Compositional Non-Interference for Fine-Grained Concurrent Programs

Dan Frumin
Radboud University

Robbert Krebbers
Delft University of Technology

Lars Birkedal
Aarhus University

***Abstract*—We present SeLoC: a relational separation logic for verifying non-interference of fine-grained concurrent programs in a compositional way. SeLoC is more expressive than previous approaches, both in terms of the features of the target programming language, and in terms of the logic. The target programming language supports dynamically allocated references (pointers), higher-order functions, and fine-grained fork-based concurrency with low-level atomic operators like compare-and-set. The logic provides an invariant mechanism to establish protocols on data that is not protected by locks. This allows us to verify programs that were beyond the reach of previous approaches.**

A key technical innovation in SeLoC is a relational version of weakest-preconditions to track information flow using separation logic resources. On top of these weakest-preconditions we build a type system-like abstraction, using invariants and logical relations. SeLoC has been mechanized on top of the Iris framework in the Coq proof assistant.

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

I. Introduction

Non-interference is a form of *information flow control* (IFC) used to express security properties like confidentiality and secrecy, which guarantee that confidential information does not leak to attackers. In order to establish non-inference of programs used in practice, it is necessary to develop techniques that scale up to programming paradigms and programming constructs found in modern programming languages. Much effort has been put into that direction—*e.g.,* to support dynamically allocated references and higher-order functions [1]–[3], and concurrency [4]–[10]. For shared-memory concurrency a lot of these efforts were focused on compositional reasoning, which is needed to facilitate reasoning about components/threads in isolation without having to take all possible interference from the environment and other threads into account. 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:

- As pointed out in [6], for many interesting program modules, non-interference relies on functional correctness. For example, it may be the case that the confidentiality of the contents of a reference depends on runtime information instead of mere static information (this is called *value-dependent classification*).
- 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.

Another reason for the discrepancy between the lack of expressiveness for techniques for non-interference compared to those for functional correctness is that a lot of prior work on non-interference has focused on type systems and type system-like logics, *e.g.,* [1], [4], [6], [9], [10]. Such systems have the benefit of providing strong automation (by means of type checking), but lack capabilities to reason about functional correctness, and therefore to establish non-interference of more challenging programs.

In order to overcome aforementioned shortcomings, we take a different and more expressive approach that combines the power of type systems and concurrent separation logic. In our approach, one assigns flexible interfaces to individual program modules using types. The program modules can then be composed using typing rules, ensuring non-interference of the whole system. Individual programs can be verified against those interfaces using a relational concurrent separation logic, which allows one to carry out non-interference proofs intertwined with functional correctness proofs.

Although ideas from concurrent separation logic have been employed for establishing non-interference (for first-order programs) before, see [9], [10], we believe that the combination of typing and separation logic is new. On top of that, our approach provides a number of other advantages compared to prior work on non-interference:

- We are the first to consider non-interference in the context of a language with fine-grained concurrency. That is, our language features low-level atomic operations like compare-and-set. These operations are used to implement lock-free concurrent data structures and high-level synchronization mechanisms like locks/mutexes, whereas in prior work locks were taken to be language primitives.
- To provide a high level of expressiveness and modularity, our separation logic involves various novel features. First, to support reasoning about multiple runs of a program with different values for confidential inputs, our separation logic is *relational*. Second, to reason about sophisticated forms of sharing and ownership, as in value-dependent classifications, our logic provides a powerful *invariant* mechanism to describe expressive protocols.

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

Figure 1. Lock-free value-dependent classification.

In order to build our logic we make use of the Iris framework for concurrent separation logic [13]–[16], which provides basic building blocks, including the invariant mechanism. In order to combine typing and separation logic, we follow recent work on *logical relations* [17]–[19], but apply it to non-interference instead of functional correctness or contextual refinement.

