v_2 \mapsto_{\mathsf{R}} w_2 * [\![\tau]\!](w_1, w_2)}^{\mathcal{N}.(v_1,v_2)} \\
[\![(\tau \to \tau')^{\chi}]\!](v_1, v_2) &\triangleq \Box\big(\forall w_1, w_2.\, [\![\tau]\!](w_1, w_2) \mathrel{-\!\!*} \\
&\qquad [\![\tau' \sqcup \chi]\!]^{\mathsf{e}}(v_1\ w_1)(v_2\ w_2)\big) \\
[\![\tau]\!]^{\mathsf{e}}(e_1, e_2) &\triangleq \mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\tau]\!]\}
\end{aligned}
$$

Figure 6. The logical relations interpretation of types.

LOGREL-IF-LOW

$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\texttt{bool}^{\mathbf{L}}]\!]\} \quad \mathsf{dwp}\ t_1\ \&\ t_2\ \{\Phi\} \quad \mathsf{dwp}\ u_1\ \&\ u_2\ \{\Phi\}}{\mathsf{dwp}\ \textbf{if}\ e_1\ \textbf{then}\ t_1\ \textbf{else}\ u_1\ \&\ \textbf{if}\ e_2\ \textbf{then}\ t_2\ \textbf{else}\ u_2\ \{\Phi\}}$$

LOGREL-STORE

$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\texttt{ref}\ \tau]\!]\} \quad \mathsf{dwp}\ t_1\ \&\ t_2\ \{[\![\tau]\!]\}}{\mathsf{dwp}\ (e_1 \leftarrow t_1)\ \&\ (e_2 \leftarrow t_2)\ \{[\![\texttt{unit}]\!]\}}$$

Figure 7. A selection of compatibility rules.

A. Logical relations model

We give a semantic model of our type system using logical relations. The key idea of logical relations is to interpret each type τ as a relation on values, *i.e.,* to each type τ we assign an *interpretation* $[\![\tau]\!] : \mathsf{Val} \times \mathsf{Val} \to \mathsf{Prop}$ where Prop is the type of SeLoC propositions. Intuitively, $[\![\tau]\!](v_1, v_2)$ expresses that v_1 and v_2 of type τ are indistinguishable by a low-sensitivity attacker. The definition of $[\![\tau]\!]$ is given in Figure 6. We will now explain some interesting cases in detail.

The interpretation $[\![\texttt{int}^{\mathbf{L}}]\!]$ contains the pairs of equal integers, while $[\![\texttt{int}^{\mathbf{H}}]\!]$ contains the pairs of any two integers. This captures the intuition that a low-sensitivity attacker can observe low-sensitivity integers, but not high-sensitivity integers.

The interpretation $[\![\texttt{ref}\ \tau]\!]$ captures that references ℓ_1 and ℓ_2 are indistinguishable iff they always hold values w_1 and w_2 that are indistinguishable at type τ. This is formalized by imposing an invariant that contains both points-to propositions $\ell_1 \mapsto_{\mathsf{L}} w_1$ and $\ell_2 \mapsto_{\mathsf{R}} w_2$, as well as the interpretation of τ that links the values w_1 and w_2. Notice that our interpretation of references does not require the locations ℓ_1 and ℓ_2 themselves to be syntactically equal. This is crucial for modeling dynamic allocation (recall that the allocation oracle described in § III-A may depend on the contents of the heap).

The interpretation $[\![(\tau \to \tau')^{\chi}]\!]$ captures that functions v_1 and v_2 are indistinguishable iff for all inputs w_1 and w_2 indistinguishable at type τ, the behaviors of the expressions $v_1\ w_1$ and $v_2\ w_2$ are indistinguishable at type $\tau' \sqcup \chi$. To formalize what it means for the behavior of expressions (in this case $v_1\ w_1$ and $v_2\ w_2$) to be indistinguishable, we define the *expression interpretation* $[\![\tau]\!]^{\mathsf{e}} : \mathsf{Expr} \times \mathsf{Expr} \to \mathsf{Prop}$ by lifting the value interpretation using double weakest preconditions.

The interpretation of functions is defined using the *persistence modality* $\Box$ of Iris [20, Section 2.3]. Intuitively, $\Box P$ states that P holds without asserting ownership of any non-shareable resources. Having the persistence modality in this definition is common in logical relations in Iris [21]—it ensures that indistinguishable functions remain indistinguishable forever.

The interpretation of expressions $[\![\_]\!]^{\mathsf{e}}$ generalizes to open terms by considering all well-typed substitutions. A (binary) substitution γ is a function $\mathsf{Var} \to \mathsf{Val} \times \mathsf{Val}$. We write $\gamma_i(e)$ for a term e where each free variable x is substituted by $\pi_i(\gamma(x))$. A substitution γ is well-typed, notation $[\![\Gamma]\!](\gamma)$, iff $\forall x.\, [\![\tau]\!](\gamma(\Gamma(x)))$. We define the *semantic typing judgment* as:

$$\Gamma \models e : \tau \triangleq \forall \gamma.\, ([\![\Gamma]\!](\gamma) * I_{\mathfrak{L}}) \mathrel{-\!\!*} [\![\tau]\!]^{\mathsf{e}}(\gamma_1(e), \gamma_2(e)).$$

Here, $I_{\mathfrak{L}}$ is the invariant on the observable locations (§ IV-C).

Theorem 7 (Soundness)**.** If $x_1 : \texttt{int}^{\mathbf{H}}, \ldots, x_n : \texttt{int}^{\mathbf{H}} \models e : \texttt{int}^{\mathbf{L}}$ is a derivable in SeLoC, then e is secure, and the configuration $(e[\vec{i}/\vec{x}], \sigma)$ is safe (*i.e.,* cannot get stuck) for any list of integers $\vec{i}$, and any heap σ with $\sigma \sim_{\mathfrak{L}} \sigma$.

Proof. This a direct consequence of Theorem 5. □

The *fundamental property* of logical relations states that any program that can be type checked is semantically typed.

Proposition 8 (Fundamental property)**.** If $\Gamma \vdash e : \tau$, then $\Gamma \models e : \tau$ is derivable in SeLoC.

Proof. This proposition is proved by induction on the typing judgment $\Gamma \vdash e : \tau$ using so-called *compatibility rules* for each case. A selection of these rules is shown in Figure 7. □

B. Typing via manual proof

When composing the fundamental property (Proposition 8) and the soundness theorem (Theorem 7) we obtain that any typed program is secure. For instance, it allows us to show that the *rand* program is secure by type checking it, instead of performing a manual proof as done in Proposition 4.

However, semantic typing gives us more—it allows us to combine type-checked code with manually verified code. Let us consider the examples from § II, which are not typed according to the typing rules, but which we can *prove* to be semantically typed by dropping down to the interpretation of the semantic typing judgment in terms of double weakest preconditions.

Proposition 9. $\models \mathit{prog} : \texttt{ref int}^{\mathbf{L}} \to \texttt{int}^{\mathbf{H}} \to \texttt{unit} \times \texttt{unit}$.

Proof. This is a direct consequence of Proposition 6. □

Proposition 10. $\models \mathit{awk} : \texttt{int}^{\mathbf{H}} \to (\texttt{unit} \to \texttt{unit})^{\mathbf{L}} \to \texttt{int}^{\mathbf{L}}$.

Proof. The proposition boils down to showing that for any $i_1, i_2 \in \mathbb{Z}$ and f_1, f_2 with $[\![(\texttt{unit} \to \texttt{unit})^{\mathbf{L}}]\!](f_1, f_2)$, we have

$\mathsf{dwp}\ awk\ i_1\ f_1\ \&\ awk\ i_2\ f_2\ \{v_1\ v_2.\ v_1 = v_2 = 0\}$. We verify this by establishing a monotone protocol similar to the one used in the proof of value-dependent classification in § IV-B. The full proof can be found in the Coq mechanization. □

After establishing the semantic typing for, *e.g.,* *prog* we can use it in any context where a function of the type $\texttt{ref int}^{\mathbf{L}} \to \texttt{int}^{\mathbf{H}} \to \texttt{unit} \times \texttt{unit}$ is expected. For example:

$$h : \texttt{int}^{\mathbf{H}}, f : \texttt{ref int}^{\mathbf{L}} \to \texttt{int}^{\mathbf{H}} \to \texttt{unit} \times \texttt{unit} \vdash \textbf{let}\ x = \textbf{ref}(0)\ \textbf{in}\ \ \textbf{fork}\ \{f\ x\ h\}\,; !\,x : \texttt{int}^{\mathbf{L}}$$

Using the fundamental property (Proposition 8) we obtain a semantic typing judgment for the above program. Using Proposition 9 we establish that if we substitute *prog* for f, the resulting program will still be semantically typed, and thus secure by the soundness theorem (Theorem 7).