Contributions:

- We introduce **SeLoC**, the first logic for non-interference that supports fine-grained concurrency, higher-order functions, and dynamic (higher-order) references (§ III-A). It is sound w.r.t. a standard notion of non-interference—probabilistic non-interference of Sabelfeld and Sands [5]—which is applicable to many schedulers (§ III-B).
- We show that SeLoC, which features a relational version of weakest preconditions, is expressive enough to verify non-interference of fine-grained concurrent lock-free examples relying on specific protocols (§ IV).
- We show that SeLoC supports compositional reasoning through an information-flow aware type system that can compose proofs of program modules. This type system is defined as an abstraction on top of SeLoC (§ V).
- We show that SeLoC supports compositional reasoning through *modular specifications* of program modules. Such specifications can be used to establish non-interference of clients without having to re-verify the implementation of the program module (§ VI).
- We use a novel technique for constructing a bisimulation out of a closed SeLoC proof to prove soundness (§ VII).
- We have mechanized SeLoC, its type system, its soundness proof, and all examples in the paper and appendix, in the Coq proof assistant (§ VIII). The mechanization can be found online at [20].

II. Motivating Examples

Before proceeding with the formal development of the paper in § III, we first provide two sample programs to identify challenging aspects, demonstrate the expressivity of SeLoC, and to motivate the design choices we made.

A. Fine-grained concurrency

Consider the program *prog* in Figure 1 (written in an ML-like language), which is a lock-free version of a similar lock-based program in [10]. It runs two threads in parallel, both of which operate on a reference $r.\mathit{data}$. The data in this reference has a *value-dependent classification*: the value of the flag $r.\mathit{is\_classified}$ determines the sensitivity of $r.\mathit{data}$. If the flag $r.\mathit{is\_classified}$ is set to **false**, then the data stored in $r.\mathit{data}$ is classified with *low-sensitivity* (*i.e.,* publicly observable), and if it is set to **true**, the the data is classified with high-sensitivity (*i.e.,* confidential). The record r initially contains confidential data from the integer variable *secret*. The first thread *thread1* checks if the record r is classified (*i.e.,* the flag $r.\mathit{is\_classified}$ is **true**), and if it is not, it leaks the data $r.\mathit{data}$ 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 $r.\mathit{data}$ can still be classified even if the bit $r.\mathit{is\_classified}$ 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* has been swapped, then we would violate the first condition:

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

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 $r.\mathit{is\_classified}$ the data $r.\mathit{data}$ has the correct classification (ruling out situation 1). The protocol should also ensure a form of *monotonicity*: whenever the classification becomes low (*i.e.,* $r.\mathit{is\_classified}$ becomes **false**), $r.\mathit{data}$ 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.

B. Higher-order functions and dynamic references

Consider the following program *awk*, a variation of the "awkward example" of Pitts and Stark [21]:

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

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 this "awkward example" to motivate the difficulties of reasoning about state in the presence of higher-order functions. They were interested in contextual equivalence, but as we can see, the same considerations are also relevant for the study of 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, called HeapLang, with higher-order mutable references, recursion, the **fork** operation, and atomic compare-and-swap **CAS**. HeapLang is the default programming language that is shipped with Iris [22]. Its values and expressions are:

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

We omit the usual operations on pairs, sums, and integers. 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 $(e_1; e_2) \triangleq (\textbf{let}\ \_ = e_1\ \textbf{in}\ \ e_2)$. The language has no primitive syntax for records, so we model them using pairs.

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

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

The operational semantics of HeapLang is split into three parts: thread-local head reductions $\to_{\mathsf{h}}$, thread-local reductions $\to_{\mathsf{t}}$, and thread-pool reductions $\to_{\mathsf{tp}}$.