The same methodology can be used to assign the types to the safe array operations from § II-B via manual proof, and compose them with the type checked set data structure from § II-A. The proof can be found in the Coq mechanization.

VI. Modular separation logic specifications

Types provide a convenient way to specify program modules, but are not always strong enough to enable the verification of sophisticated clients. This is particularly relevant if the specification of a program module is to be used in a manual proof or relies on function correctness. We show that in addition to specifications through types, SeLoC can also be used to prove modular specification in separation logic. We demonstrate this approach on dynamically created locks (§ VI-A) and dynamically classified references (§ VI-B).

A. Locks

The HeapLang language we consider does not provide locks as primitive constructs, but provides the low-level compare-and-set (**CAS**) operation with which different locking mechanisms can be implemented. Figure 8 displays the implementation and specification of a spin lock. The specification makes use of a relational generalization of the common *lock predicates* in separation logic [34]–[36]. The predicate $\mathsf{isLock}(lk_1, lk_2, R)$ expresses that the pair of locks lk_1 and lk_2 protect the resources R, and the predicate $\mathsf{locked}(lk_1, lk_2)$ expresses that the pair of locks is in acquired state.

To verify that the spin lock implementation conforms to the lock specification, we define the lock predicates using Iris's mechanism for invariants and user-defined ghost state. The proof (and invariant) are generalizations of the ordinary proof (and invariant) for functional correctness in Iris.

The rules of our lock specification are similar to the rules in logics with locks as primitives constructs, such as [6], [10]. There are two notable exceptions. First, in *loc. cit.* one needs to fix the set of locks and associated resources upfront, whereas in SeLoC one can create locks dynamically and attach an arbitrary resource R to each lock during the proof. Second, since locks are not primitive constructs in SeLoC, the specification also applies to different lock implementations, *e.g.,* a ticket lock, as we have shown in the Coq mechanization.

a) Implementation of a spin lock:

$$\begin{aligned} &\textbf{let}\ new\_lock\ () = \textbf{ref}(\textbf{false}) \\ &\textbf{let rec}\ acquire\ lk = \textbf{if CAS}(lk, \textbf{false}, \textbf{true})\ \textbf{then}\ () \\ &\qquad\qquad\qquad\qquad \textbf{else}\ acquire\ lk \\ &\textbf{let}\ release\ lk = lk \leftarrow \textbf{false} \end{aligned}$$

b) Modular separation logic specification of locks:

NEWLOCK-SPEC

$$\frac{R}{\mathsf{dwp}\ new\_lock\ ()\ \&\ new\_lock\ ()\ \{lk_1\ lk_2.\ \mathsf{isLock}(lk_1, lk_2, R)\}}$$

ISLOCK-DUP

$$\frac{\mathsf{isLock}(lk_1, lk_2, R)}{\mathsf{isLock}(lk_1, lk_2, R) * \mathsf{isLock}(lk_1, lk_2, R)}$$

ACQUIRE-SPEC

$$\frac{\mathsf{isLock}(lk_1, lk_2, R)}{\mathsf{dwp}\ acquire\ lk_1\ \&\ acquire\ lk_2\ \{R * \mathsf{locked}(lk_1, lk_2)\}}$$

RELEASE-SPEC

$$\frac{\mathsf{isLock}(lk_1, lk_2, R) \qquad R \qquad \mathsf{locked}(lk_1, lk_2)}{\mathsf{dwp}\ release\ lk_1\ \&\ release\ lk_2\ \{\mathsf{True}\}}$$

Figure 8. Dynamically allocated locks in SeLoC.

B. Dynamically classified references

We consider a program module that encapsulates and generalizes dynamically classified references[7] as used in § II-C. This program module generalizes to clients with multiple threads and different sharing models. For example, clients in which multiple threads read and write to the dynamically classified reference, or in which the data gets classified again. The Coq mechanization contains such an example. The implementation and specification[8] of the module for dynamically classified references is shown in Figure 9.

The main ingredient of the specification is the representation predicate $\mathsf{val\_dep}(\tau, r_1, r_2)$, which expresses that the dynamically classified references r_1 and r_2 contain related data of type τ at all times. Since $\mathsf{val\_dep}(\tau, r_1, r_2)$ expresses mere knowledge instead of ownership, it is duplicable (VALDEP-DUP). With the representation predicate at hand we can formulate weak specifications for some operations. For instance, the rule READ-SAFE over-approximates the sensitivity-level of the values returned by the *read* operation, and dually, the rule STORE-SAFE under-approximates the sensitivity-level of the values stored using the *store* operation. Of course, at times we want to track the precise sensitivity-level. For that we use a *fractional token* $\mathsf{class}_{(r_1,r_2)}(\chi, q)$ with $q \in (0, 1]_{\mathbb{Q}}$. This token is reminiscent of fractional permissions in separation logic. The proof rules

[7] In this context declassification refers to changing the dynamic classification of the reference. It is thus unrelated to static declassification policies [37], and the *declassify* function is unrelated to the eponymous function from [38].

[8] The specification in Figure 9 is derived from a more general HOCAP-style logically atomic specifications [39], which can be found in Appendix B and the Coq mechanization.

a) Implementation of dynamically classified references:

$$\textbf{let}\ new\_vdep\ v = \left\{\begin{array}{l} data = \textbf{ref}(v); \\ is\_classified = \textbf{ref}(\textbf{false}) \end{array}\right\} \qquad \textbf{let}\ classify\ r = r.is\_classified \leftarrow \textbf{true}$$

$$\textbf{let}\ read\ r = !\,r.data \qquad \textbf{let}\ declassify\ r\ v = r.data \leftarrow v; r.is\_classified \leftarrow \textbf{false}$$

$$\textbf{let}\ store\ r\ v = r.data \leftarrow v \qquad \textbf{let}\ get\_classified\ r = !\,r.is\_classified$$

b) Modular separation logic specification of dynamically classified references:

NEW-VDEP

$$\frac{}{\mathsf{dwp}\ new\_vdep\ v_1\ \&\ new\_vdep\ v_2\ \{r_1\ r_2.\ \mathsf{val\_dep}(\tau, r_1, r_2) * \mathsf{class}_{(r_1,r_2)}(\chi, 1)\}} \quad [\![\tau \sqcup \chi]\!](v_1, v_2)$$

VALDEP-DUP

$$\mathsf{val\_dep}(\tau, r_1, r_2) \vdash \mathsf{val\_dep}(\tau, r_1, r_2) * \mathsf{val\_dep}(\tau, r_1, r_2)$$

CLASS-SPLIT

$$\mathsf{class}_{(r_1,r_2)}(\chi, q_1) * \mathsf{class}_{(r_1,r_2)}(\chi, q_2) \dashv\vdash \mathsf{class}_{(r_1,r_2)}(\chi, q_1 + q_2)$$

READ-SAFE

$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2)}{\mathsf{dwp}\ read\ r_1\ \&\ read\ r_2\ \{v_1\ v_2.\ [\![\tau \sqcup \mathbf{H}]\!](v_1, v_2)\}}$$

READ-SEQ

$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q)}{\mathsf{dwp}\ read\ r_1\ \&\ read\ r_2\ \{v_1\ v_2.\ [\![\tau \sqcup \chi]\!](v_1, v_2) * \mathsf{class}_{(r_1,r_2)}(\chi, q)\}}$$

STORE-SAFE

$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad [\![\tau]\!](v_1, v_2)}{\mathsf{dwp}\ store\ r_1\ v_1\ \&\ store\ r_2\ v_2\ \{\mathsf{True}\}}$$

STORE-SEQ

$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q) \qquad [\![\tau \sqcup \chi]\!](v_1, v_2)}{\mathsf{dwp}\ store\ r_1\ v_1\ \&\ store\ r_2\ v_2\ \{\mathsf{class}_{(r_1,r_2)}(\chi, q)\}}$$

CLASSIFY-SEQ

$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, 1)}{\mathsf{dwp}\ classify\ r_1\ \&\ classify\ r_2\ \{\mathsf{class}_{(r_1,r_2)}(\mathbf{H}, 1)\}}$$

DECLASSIFY-SEQ

$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, 1) \qquad [\![\tau]\!](v_1, v_2)}{\mathsf{dwp}\ declassify\ r_1\ v_1\ \&\ declassify\ r_2\ v_2\ \{\mathsf{class}_{(r_1,r_2)}(\mathbf{L}, 1)\}}$$

GET-CLASSIFIED-SEQ

$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q)}{\mathsf{dwp}\ get\_classified\ r_1\ \&\ get\_classified\ r_2\ \{b_1\ b_2.\ (b_1 = b_2) * \mathsf{class}_{(r_1,r_2)}(\chi, q) * ((b_1 = \textbf{false}) \to (\chi = \mathbf{L}))\}}$$

c) The transition system used for the proof:

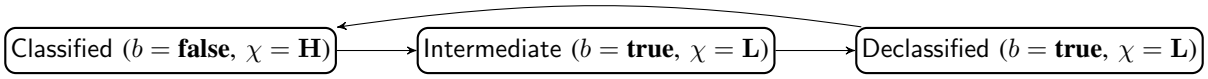

Figure 9. Dynamically classified references in SeLoC.

for *declassify* and *classify* (DECLASSIFY-SEQ and CLASSIFY-SEQ) require the full fraction ($q = 1$) since they change the classification. The precise rules for *read* and *store* (READ-SEQ and STORE-SEQ) do not change the classification, and thus require an arbitrary fraction. The token is splittable according to CLASS-SPLIT so it can be shared between threads.

Since the rules for *declassify* and *classify* require a full fraction ($q = 1$), they do not allow for fine-grained sharing[9], *i.e.,* they cannot be used to verify a program that runs *declassify* in parallel with *classify*. It is good that this is impossible—running these operations in parallel results in a race-condition, making it impossible to know what the final classification would be. However, it *is* possible to verify a program that runs *declassify* in parallel with *read* or *store* (using precise rules for these two operations) by sharing the token via an invariant. To access such a shared token one has to use the more general HOCAP-style logically atomic specifications found in Appendix B and the Coq mechanization.

Proof: In order to verify the implementation, we follow the usual approach of defining the representation predicate $\mathsf{val\_dep}(\tau, r_1, r_2)$ and token $\mathsf{class}_{(r_1,r_2)}(\chi, q)$ using Iris's invariant and protocol mechanism. The invariant expresses that, at all times, the fields *is_classified* of both records contain the same Boolean value b, and that the data in the records are related by $[\![\tau \sqcup \chi]\!]$. The relation between the Boolean values b and the security label χ, and the way it evolves, are expressed using a protocol visualized as the transition system in Figure 9.

VII. Soundness

To prove soundness of SeLoC (Theorem 7), we give a model of double weakest preconditions in Iris (§ VII-A), and then construct a bisimulation out of this model (§ VII-B). We only give a high-level overview, the details are in Appendix C.

[9]We can still achieve sharing by storing the token $\mathsf{class}_{(r_1,r_2)}(\chi, 1)$ in a lock, as outlined in § VI-A.

A. Model of double weakest preconditions

The model of the Iris logic [19], [20] consists of three layers:

- The Iris base logic, which contains the standard separation logic connectives (*e.g.,* $*$ and $-\!\!*$), modalities (*e.g.,* $\triangleright$, $\Box$), and the machinery for user-defined ghost state.
- The invariant mechanism, which is built as a library on top of the Iris base logic.
- The Iris program logic, which is built as a library on top of the Iris base logic and invariant mechanism. It provides weakest preconditions for proving safety and functional correctness of concurrent programs.

We reuse the first two layers of Iris (the base logic and the invariant mechanism), on top of which we model our new notion of double weakest preconditions. This model is inspired by the model of ordinary (unary) weakest preconditions in Iris and the *product program* construction [40]. Intuitively, $\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}$ captures that the expressions e_1 and e_2 are executed in lock-step. This is done by case analysis:

- Either, both expressions e_1 and e_2 are values that are related by the postcondition Φ.
- Otherwise, both expressions e_1 and e_2 are reducible, and for any reductions $(e_1, \sigma_1) \to_{\mathsf{t}} (e_1', \sigma_1')$ and $(e_2, \sigma_2) \to_{\mathsf{t}} (e_2', \sigma_2')$, the expressions e_1' and e_2' are still related by dwp. If e_1 and e_2 fork off threads $\vec{e_1'}$ and $\vec{e_2'}$, then all of the forked-off threads are related pairwise by dwp.

B. Constructing a bisimulation

The main challenge of constructing a strong low-bisimulation lies in connecting double weakest preconditions, at the level of separation logic, with strong-low bisimulations, at the meta level. The construction is done as follows:

1) We define a relation $\mathcal{R}$ that "lifts" double weakest preconditions out of the SeLoC logic into the meta-level.
2) We then show that the relation $\mathcal{R}$ satisfies a number of bisimulation-like properties.
3) The relation $\mathcal{R}$ is not a bisimulation because it is not transitive. To fix this, we take its transitive closure $\mathcal{R}^*$.

Finally, we show that the dwp predicate is sound w.r.t. the relation $\mathcal{R}$: if $I_{\mathfrak{L}} \vdash \mathsf{dwp}\ e\ \&\ s\ \{v_1\ v_2.\ v_1 = v_2\}$ can be derived in SeLoC, then $(e, \sigma_1)\ \mathcal{R}\ (s, \sigma_2)$ for $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$.

VIII. Mechanization in Coq

We have mechanized the definition of SeLoC, the type system, the soundness proof, and all examples and derived constructions in the paper and the appendix in Coq. The mechanization has been built on top of the mechanization of Iris [18]–[20], which readily provides the Iris base logic, the invariant mechanism, and the HeapLang language.

To carry out the mechanization effectively, we have made extensive use of the tactic language MoSeL (formerly Iris Proof Mode) for separation logic in Coq [21], [41]. Using MoSeL we were able to carry out in Coq the typical kind of reasoning steps one would do on paper. This was essential to mechanize the SeLoC logic (1818 line of Coq code), the type system (1355 lines), and all the examples (3223 lines).

IX. Related work

A. Security based on strong low-bisimulations

The security condition we use, a strong low-bisimulation due to Sabelfeld and Sands [5], has been studied in a variety of related work. In *loc. cit.* the notion of a strong low-bisimulation is applied to a first-order stateful language with concurrency. It is also shown that this notion implies a scheduler-independent bisimulation known as ρ-specific probabilistic bisimulation. Sabelfeld and Sands presented both strong low-bisimulation on thread pools and configurations. We use the bisimulation relation on configurations because it allows for a flow-sensitive analysis and readily supports dynamic allocation.

Strong low-bisimulations are highly compositional: if a thread e is secure w.r.t. a strong low-bisimulation, then the composition of e with *any other* thread is secure. Unfortunately, this property makes it non-trivial to adapt strong low-bisimulations for analyses that are flow-sensitive in thread composition. We work around this issue by composing the components at the level of the logic (as double weakest preconditions), and not at the level of the bisimulations, despite the fact that we use strong low-bisimulations as an auxiliary notion in our soundness proof. By performing the composition at the level of the logic, we can use Iris invariants and modular specifications to put restrictions onto which threads can be composed.

Another way of enabling flow-sensitive analysis was developed by Mantel *et al.* [4], who relaxed the notion of a strong low-bisimulation to a *strong low-bisimulation modulo modes*. Their approach enables rely-guarantee style reasoning at the level of the bisimulation. Notably, using the notion of strong low-bisimulations modulo modes one can specify that no other threads can read or write to a certain location.