The thread-local head reductions are of the form $(e_1, \sigma_1) \to_{\mathsf{h}} (e_2, \sigma_2)$, where each e_i is an expression and each σ_i is a heap, i.e., a finite map from locations to values ($\mathsf{State} \triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val}$). To make the thread-local reductions deterministic, we parameterize the operational semantics by an *allocation oracle* $A : \mathsf{State} \to \mathsf{Loc}$: a function from heaps to locations satisfying $A(\sigma) \notin \sigma$. The thread-local reductions have to be deterministic for the resulting thread-pool semantics to be sound (to form a Markov process). With the allocation oracle, the allocation head reduction is as follows:

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

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

Thread-local head reductions are lifted to thread-local reductions using *call-by-value evaluation contexts*:

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

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

$$\textsc{step-lift}\ \frac{(e_1, \sigma_1) \to_{\mathsf{h}} (e_2, \sigma_2)}{(K[\,e_1\,], \sigma_1) \to_{\mathsf{t}} (K[\,e_2\,], \sigma_2)} \qquad \textsc{step-fork}\ \frac{\vec{e} = K[\,()\,]\ e}{(K[\,\textbf{fork}\ \{e\}\,], \sigma) \to_{\mathsf{t}} (\vec{e}, \sigma)}$$

The thread-pool reduction relation $\to_{\mathsf{tp}}$ is defined on *configurations* $(\vec{e}, \sigma)$. The reductions between the configurations are constructed in accordance with a *probabilistic scheduler*. We adapt the definition from [5]. A probabilistic scheduler picks the next thread to step based on the number of the active threads and the *execution history*. Unlike the schedulers in [5], for simplicity we do not allow a scheduler to depend on the "low part" of the state, but we allow for a scheduler to depend on the history of the execution.

Formally, a *history* $H \in \mathsf{Hist}$ is a list of pairs of natural numbers (i, m), where m is the total number of the threads in the thread pool at a point in the history, and i is the last active thread at that point. For all consecutive pairs $(i, m)(j, n)$ we require $j < m \le n$. For a history, the number of *live threads* is the size of the thread pool at the last moment in the history:

$$\mathit{live}(\epsilon) \triangleq 1 \qquad \mathit{live}(H(i, m)) \triangleq m$$

A *scheduler* ρ is a dependently-typed function:

$$\rho : \prod (H : \mathsf{Hist}), \mathcal{D}(\{0 \ldots \mathit{live}(H) - 1)$$

where $\mathcal{D}(X)$ is a set of probability distributions on a set X.

Finally, thread-pool reductions for a scheduler ρ are obtained by lifting thread-local reductions using a scheduler ρ:

$$\frac{(e_i, \sigma_1) \to_{\mathsf{t}} (e_i'\vec{u}, \sigma_2) \qquad H_2 = H_1(i, n + |\vec{u}|)}{(H_1, e_0 \ldots e_i \ldots e_n, \sigma_1) \xrightarrow{\rho(H_1, i)}_{\mathsf{tp}} (H_2, e_0 \ldots e_i'\vec{u} \ldots e_n, \sigma_2)}$$

Here $\rho(H, i)$ represents the probability of i-th thread firing giving the history H. We write $(H_1, \vec{e}_1, \sigma_1) \to_{\mathsf{tp}} (H_2, \vec{e}_2, \sigma_2)$ for $\exists p > 0.\, (H_1, \vec{e}_1, \sigma_1) \xrightarrow{p}_{\mathsf{tp}} (H_2, \vec{e}_2, \sigma_2)$.

B. Probabilistic non-interference

To state the soundness of SeLoC, we adapt a well-established security condition known as *scheduler specific probabilistic bisimulation* by Sabelfeld and Sands [5], which is a timing-sensitive notion of non-interference for concurrent programs.

We define $=_\rho$, expressing that *histories are indistinguishable for a scheduler* ρ, as the greatest relation satisfying:

$$\frac{H_1 =_\rho H_2}{H_1(i,m) =_\rho H_2(i,m)} \qquad \frac{H_1 =_\rho H_2}{\forall i.\, \rho(H_1,i) = \rho(H_2,i)}$$

We 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. We write $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$ when σ_1 and σ_2 are *low-equivalent*, *i.e.,* they agree on all the $\mathfrak{L}$-locations: $\forall \ell \in \mathfrak{L}.\, \sigma_1(\ell) = \sigma_2(\ell) \neq \bot \wedge \sigma_1(\ell) \in \mathbb{Z}$.