Based on the notion of strong low-bisimulations modulo modes, the Covern project [6], [7], [42] developed a series of logics for rely/guarantee reasoning. Notably, Murray *et al.* [6] presented the first fully mechanized program logic for non-interference of concurrent programs with shared memory, which is also called Covern. While Covern is not a separation logic, it has been extended to allow for flexible reasoning about non-interference in presence of value-dependent classifications [7]. In terms of the object language, Covern does not support fine-grained concurrency, arrays, or dynamically allocated references. Since Covern does not support fine-grained concurrency, locks are modeled as primitives in the language and logic, while they are derived constructs in our work. As a result of that, Covern's notion of strong low-bisimulations is tied to the operational semantics of locks, *i.e.,* it is considered *modulo* the variables that are held by locks. The set of locks, and the variables they protect, has to be provided statically. Hence their approach does not immediately generalize to support dynamically allocated locks, nor to reason about locks that protect other resources than permissions to write to or read from variables. Value-dependent classifications are also primitive in Covern [7], while they are derived constructs in our work. Covern has two separate primitive rules for assignment to

"normal" variables and for assignment to "control" variables (*i.e.,* variables that signify the classification levels).

B. Program logics for non-interference

Early work by Beringer and Hofmann [43] established a connection between Hoare logic and non-interference. They did so for a first-order sequential language with a simple non-interference condition. Non-interference was encoded through self-composition and renaming, making sure that both parts of the composed program operate on different parts of the heap (something that one gets by construction in separation logic). Notably, they proved the non-interference property of two type systems by constructing models of the type systems in their Hoare logic. They also showed how to extend their approach to object-oriented type systems.

C. Separation logics for non-interference

Karbyshev *et al.* [9] devised a compositional type-and-effect system based on separation logic to prove non-interference of concurrent programs with channels. Their system is sound w.r.t. termination-insensitive non-interference allowing for races on low-sensitivity locations. They consider security for arbitrary (deterministic) schedulers, and allow for a *rescheduling* operation in the programming language to prevent scheduler tainting. To achieve that, their logical rule for rescheduling treats the scheduler as a splittable separation logic resource, allowing one to share it between threads. In terms of the object language, they consider a first-order language without dynamic memory allocation, and the concurrency primitives are based on channels with send and receive operations rather than our low-level fine-grained concurrency model. They do not provide a logic for modular reasoning about program modules.

The recently proposed separation logic SecCSL [10] enables reasoning about value-dependent information flow control policies through a relational interpretation of separation logic. One of the main advantages of the SecCSL approach is its amenability to automation. However, to achieve that, they restrict to a first-order separation logic with restricted language features, *i.e.,* a first-order language with first-order references, and a coarse-grained synchronization mechanism. SecCSL does not support dynamically allocated references out of the box. However, we believe that it can be extended to support dynamic allocation, as long as the semantics for allocation are deterministic and do not depend on the global heap.

The security condition in SecCSL [10] is non-standard, and is geared to providing meaning to the intermediate Hoare triples. Because of that, their formulation of non-interference is closely intertwined with the semantics of the logic.

Costanzo and Shao [44] devised a separation logic for proving non-interference of first-order sequential programs. One of the novelties of their system is the support for declassification in the form of *delimited release* [38]. While we do not study declassification policies in this paper, we believe that the approach of Costanzo and Shao can be adapted to our setting, provided that we are willing to relax the notion of a strong low-bisimulation.

D. Type systems for non-interference

As discussed in the introduction (§ I), a lot of work on non-interference in the programming languages area has focused on type-system based approaches. Such approaches are amendable to high degrees of automation, but lack the ability to reason about functional correctness. Due to an abundance of prior work on in this area, we restrict to directly related work.

Pottier and Simonet developed Flow Caml [1], a type system for termination-insensitive non-interference for sequential higher-order language in the spirit of Caml. Soundness w.r.t. non-interference is proven with the *product programs* technique. This kind of self-composition was an inspiration for our model of double weakest preconditions, although we avoid self-composition of programs at the syntactic level.

Terauchi [33] devised a capabilities-based type system for *observational determinism* [45]. Observational determinism is a formulation of non-interference for concurrent programs that is substantially different from the strong low-bisimulation considered in this paper. In particular, under observational determinism, no races on low-sensitivity locations are allowed, ruling out *e.g.,* the $rand$ function from § III-C.

E. Logical relation models

The technique of logical relations is widely used for proving the soundness of type systems and logics. The work on step-indexing [46], [47] made it possible to scale logical relations to languages with higher-order references and recursive types. Notably, Rajani and Garg [2] describe a step-indexed Kripke-style model for two information flow aware type systems for a sequential language with higher-order references. While they do not consider concurrency and their notion of non-interference is different from ours (their notion is termination- and progress-insensitive), their model is similar in spirit. However, we make use of the "logical" approach to step-indexing [48] in Iris to avoid explicit step-indexes in definitions and proofs.

The relational model of our type system is directly inspired by a line of work on interpretation of type systems and logical relations in Iris [21]–[26], but this previous work focused on reasoning about safety and contextual equivalence of programs, while we target non-interference. For that purpose we developed double weakest preconditions.

The idea of using logical relations to reason about the combination of typed and manually verified code has been used before in the context of Iris. Jung *et al.* [25], [26] use it to reason about unsafe code in Rust, and Krogh-Jespersen *et al.* [22] use it in the context of type-and-effect systems.

X. Conclusions and Future Work

We have presented SeLoC—the first separation logic for non-interference that combines type checking and manual proof. It supports fine-grained concurrency, higher-order functions, and dynamic (higher-order) references. The key feature of SeLoC is its novel connective for double weakest preconditions, which in combination with Iris-style invariants, allows for compositional reasoning. We have proved soundness of SeLoC with respect to a standard notion of security.

In future work we want to develop a more expressive type system. To develop such a type system, we want to transfer back reasoning principles from SeLoC into constructs that can be type checked automatically. Moreover, we would like to study declassification in the sense of delimited information release and static declassification policies [37], [38], [44], [49].

Acknowledgments

We thank the anonymous reviewers and Toby Murray for their comments, and Aslan Askarov and Alexandr Karbyshev for helpful discussions.

Dan Frumin was supported by the Netherlands Organisation for Scientific Research (NWO), STW project 14319. Robbert Krebbers was supported by the Netherlands Organisation for Scientific Research (NWO), project 016.Veni.192.259. Lars Birkedal was supported by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation and by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

[1] F. Pottier and V. Simonet, "Information flow inference for ML," *TOPLAS*, vol. 25, no. 1, pp. 117–158, 2003.

[2] V. Rajani and D. Garg, "Types for information flow control: Labeling granularity and semantic models," in *CSF*, 2018, pp. 233–246.

[3] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Cornell University, 2002.

[4] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *CSF*, 2011, pp. 218–232.

[5] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *FCS*, 2000, pp. 200–214.

[6] T. Murray, R. Sison, and K. Engelhardt, "Covern: A logic for compositional verification of information flow control," in *EuroS&P*, 4 2018, pp. 16–30.

[7] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *CSF*, 6 2016, pp. 417–431.

[8] D. Schoepe, T. Murray, and A. Sabelfeld, "Veronica: Verified concurrent information flow security unleashed," 2019, in submission.

[9] A. Karbyshev, K. Svendsen, A. Askarov, and L. Birkedal, "Compositional non-interference for concurrent programs via separation and framing," in *POST*, 2018, pp. 53–78.

[10] G. Ernst and T. Murray, "SecCSL: Security concurrent separation logic," in *CAV*, 2019, pp. 208–230.

[11] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *TCS*, vol. 375, no. 1–3, pp. 271–307, 2007.

[12] S. Brookes, "A semantics for concurrent separation logic," *TCS*, vol. 375, no. 1–3, pp. 227–270, 2007.

[13] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, no. 2-3, pp. 67–84, 2007.

[14] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *S&P*, 2011, pp. 165–179.

[15] L. Lourenço and L. Caires, "Dependent information flow types," in *POPL*, 2015, pp. 317–328.

[16] S. Gregersen, S. E. Thomsen, and A. Askarov, "A dependently typed library for static information-flow control in Idris," in *POST*, ser. LNCS, vol. 11426, 2019, pp. 51–75.