Definition 1. Given a scheduler ρ, a *ρ-specific probabilistic bisimulation* is a partial equivalence relation $\mathcal{R}$ on configurations such that:

1) If $(\vec{e},\sigma_1)\ \mathcal{R}\ (\vec{e'},\sigma_2)$, then $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$.
2) If $(v_1\vec{e},\sigma_1)\ \mathcal{R}\ (v_2\vec{s},\sigma_2)$, then $v_1 = v_2$.
3) If $(\vec{e},\sigma_1)\ \mathcal{R}\ (\vec{s},\sigma_2)$, then for all H_1, H_2 with $H_1 =_\rho H_2$, and for all H_1', $\vec{e'}$, σ_1' with $(H_1,\vec{e},\sigma_1) \to_{\mathsf{tp}} (H_1',\vec{e'},\sigma_1')$ there exists a H_2', $\vec{s'}$, and σ_2' such that:
   a) $(H_2,\vec{s},\sigma_2) \to_{\mathsf{tp}} (H_2',\vec{s'},\sigma_2')$, and $H_1' =_\rho H_2'$, and $(\vec{e'},\sigma_1')\ \mathcal{R}\ (\vec{s'},\sigma_2')$;
   b) $\sum\{p \mid (H_1,\vec{e},\sigma_1) \xrightarrow{p}_{\mathsf{tp}} (H,\vec{t},\sigma) \wedge H =_\rho H_1' \wedge (\vec{t},\sigma)\ \mathcal{R}\ (\vec{e'},\sigma_1')\} =$
   $\sum\{p \mid (H_2,\vec{s},\sigma_2) \xrightarrow{p}_{\mathsf{tp}} (H,\vec{t},\sigma) \wedge H =_\rho H_2' \wedge (\vec{t},\sigma)\ \mathcal{R}\ (\vec{s'},\sigma_2')\}$

where the summation is done over multisets.

The locations from the set $\mathfrak{L}$ of output locations are assumed to be observable by the attacker. To model the input/high-sensitivity data we use free variables. For simplicity we assume that the input data also consists of integers. We then arrive at the following top-level definition of security.

Definition 2 (Security)**.** Let e be an expression with free variables $\vec{x}$. We say that e *is secure*, if for any scheduler ρ, 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 ρ-specific probabilistic 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. Although we have not investigated how to modify the semantics and the security condition to account for non-determinism on a per-thread level, 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 2 (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

This section provides 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) to verify the program *prog* from § II. The grammar of SeLoC is as follows:

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

SeLoC features the standard separation logic connectives like separating conjunction ($*$) and magic wand ($-\!\!*$). Since SeLoC is based on Iris [13]–[16], it also features 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 all of the Iris connectives in detail, but rather explain them on a by-need basis. An interested reader is referred to [16] and [23] 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. Furthermore, for convenience, 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 2. Each inference rule $\dfrac{P_1 \ \ldots\ P_n}{Q}$ in this paper should be read as an entailment $P_1 * \ldots * P_n \vdash Q$. 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 $\forall \vec{n}_1\, \vec{n}_2 \in \mathbb{Z}.\ \mathsf{dwp}\ e[\vec{n}_1/\vec{x}]\ \&\ e[\vec{n}_2/\vec{x}]\ \{v_1\, v_2.\ v_1 = v_2\}$ (with $\vec{x}$ the free variable of e), then e is secure. We defer the precise soundness statement to § IV-C.

A selection of rules for SeLoC's double weakest precondition connective[1] are given in Figure 2. Some of these rules are straightforward 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 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.,* $(\textbf{if}\ \textbf{true}\ \textbf{then}\ e\ \textbf{else}\ t) \to_{\mathsf{pure}} e$). 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. In order 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 the following symbolic execution rule:

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

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 3. $\mathsf{dwp}\ \mathit{rand}\ ()\ \&\ \mathit{rand}\ ()\ \{v_1\, v_2.\ v_1 = v_2\}$.