[17] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *POPL*, 2015, pp. 637–650.

[18] R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer, "Higher-order ghost state," in *ICFP*, 2016, pp. 256–269.

[19] R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal, "The essence of higher-order concurrent separation logic," in *ESOP*, ser. LNCS, vol. 10201, 2017, pp. 696–723.

[20] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *JFP*, vol. 28, p. e20, 2018.

[21] R. Krebbers, A. Timany, and L. Birkedal, "Interactive proofs in higher-order concurrent separation logic," in *POPL*, 2017, pp. 205–217.

[22] M. Krogh-Jespersen, K. Svendsen, and L. Birkedal, "A relational model of types-and-effects in higher-order concurrent separation logic," in *POPL*, 2017, pp. 218–231.

[23] A. Timany, L. Stefanesco, M. Krogh-Jespersen, and L. Birkedal, "A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST," *PACMPL*, vol. 2, no. POPL, pp. 64:1–64:28, 2018.

[24] D. Frumin, R. Krebbers, and L. Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency," in *LICS*, 2018, pp. 442–451.

[25] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *PACMLPL*, vol. 2, no. POPL, pp. 66:1–66:34, 2018.

[26] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in Rust: The promise and the challenge." 2020, to appear in CACM.

[27] D. Frumin, R. Krebbers, and L. Birkedal, "Coq mechanization of SeLoC," 2020, available online at https://cs.ru.nl/~dfrumin/arch/seloc-0.2.tgz.

[28] J. Agat and D. Sands, "On confidentiality and algorithms," in *S&P*, 2001, pp. 64–77.

[29] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The definition of standard ML: revised*. MIT press, 1997.

[30] A. Pitts and I. Stark, "Operational reasoning for functions with local state," in *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute, 1998, pp. 227–273.

[31] Iris project, "A higher-order concurrent separation logic framework implemented and verified in the proof assistant Coq," 2019. [Online]. Available: https://iris-project.org/

[32] L. Birkedal and A. Bizjak, "Lecture notes on Iris: Higher-order concurrent separation logic," 2019. [Online]. Available: https://iris-project.org/tutorial-material.html

[33] T. Terauchi, "A type system for observational determinism," in *CSF*, 2008, pp. 287–300.

[34] B. Biering, L. Birkedal, and N. Torp-Smith, "BI-hyperdoctrines, higher-order separation logic, and abstraction," *TOPLAS*, vol. 29, no. 5, 2007.

[35] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *ECOOP*, ser. LNCS, vol. 6183, 2010, pp. 504–528.

[36] K. Svendsen and L. Birkedal, "Impredicative concurrent abstract predicates," in *ESOP*, ser. LNCS, vol. 8410, 2014, pp. 149–168.

[37] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *JCS*, vol. 17, no. 5, pp. 517–548, 2009.

[38] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *ISSS*, 2004, pp. 174–191.

[39] K. Svendsen, L. Birkedal, and M. Parkinson, "Modular reasoning about separation of concurrent data structures," in *ESOP*, ser. LNCS, vol. 7792, 2013, pp. 169–188.

[40] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *FM*, ser. LNCS, vol. 6664, 2011, pp. 200–214.

[41] R. Krebbers, J. Jourdan, R. Jung, J. Tassarotti, J. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer, "MoSeL: A general, extensible modal framework for interactive proofs in separation logic," *PACMPL*, vol. 2, no. ICFP, pp. 77:1–77:30, 2018.

[42] R. Sison and T. Murray, "Verifying that a compiler preserves concurrent value-dependent information-flow security," 2019, in submission.

[43] L. Beringer and M. Hofmann, "Secure information flow and program logics," in *CSF*, 7 2007, pp. 233–248.

[44] D. Costanzo and Z. Shao, "A separation logic for enforcing declarative information flow control policies," in *POST*, ser. LNCS, vol. 8414, 2014, pp. 179–198.

[45] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *CSFW*, 2003, pp. 29–43.

[46] A. Ahmed, "Step-indexed syntactic logical relations for recursive and quantified types," in *ESOP*, ser. LNCS, vol. 3924, 2006, pp. 69–83.

[47] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon, "A very modal model of a modern, major, general type system," in *POPL*, 2007, pp. 109–122.

[48] D. Dreyer, A. Ahmed, and L. Birkedal, "Logical step-indexed logical relations," in *LICS*, 2009, pp. 71–80.

[49] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Towards a logical account of declassification," in *PLAS*, 2007, pp. 61–66.

APPENDIX A
TYPE SYSTEM

A. Typing rules

The subtyping and typing rules can be found in Figure 10; the compatibility rules can be found in Figure 11. The *level stamping* function is defined as:

$$\begin{array}{ll} \texttt{unit} \sqcup \xi \triangleq \texttt{unit} & (\tau \times \tau') \sqcup \xi \triangleq (\tau \sqcup \xi) \times (\tau' \sqcup \xi) \\ \texttt{int}^{\chi} \sqcup \xi \triangleq \texttt{int}^{\chi \sqcup \xi} & (\texttt{ref}\ \tau) \sqcup \xi \triangleq \texttt{ref}\ \tau \\ \texttt{bool}^{\chi} \sqcup \xi \triangleq \texttt{bool}^{\chi \sqcup \xi} & (\tau \to \tau')^{\chi} \sqcup \xi \triangleq (\tau \to \tau')^{\chi \sqcup \xi} \end{array}$$

Flat types are defined inductively as:

$$\texttt{unit} \text{ is flat} \qquad \texttt{int}^{\mathbf{H}} \text{ is flat} \qquad \texttt{bool}^{\mathbf{H}} \text{ is flat}$$

$$\frac{\tau_1 \text{ is flat} \qquad \tau_2 \text{ is flat}}{\tau_1 \times \tau_2 \text{ is flat}}$$

B. Sensitivity labels on references and aliasing

Most type systems for non-interference for languages with (higher-order) references annotate reference types with sensitivity labels, and annotate the typing judgment with a *program counter* label [1]–[3], [33]. These annotations are used to prevent leaks via aliasing, while allowing more programs to be typed. Our type system (§ V) does not have such annotations because some programs that are typeable using such annotations are not secure w.r.t. a termination-sensitive notion of non-interference (*e.g.,* strong low-bisimulation). For example, termination-insensitive type systems usually accept the following program as secure:

$$(\textbf{if}\ h\ \textbf{then}\ f\ \textbf{else}\ g)\ ()$$

Here, h is a high-sensitivity Boolean, and f and g are functions of type $(\texttt{unit} \to \texttt{unit})^{\mathbf{L}}$. Under a termination-sensitive notion of security, the program is not secure because f and g can examine different termination behavior.

Despite this, let us examine why exactly we do not need labels on reference types to prevent leaks via aliasing, and argue that our approach still allows for benign aliasing of references. A classic example of an information leak via aliasing is:

$$\begin{array}{l} \textbf{let}\ p_1\ r\ s\ h = r \leftarrow \textbf{true}; s \leftarrow \textbf{true}; \\ \qquad \textbf{let}\ x = (\textbf{if}\ h\ \textbf{then}\ r\ \textbf{else}\ s)\ \textbf{in} \\ \qquad x \leftarrow \textbf{false}; !\, r \end{array}$$

Both r and s contain low-sensitivity data, but by aliasing one or the other with x, the program leaks the high-sensitivity value h. In previous approaches such leaks are avoided by tracking aliasing information through sensitivity labels on references. The variable x would be typed as $(\texttt{ref int}^{\mathbf{L}})^{\mathbf{H}}$ because it was aliased in a high-sensitivity context (branching on h). The consequent assignment $x \leftarrow \textbf{false}$ is then prevented by the type system since the label on the reference ($\mathbf{H}$) is not a below the label of the values that are stored in the reference ($\mathbf{L}$).

In SeLoC, the variable x will not be typeable at all. To see why that is the case, suppose we want to prove that the program is secure. For this, we let h_1 and h_2 denote high-sensitivity inputs for two runs of the program, and r_1, s_1 (resp. r_2, s_2) denote the low-sensitivity references arguments for the left-hand side program (resp. right-hand side program). Under these high-sensitivity inputs, we need to prove that the bodies of the let-expressions are indistinguishable, *i.e.,*

$$\mathsf{dwp}\ \textbf{if}\ h_1\ \textbf{then}\ r_1\ \textbf{else}\ s_1\ \&\ \textbf{if}\ h_2\ \textbf{then}\ r_2\ \textbf{else}\ s_2\ \{\![\texttt{ref int}^{\mathbf{L}}]\!]\}$$

Proving this proposition, would in particular require proving $\mathsf{dwp}\ r_1\ \&\ s_2\ \{\![\texttt{ref int}^{\mathbf{L}}]\!]\}$, which is impossible in SeLoC.

If we remove the trailing assignment $x \leftarrow \textbf{false}$ the resulting program p_2 becomes trivially secure, and many termination-insensitive type systems accept it as such:

$$\begin{array}{l} \textbf{let}\ p_2\ r\ s\ h = r \leftarrow \textbf{true}; s \leftarrow \textbf{true}; \\ \qquad \textbf{let}\ x = (\textbf{if}\ h\ \textbf{then}\ r\ \textbf{else}\ s)\ \textbf{in} \\ \qquad !\, r \end{array}$$

Our type system cannot be used to type check this example: as we have just explained, we cannot type the let-expression at all. Despite this, we can fall back on the double weakest preconditions to verify the security of p_2, *i.e.,* we can prove:

$$\begin{array}{l} [\![\texttt{ref bool}^{\mathbf{L}}]\!](r_1, r_2) * [\![\texttt{ref bool}^{\mathbf{L}}]\!](s_1, s_2) * \\ \quad [\![\texttt{bool}^{\mathbf{H}}]\!](h_1, h_2) \vdash \mathsf{dwp}\ p_2\ r_1\ s_1\ h_1\ \&\ p_2\ r_2\ s_2\ h_2\ \{[\![\texttt{unit}]\!]\} \end{array}$$

by symbolic execution. Using our logic, we can perform a case distinction on the Boolean values h_1 and h_2, which amounts to proving $\mathsf{dwp}\ p_2\ r_1\ s_1\ \textbf{true}\ \&\ p_2\ r_2\ s_2\ \textbf{true}\ \{[\![\texttt{unit}]\!]\}$, $\mathsf{dwp}\ p_2\ r_1\ s_1\ \textbf{true}\ \&\ p_2\ r_2\ s_2\ \textbf{false}\ \{[\![\texttt{unit}]\!]\}$, *etc.* We solve all these goals by symbolic execution. This example demonstrates the advantages of combining typing with manual proofs.

We believe that the restriction on the typing of the **let** x-binding is not unreasonable in case of termination-sensitive and progress-sensitive security condition. As we have mentioned, if we take termination and timing behavior into account, the liberal compositional reasoning that is enjoyed by termination-insensitive type systems is no longer sound. In presence of higher-order functions and store, we can write the counterexample from the beginning of this section in the form of p_2 to obtain the program p_3 below:

$$\begin{array}{l} \textbf{let}\ p_3\ f\ g\ h = r \leftarrow f; s \leftarrow g; \\ \qquad \textbf{let}\ x = (\textbf{if}\ h\ \textbf{then}\ r\ \textbf{else}\ s)\ \textbf{in} \\ \qquad (!\, x)() \end{array}$$

The variable x now aliases a reference to a function. If f and g exhibit different termination behavior, then the value of h can be observed by invoking $!\, x$.

C. Generalized rule for branching

The notion of security that we use (strong low-bisimulation) allows for branching on high-sensitivity data, provided that the timing behavior of the branches is indistinguishable. However, if we branch on a high-sensitivity Boolean, it is insufficient

a) Subtyping rules:

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\frac{\chi_1 \sqsubseteq \chi_2}{\mathtt{int}^{\chi_1} <: \mathtt{int}^{\chi_2}} \qquad \frac{\chi_1 \sqsubseteq \chi_2}{\mathtt{bool}^{\chi_1} <: \mathtt{bool}^{\chi_2}} \qquad \frac{\chi_1 \sqsubseteq \chi_2 \qquad \tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{(\tau_1 \to \tau_2)^{\chi_1} <: (\tau_1' \to \tau_2')^{\chi_2}} \qquad \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

b) Typing rules:

$$\frac{\tau <: \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\ell \in \mathfrak{L}}{\Gamma \vdash \ell : \mathtt{ref\ int}^{\mathbf{L}}} \qquad \Gamma \vdash () : \mathtt{unit} \qquad \frac{i \in \mathbb{Z}}{\Gamma \vdash i : \mathtt{int}^{\chi}} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \mathtt{bool}^{\chi}}$$

$$\frac{\Gamma \vdash e : \mathtt{int}^{\chi} \qquad \Gamma \vdash s : \mathtt{int}^{\xi}}{\Gamma \vdash e + s : \mathtt{int}^{\chi \sqcup \xi}} \qquad \frac{f : (\tau \to \tau')^{\chi}, x : \tau, \Gamma \vdash e : \tau' \sqcup \chi}{\Gamma \vdash \mathbf{rec}\ f\ x = e : (\tau \to \tau')^{\chi}} \qquad \frac{\Gamma \vdash e : (\tau \to \tau')^{\chi} \qquad \Gamma \vdash s : \tau}{\Gamma \vdash e\ s : \tau' \sqcup \chi}$$

$$\frac{\Gamma \vdash e : \mathtt{bool}^{\mathbf{L}} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : \mathtt{bool}^{\mathbf{H}} \quad \Gamma \vdash v : \tau \quad \Gamma \vdash w : \tau \quad v, w \text{ are values or variables in } \Gamma \quad \tau \text{ is flat}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ v\ \mathbf{else}\ w : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{fork}\ \{e\} : \mathtt{unit}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}(e) : \mathtt{ref}\ \tau} \qquad \frac{\Gamma \vdash e : \mathtt{ref}\ \tau}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathtt{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \leftarrow e_2 : \mathtt{unit}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{ref\ int}^{\chi} \qquad \Gamma \vdash e_2 : \mathtt{int}^{\chi}}{\Gamma \vdash \mathbf{FAA}(e_1, e_2) : \mathtt{int}^{\chi}}$$

Figure 10. Typing rules of the SeLoC type system.

to verify that each individual branch is secure, we also have to verify that the two different branches are indistinguishable for the attacker. This kind of condition is present in the rule LOGREL-IF in Figure 11. We can speak of two different branches being indistinguishable because we have moved from a unary typing system to a binary logic.

Recall, that our inference rules are interpreted as an separating implication, where the premises are joined together by a separating conjunction. To prove each premise, the user of the rule has to distribute the resources they currently have among the premises. The last four premises in LOGREL-IF, however, are joined by a regular intuitionistic conjunction ($\wedge$). The user still has to prove both of those premises if they wish to apply the rule, but this time they do not have to split their resources, *i.e.,* they are able to reuse the same resource to prove all the premises. This corresponds to the fact that there are four possible combinations of branches, but only one of the combinations can actually occur.

APPENDIX B
HOCAP-STYLE MODULAR SPECIFICATIONS

We provide modular logically atomic specifications for the module of dynamically classified references (§ VI-B) in Figure 12. These specifications are stronger than the ones given in Figure 12 in the sense that they are *logically atomic*, *i.e.,* they allow one to open invariants around operations. This is achieved using the HOCAP [39] approach to logical atomicity. Note that the weaker specifications in Figure 9 (from § VI-B) can be derived from the HOCAP-style specifications in Figure 12.

APPENDIX C
SOUNDNESS

Figure 13 contains the formal model of $\mathsf{dwp}\ e_1 \,\&\, e_2\ \{\Phi\}$ as a definition in the Iris framework. Recall that this definition captures that the expressions e_1 and e_2 are executed in a lock-step manner. Since this definition is inspired by the definition of ordinary weakest preconditions in Iris and the *product program* construction [40], instead of Iris's *state interpretation* $S : \mathsf{State} \to \mathsf{Prop}$, we have a *state relation* $SR : \mathsf{State} \times \mathsf{State} \to \mathsf{Prop}$ that keeps track of both the left and right-hand side heaps.