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

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

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

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

At this point we are tempted to apply DWP-FORK; however, in both the main thread and the forked-off thread we need the points-to connectives $\ell_1 \mapsto_{\mathsf{L}} -$ 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 denoted using boxes: $\boxed{P}^{\mathcal{N}}$, which are duplicable resources (see INV-DUP). 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 allows to allocate a new invariant $\boxed{P}^{\mathcal{N}}$ with a name space $\mathcal{N} \in \mathit{InvName}$ from a resource described by P. This allows for P to be shared 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 following 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 we have established can be shared for proving both of those goals. The first goal requires us to prove that assigning **false** to both ℓ_1 and ℓ_2 is safe. We verify this by applying 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. □

[1] Some of the SeLoC rules involve the *later modality* $\triangleright$, which is standard for dealing with recursion and impredicative invariants [16, Section 5.5]. The occurrences of $\triangleright$ can be ignored for the purposes of this paper.

DWP-VAL
$$\frac{\Phi(v_1, v_2)}{\mathsf{dwp}_{\mathcal{E}}\ v_1\ \&\ v_2\ \{\Phi\}}$$

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

DWP-FUPD
$$\frac{\Rrightarrow_{\mathcal{E}} \mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{v_1\ v_2.\ \Rrightarrow_{\mathcal{E}} \Phi(v_1, v_2)\}}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

DWP-BIND
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{v_1\ v_2.\ \mathsf{dwp}\ K_1[\,v_1\,]\ \&\ K_2[\,v_2\,]\ \{\Phi\}\}}{\mathsf{dwp}\ K_1[\,e_1\,]\ \&\ K_2[\,e_2\,]\ \{\Phi\}}$$

DWP-PURE
$$\frac{e_1 \to_{\mathsf{pure}} e_1' \qquad e_2 \to_{\mathsf{pure}} e_2' \qquad \triangleright\, \mathsf{dwp}\ e_1'\ \&\ e_2'\ \{\Phi\}}{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}}$$

DWP-FORK
$$\frac{\triangleright\, \mathsf{dwp}\ e_1\ \&\ e_2\ \{\mathsf{True}\} \qquad \triangleright\, \Phi()()}{\mathsf{dwp}_{\mathcal{E}}\ (\textbf{fork}\ \{e_1\})\ \&\ (\textbf{fork}\ \{e_2\})\ \{\Phi\}}$$

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

AWP-STORE
$$\frac{\ell \mapsto_\theta v_1 \qquad (\ell \mapsto_\theta v_2 \mathrel{-\!\!*} \Phi())}{\mathsf{awp}_\theta\ \ell \leftarrow v_2\ \{\Phi\}}$$

AWP-LOAD
$$\frac{\ell \mapsto_\theta v \qquad (\ell \mapsto_\theta v \mathrel{-\!\!*} \Phi(v))}{\mathsf{awp}_\theta\ !\ell\ \{\Phi\}}$$

AWP-ALLOC
$$\frac{\forall \ell.\ \ell \mapsto_\theta v \mathrel{-\!\!*} \Phi(\ell)}{\mathsf{awp}_\theta\ \textbf{ref}(v)\ \{\Phi\}}$$

DWP-INV-ALLOC
$$\frac{P \qquad (\boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\})}{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}}$$

INV-DUP
$$\frac{\boxed{P}^{\mathcal{N}}}{\boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}}$$

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 2. A selection of the proof rules of SeLoC.

C. Soundness

We now state SeLoC's soundness theorem, which guarantees that verified programs are actually secure w.r.t. Definition 2.