We now provide the details of the construction of a strong low-bisimulation out of a double weakest precondition proof.

Definition 11. We define the relation $\mathcal{R}$ on configurations of the same size to be the following:

$$\begin{aligned}(e_0 e_1 \ldots e_m, \sigma_1)\ \mathcal{R}\ (s_0 s_1 \ldots s_m, \sigma_2) \triangleq \exists n : \mathbb{N}.\\ \mathsf{True} \vdash \left({}^{\top}\!\Rrightarrow^{\emptyset} \triangleright\, {}^{\emptyset}\!\Rrightarrow^{\top}\right)^n \Rrightarrow_{\top} SR(\sigma_1, \sigma_2) * I_{\mathfrak{L}} *\\ \mathsf{dwp}\ e_0 \,\&\, s_0\ \{v_1\, v_2.\ v_1 = v_2\} *\\ \ast_{1 \le i \le m}.\ \mathsf{dwp}\ e_i \,\&\, s_i\ \{\mathsf{True}\}\end{aligned}$$

Note that $\mathcal{R}$ is defined at the meta-level, *i.e.,* outside SeLoC; in particular the existential quantifier $\exists n : \mathbb{N}$ is at the meta-level. The relation $\mathcal{R}$ relates two configurations if all the threads are related by a double weakest precondition, and execution of the main threads furthermore result in the same value. The invariant $I_{\mathfrak{L}}$ (which has been defined in § IV-C) guarantees that the output locations $\mathfrak{L}$ always contain the same data between any executions of the two configurations. The existentially quantified natural number n bounds the number of times the

INTERP-SUB
$$\frac{\tau_1 <: \tau_2 \qquad [\![\tau_1]\!](v_1, v_2)}{[\![\tau_2]\!](v_1, v_2)}$$

LOGREL-SUB
$$\frac{\tau_1 <: \tau_2 \qquad \mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\tau_1]\!]\}}{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\tau_2]\!]\}}$$

LOGREL-INT-LOW
$$\frac{i \in \mathbb{Z}}{\mathsf{dwp}\ i\ \&\ i\ \{[\![\mathtt{int}^{\chi}]\!]\}}$$

LOGREL-INT-HIGH
$$\frac{i_1, i_2 \in \mathbb{Z}}{\mathsf{dwp}\ i_1\ \&\ i_2\ \{[\![\mathtt{int}^{\mathbf{H}}]\!]\}}$$

LOGREL-BOOL-LOW
$$\frac{b \in \mathbb{B}}{\mathsf{dwp}\ b\ \&\ b\ \{[\![\mathtt{bool}^{\chi}]\!]\}}$$

LOGREL-BOOL-HIGH
$$\frac{b_1, b_2 \in \mathbb{B}}{\mathsf{dwp}\ b_1\ \&\ b_2\ \{[\![\mathtt{bool}^{\mathbf{H}}]\!]\}}$$

LOGREL-BINOP
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathtt{int}^{\chi}]\!]\} \qquad \mathsf{dwp}\ s_1\ \&\ s_2\ \{[\![\mathtt{int}^{\xi}]\!]\}}{\mathsf{dwp}\ e_1 + s_1\ \&\ e_2 + s_2\ \{[\![\mathtt{int}^{\chi \sqcup \xi}]\!]\}}$$

LOGREL-REC
$$\frac{\square \forall f_1\ f_2\ v_1\ v_2.\ [\![(\tau_1 \to \tau_2)^{\chi}]\!](f_1, f_2) * [\![\tau_1]\!](v_1, v_2) \mathrel{-\!\!*} \mathsf{dwp}\ e_1[v_1/x][f_1/f]\ \&\ e_2[v_2/x][f_2/f]\ \{[\![\tau_2 \sqcup \chi]\!]\}}{\mathsf{dwp}\ (\mathbf{rec}\ f\ x := e_1)\ \&\ (\mathbf{rec}\ f\ x := e_2)\ \{[\![(\tau_1 \to \tau_2)^{\chi}]\!]\}}$$

LOGREL-APP
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![(\tau_1 \to \tau_2)^{\chi}]\!]\} \qquad \mathsf{dwp}\ s_1\ \&\ s_2\ \{[\![\tau_1]\!]\}}{\mathsf{dwp}\ e_1\ s_1\ \&\ e_2\ s_2\ \{[\![\tau_2 \sqcup \chi]\!]\}}$$

LOGREL-IF
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathtt{bool}^{\chi}]\!]\} \qquad \mathsf{dwp}\ t_1\ \&\ t_2\ \{\Phi\} \wedge \mathsf{dwp}\ u_1\ \&\ u_2\ \{\Phi\} \wedge (\chi \not\sqsubseteq \mathbf{L} \to (\mathsf{dwp}\ u_1\ \&\ t_2\ \{\Phi\} \wedge \mathsf{dwp}\ t_1\ \&\ u_2\ \{\Phi\}))}{\mathsf{dwp}\ (\mathbf{if}\ e_1\ \mathbf{then}\ t_1\ \mathbf{else}\ u_1)\ \&\ (\mathbf{if}\ e_2\ \mathbf{then}\ t_2\ \mathbf{else}\ u_2)\ \{\Phi\}}$$

LOGREL-IF-LOW
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathtt{bool}^{\mathbf{L}}]\!]\} \qquad \mathsf{dwp}\ t_1\ \&\ t_2\ \{\Phi\} \qquad \mathsf{dwp}\ u_1\ \&\ u_2\ \{\Phi\}}{\mathsf{dwp}\ \mathbf{if}\ e_1\ \mathbf{then}\ t_1\ \mathbf{else}\ u_1\ \&\ \mathbf{if}\ e_2\ \mathbf{then}\ t_2\ \mathbf{else}\ u_2\ \{\Phi\}}$$

LOGREL-IF-HIGH
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathtt{bool}^{\mathbf{H}}]\!]\} \qquad \mathsf{dwp}\ v_1\ \&\ v_2\ \{[\![\tau]\!]\} \qquad \mathsf{dwp}\ w_1\ \&\ w_2\ \{[\![\tau]\!]\} \qquad \tau \text{ is flat}}{\mathsf{dwp}\ \mathbf{if}\ e_1\ \mathbf{then}\ v_1\ \mathbf{else}\ w_1\ \&\ \mathbf{if}\ e_2\ \mathbf{then}\ v_2\ \mathbf{else}\ w_2\ \{[\![\tau]\!]\}}$$

LOGREL-FORK
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}}{\mathsf{dwp}\ \mathbf{fork}\ \{e_1\}\ \&\ \mathbf{fork}\ \{e_2\}\ \{[\![\mathtt{unit}]\!]\}}$$

LOGREL-ALLOC
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\tau]\!]\}}{\mathsf{dwp}\ \mathbf{ref}(e_1)\ \&\ \mathbf{ref}(e_2)\ \{[\![\mathtt{ref}\ \tau]\!]\}}$$

LOGREL-LOAD
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathtt{ref}\ \tau]\!]\}}{\mathsf{dwp}\ !e_1\ \&\ !e_2\ \{[\![\tau]\!]\}}$$

LOGREL-STORE
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathtt{ref}\ \tau]\!]\} \qquad \mathsf{dwp}\ t_1\ \&\ t_2\ \{[\![\tau]\!]\}}{\mathsf{dwp}\ (e_1 \leftarrow t_1)\ \&\ (e_2 \leftarrow t_2)\ \{[\![\mathtt{unit}]\!]\}}$$

LOGREL-FAA
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathtt{ref\ int}^{\chi}]\!]\} \qquad \mathsf{dwp}\ t_1\ \&\ t_2\ \{[\![\mathtt{int}^{\chi}]\!]\}}{\mathsf{dwp}\ \mathbf{FAA}(e_1, t_1)\ \&\ \mathbf{FAA}(e_2, t_2)\ \{[\![\mathtt{int}^{\chi}]\!]\}}$$

Figure 11. Compatibility rules of the SeLoC type system.

definition of double weakest preconditions has been unfolded. It is needed to show that $\mathcal{R}$ is closed under reductions.