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 that these locations always contains the same data in both runs of the program. To reflect this in the logic, we define a proposition that owns the observable locations and forces them to contain the same values in both heaps:

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

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 4 (Soundness)**.** Suppose that:

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

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 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 1. We prove the following proposition, which serves as a premise for Theorem 4, and therefore implies the security of *prog*.

Proposition 5. For any integers $i_1, i_2 \in \mathbb{Z}$, we have $I_{\{out\}} \vdash \mathsf{dwp}\ prog\ out\ i_1\ \&\ prog\ out\ i_2\ \{v_1\ v_2.\ v_1 = v_2 = ((), ())\}$.

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]:

DWP-PAR
$$\frac{\mathsf{dwp}\ e_1\ \&\ s_1\ \{\Psi_1\} \qquad \mathsf{dwp}\ e_2\ \&\ s_2\ \{\Psi_2\} \qquad \begin{array}{l}(\forall v_1, v_2, w_1, w_2.\ (\Psi_1(v_1, w_1) * \Psi_2(v_2, w_2)) \mathrel{-\!\!*} \\ \qquad \Phi((v_1, w_1), (v_2, w_2)))\end{array}}{\mathsf{dwp}\ (e_1\ ||\ e_2)\ \&\ (s_1\ ||\ s_2)\ \{\Phi\}}$$

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 3:

1) Classified, if the data stored in the record is classified, and $r.is\_classified$ points to **true**;
2) Intermediate, when the data stored in the record is not classified anymore, but $r.is\_classified$ still points to **true**;
3) Declassified, when the data stored in the record is not classified and $r.is\_classified$ 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

The protocol as a transition system:

The rules for ghost state:

$$\textsc{state-agree}\quad \frac{\mathsf{in\_state}(s_1) \qquad \mathsf{state\_token}(s_2)}{s1 = s2}$$

$$\textsc{state-change}\quad \frac{s_1 \rightarrow s_2 \qquad \mathsf{in\_state}(s_1) \qquad \mathsf{state\_token}(s_1)}{\Rrightarrow_{\mathcal{E}} \mathsf{in\_state}(s_2) * \mathsf{state\_token}(s_2)}$$

$$\textsc{declassified-dup}\quad \frac{\mathsf{state\_token}(\mathsf{Declassified})}{\mathsf{state\_token}(\mathsf{Declassified}) * \mathsf{state\_token}(\mathsf{Declassified})}$$

The invariant:

$$\boxed{\begin{aligned} &\big(\mathsf{in\_state}(\mathsf{Classified}) * \exists i_1, i_2.\, r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true} * \\ &\quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i_1 * r_2.\mathit{data} \mapsto_{\mathsf{R}} i_2\big) \\ \vee &\big(\mathsf{in\_state}(\mathsf{Intermediate}) * \exists i.\, r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true} * \\ &\quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i * r_2.\mathit{data} \mapsto_{\mathsf{R}} i\big) \\ \vee &\big(\mathsf{in\_state}(\mathsf{Declassified}) * \exists i.\, r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{false} * \\ &\quad r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{false} * r_1.\mathit{data} \mapsto_{\mathsf{L}} i * r_2.\mathit{data} \mapsto_{\mathsf{R}} i\big) \end{aligned}}^{\mathcal{N}}$$

Figure 3. Value-dependent classification.

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 not important, and is described in [13], [16]. What is important, is that it enables us to define tokens $\mathsf{in\_state}, \mathsf{state\_token} : \mathsf{State} \to \mathsf{Prop}$ that satisfy the laws in Figure 3. The token $\mathsf{in\_state}$ will be shared using an invariant, while *thread2* will own the token $\mathsf{state\_token}$. Rule STATE-AGREE states that the tokens $\mathsf{in\_state}$ and $\mathsf{state\_token}$ agree on the logical state. If a thread has the token, it can change the logical state using STATE-CHANGE, but only in way that respects the protocol described by the transition system. Finally, the 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 invariant that ties together the ghost and physical state is shown in Figure 3. It is defined for the records r_1 and r_2, for the left hand side 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.