The relation $\mathcal{R}$ allows one to "lift" double weakest precondition proofs from inside the logic:

Proposition 12. If $I_{\mathfrak{L}} \vdash \mathsf{dwp}\ e\ \&\ s\ \{v_1\ v_2.\ v_1 = v_2\}$ is derivable in SeLoC, then $(e, \sigma_1)\ \mathcal{R}\ (s, \sigma_2)$ for any $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$.

Proof. For showing $(e, \sigma_1)\ \mathcal{R}\ (s, \sigma_2)$, pick $n = 0$. Because σ_1 and σ_2 agree on the $\mathfrak{L}$-locations (*i.e.,* $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$), we can establish the state relation $SR(\sigma_1, \sigma_2)$ and the invariant $I_{\mathfrak{L}}$. □

Lemma 13. The following properties hold:

1) $\mathcal{R}$ is symmetric;
2) If $(v\vec{e}, \sigma_1)\ \mathcal{R}\ (w\vec{s}, \sigma_2)$, then $v = w$;
3) If $(\vec{e}, \sigma_1)\ \mathcal{R}\ (\vec{s}, \sigma_2)$, then $|\vec{e}| = |\vec{s}|$ and $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$;
4) If $(e_0 \dots e_i \dots, \sigma_1)\ \mathcal{R}\ (s_0 \dots s_i \dots, \sigma_2)$ and $(e_i, \sigma_1) \to_{\mathfrak{t}} (e_i'\vec{e}, \sigma_1')$, then there exist an s_i', $\vec{s}$ and σ_2' such that:
   - $(s_i, \sigma_2) \to_{\mathfrak{t}} (s_i'\vec{s}, \sigma_2')$;
   - $(e_0 \dots e_i'\vec{e} \dots, \sigma_1')\ \mathcal{R}\ (s_0 \dots s_i'\vec{s} \dots, \sigma_2')$.

By the above lemma, we now know that $\mathcal{R}$ has all the properties of a strong low-bisimulation (*cf.* [5, Definition 6]), short of being a partial equivalence relation. Since $\mathcal{R}$ is not transitive, we consider its transitive closure $\mathcal{R}^*$, and verify that all the properties of a strong low-bisimulation hold for $\mathcal{R}^*$.

Theorem 14. The relation $\mathcal{R}^*$ is *a strong low-bisimulation* on configurations.

The theorem Theorem 14 in combination with Proposition 12 implies the soundness of SeLoC (Theorem 7).

VALDEP-PERSISTENT
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2)}{\Box\,\mathsf{val\_dep}(\tau, r_1, r_2)}$$

CLASSIFICATION-AGREE
$$\frac{\mathsf{class}_{(r_1,r_2)}(\chi_1, q_1) \qquad \mathsf{class}_{(r_1,r_2)}(\chi_2, q_2)}{\chi_1 = \chi_2}$$

CLASSIFICATION-OP
$$\mathsf{class}_{(r_1,r_2)}(\chi, q_1) * \mathsf{class}_{(r_1,r_2)}(\chi, q_2) \dashv\vdash \mathsf{class}_{(r_1,r_2)}(\chi, q_1 + q_2)$$

CLASSIFICATION-1-EXCLUSIVE
$$\frac{\mathsf{class}_{(r_1,r_2)}(\chi, 1) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q)}{\mathsf{False}}$$

CLASSIFICATION-AUTH-AGREEE
$$\frac{\mathsf{class\_auth}_{(r_1,r_2)}(\chi_1) \qquad \mathsf{class}_{(r_1,r_2)}(\chi_2, q)}{\chi_1 = \chi_2}$$

CLASSIFICATION-UPDATE
$$\frac{\mathsf{class\_auth}_{(r_1,r_2)}(\chi) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, 1)}{\Rightarrow\!\mathsf{class\_auth}_{(r_1,r_2)}(\chi') * \mathsf{class}_{(r_1,r_2)}(\chi', 1)}$$

READ-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad (\forall \chi\, v_1\, v_2.\, \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * [\![\tau \sqcup \chi]\!](v_1, v_2) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * \Phi(v_1, v_2))}{\mathsf{dwp}\ \mathit{read}\ r_1\ \&\ \mathit{read}\ r_2\ \{\Phi\}}$$

WRITE-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad (\forall \chi.\, \mathsf{class\_auth}_{(r_1,r_2)}(\chi) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * [\![\tau \sqcup \chi]\!](v_1, v_2) * \Phi((),()))}{\mathsf{dwp}\ \mathit{store}\ r_1\ v_1\ \&\ \mathit{store}\ r_2\ v_2\ \{\Phi\}}$$

IS-CLASSIFIED-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad (\forall \chi\, b.\, \mathsf{class\_auth}_{(r_1,r_2)}(\chi) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * ((b = \mathbf{false} \to \chi = \mathbf{L}) \mathrel{-\!\!*} \Phi(b, b)))}{\mathsf{dwp}\ \mathit{get\_classified}\ r_1\ \&\ \mathit{get\_classified}\ r_2\ \{\Phi\}}$$

DECLASSIFY-SPEC
$$\frac{\begin{array}{c}\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q) \\ (\mathsf{class\_auth}_{(r_1,r_2)}(\chi) * \mathsf{class}_{(r_1,r_2)}(\chi, q) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\mathbf{L}) * \mathsf{class}_{(r_1,r_2)}(\mathbf{L}, q) * (\mathsf{class}_{(r_1,r_2)}(\mathbf{L}, q) \mathrel{-\!\!*} \Phi((),())))\end{array}}{\mathsf{dwp}\ \mathit{declassify}\ r_1\ v_1\ \&\ \mathit{declassify}\ r_2\ v_2\ \{\Phi\}}$$

CLASSIFY-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q) \qquad (\mathsf{class\_auth}_{(r_1,r_2)}(\chi) * \mathsf{class}_{(r_1,r_2)}(\chi, q) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\mathbf{H}) * \Phi((),()))}{\mathsf{dwp}\ \mathit{classify}\ r_1\ \&\ \mathit{classify}\ r_2\ \{\Phi\}}$$

NEW-VDEP-SPEC
$$\frac{[\![\tau \sqcup \chi]\!](v_1, v_2) \qquad (\forall r_1\, r_2.\, \mathsf{val\_dep}(\tau, r_1, r_2) * \mathsf{class}_{(r_1,r_2)}(\chi, 1) \mathrel{-\!\!*} \Phi(r_1, r_2))}{\mathsf{dwp}\ \mathit{new\_vdep}\ v_1\ \&\ \mathit{new\_vdep}\ v_2\ \{\Phi\}}$$

Figure 12. HOCAP-style specifications for dynamically classified references.

$$\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\} \triangleq \begin{cases} \Rrightarrow_{\top} \Phi(e_1, e_2) & \text{if } e_1, e_2 \in \mathsf{Val} \\ \Rrightarrow_{\top} \mathsf{False} & \text{if } e_1 \in \mathsf{Val} \text{ xor } e_2 \in \mathsf{Val} \\ \forall \sigma_1\, \sigma_2.\, SR(\sigma_1, \sigma_2) \mathrel{-\!\!*} {}^{\top}\!\Rrightarrow^{\emptyset} \mathrm{red}(e_1, \sigma_1) * \mathrm{red}(e_2, \sigma_2) * & \\ \quad \forall e_1'\, \sigma_1'\, \vec{e_1}\, e_2'\, \sigma_2'\, \vec{e_2}.\, (e_1, \sigma_1) \to_{\mathtt{t}} (e_1' \vec{e_1}, \sigma_1') \wedge (e_2, \sigma_2) \to_{\mathtt{t}} (e_2' \vec{e_2}, \sigma_2') \mathrel{-\!\!*} & \\ \quad\quad {}^{\emptyset}\!\Rrightarrow^{\emptyset} \triangleright {}^{\emptyset}\!\Rrightarrow^{\top} SR(\sigma_1', \sigma_2') * \mathsf{dwp}\ e_1'\ \&\ e_2'\ \{\Phi\} * \mathop{\Large *}_{(e_1'', e_2'') \in \vec{e_1} \times \vec{e_2}} \mathsf{dwp}\ e_1''\ \&\ e_2''\ \{\mathsf{True}\} & \text{otherwise} \end{cases}$$

Figure 13. The model of double weakest preconditions.