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 **true**. 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 token $\mathsf{state\_token}(\mathsf{Declassified})$ 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 token $\mathsf{state\_token}(\mathsf{Classified})$ and update the logical state with each assignment. The complete formalized proof can be found in the Coq formalization. □

$$\textsc{typed-if}\quad \frac{\Gamma \vdash e : \mathtt{bool}^{\mathbf{L}} \qquad \Gamma \vdash t : \tau \qquad \Gamma \vdash u : \tau}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ t\ \mathbf{else}\ u : \tau}$$

$$\textsc{typed-store}\quad \frac{\Gamma \vdash e : \mathtt{ref}\ \tau \qquad \Gamma \vdash t : \tau}{\Gamma \vdash e \leftarrow t : \mathtt{unit}} \qquad \textsc{typed-out}\quad \frac{\ell \in \mathfrak{L}}{\Gamma \vdash \ell : \mathtt{ref\ int}^{\mathbf{L}}}$$

Figure 4. A selection of the typing rules.

V. Type system and logical relations

We show how to define a simple information-flow aware type system 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 [17]–[19], [24], [25], we use them—for the first time—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 (§ 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, τ is a type. Some typing rules are given in Figure 4, 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.

Notice that the type system we consider has no sensitivity labels on reference types and no program counter label on the typing judgment, which is usual for security type systems for languages with (higher-order) references [1]–[3], [26]. Direct adaptation of such type systems is not sound with respect to the termination-sensitive notion of non-interference that we consider. A counterexample is provided in Appendix C-A.

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

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

Figure 5. The logical relations interpretation of types.

LOGREL-IF-LOW

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

LOGREL-STORE

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

Figure 6. A selection of compatibility rules.

attacker. The definition of $[\![\tau]\!]$ is given in Figure 5. 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 [16, 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 a technical requirement commonly employed when encoding logical relations in Iris [17]. It ensures that indistinguishable functions remain indistinguishable forever.

The interpretation of expressions $[\![\_]\!]^{\mathsf{e}}$ generalizes to open terms by considering all possible 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 $\pi_i(\gamma(x))$. We say that a substitution γ is well-typed, denoted as $[\![\Gamma]\!](\gamma)$, iff $\forall x.\, [\![\tau]\!](\gamma(\Gamma(x)))$. We then define the *semantic typing judgment* as follows:

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

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

Theorem 6 (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.

Proof. This a direct consequence of Theorem 4. □

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

Proposition 7 (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 6. □

B. Combining binary and unary reasoning

When composing the fundamental property (Proposition 7) and the soundness theorem (Theorem 6) 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 3.

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

Proof. This is a direct consequence of Proposition 5. □

Proposition 9. $\models \mathit{awk} : \texttt{int}^{\mathbf{H}} \to (\texttt{unit} \to \texttt{unit})^{\mathbf{L}} \to \texttt{int}^{\mathbf{L}}$.

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 $\mathsf{dwp}\ \mathit{awk}\ i_1\ f_1\ \&\ \mathit{awk}\ i_2\ f_2\ \{v_1\, v_2.\, v_1 = v_2 = 0\}$. 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 formalization. □

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

Figure 7. Implementation of a spin lock.

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. Proof rules for locks.

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:

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

Using the fundamental property (Proposition 7) we obtain a semantic typing judgment for the above program. Using Proposition 8 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 6).

VI. Modular specifications

We show that SeLoC supports compositional reasoning through modular specifications of program modules. That is, we show how to provide specifications of program modules that can be used opaquely by clients, and show how clients can be proved secure against such specifications without the need for examining the module's source code. A tangible benefit of this approach is that the implementation of the module can be replaced by a different one, as long as it still satisfies the same specification. We demonstrate this approach on two examples: we give modular specifications of dynamically created locks (§ VI-A) and dynamically classified references (§ VI-B).

A. Locks

The programming language considered in this paper does not provide locks as primitive constructs. Instead, it provides the low-level compare-and-set (**CAS**) operation using which

$$\begin{aligned}
&\textbf{let}\ new_vdep\ v\ =\ \left\{\begin{array}{l} data = \textbf{ref}(v);\\ is_classified = \textbf{ref}(\textbf{false}) \end{array}\right\}\\
&\textbf{let}\ read\ r\ =\ !\, r.data\\
&\textbf{let}\ store\ r\ v\ =\ r.data \leftarrow v\\
&\textbf{let}\ classify\ r\ =\ r.is_classified \leftarrow \textbf{true}\\
&\textbf{let}\ declassify\ r\ v\ =\ r.data \leftarrow v; r.is_classified \leftarrow \textbf{false}\\
&\textbf{let}\ get_classified\ r\ =\ !\, r.is_classified
\end{aligned}$$

Figure 9. Dynamically classified references.

different kinds of locking mechanisms can be implemented. In this section we consider the implementation and specification of a spin-lock, whose code is displayed in Figure 7.

Figure 8 displays the specification of the lock. The specification makes use of a relational generalization of the common *lock predicates* in separation logic [27]–[29]. 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.

In order 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 used invariant and proof is a generalization of the ordinary proof 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 here 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.

B. Dynamically classified references

We now consider a program module and specification for dynamically classified references, *i.e.,* references that contain data with value-dependent security classification. This module encapsulates and generalizes the code pertaining to dynamically classified references that we used in the example in § II-A. Encapsulating this code as a module has two advantages:

- **Modularity.** While we manually crafted an invariant and protocol in the proof of the example (Proposition 5), we now give a modular specification to the program module. We have to verify this specification once, and can then use it for the verification of any client.
- **Generality.** The specification that we present 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 formalization contains such an example.

NEW-VDEP

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

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

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

READ-SAFE

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

READ-SEQ

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

STORE-SAFE

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

STORE-SEQ

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

CLASSIFY-SEQ

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

DECLASSIFY-SEQ

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

GET-CLASSIFIED-SEQ

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

Figure 10. Derived specifications for dynamically classified references.

The implementation of the module for dynamically classified references[2] is shown in Figure 9. The proof rules for the module are shown in Figure 10. These rules are derived from more general HOCAP-style logically atomic specifications [32], which can be found in Appendix B and the Coq formalization.

The main ingredient 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 ordinary separation logic. The proof rules 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 multiple threads.

[2]In this context declassification refers to changing the dynamic classification of the reference. It is thus unrelated to static declassification policies [30], and the *declassify* function is unrelated to the eponymous function from [31].

Since the rules for *declassify* and *classify* require a full fraction ($q = 1$), they do not allow for fine-grained sharing[3]. It is therefore not possible to verify a program that runs *declassify* in parallel with *classify*. Note that it is good that this is prohibited—running these operations in parallel result in a race-condition, and you would not 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 formalization.

Verification: 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, is expressed using a protocol visualized as the transition system in Figure 11.

VII. Soundness

We discuss the proof of SeLoC's soundness theorem (Theorem 6), which connects double weakest preconditions to the notion of security, namely ρ-specific probabilistic bisimulations (Definition 1). The proof consists of two steps. First, we explain

[3]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.

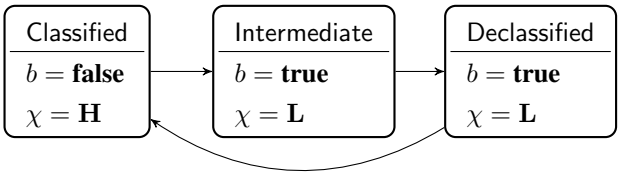

Figure 11. Transition system for the value-dependent classifications library.

the model of double weakest preconditions in the Iris framework (§ VII-A). Second, we explain how to construct a bisimulation out of this model (§ VII-B). For reasons of space, we cannot explain the details of Iris, so in this section we presume the reader is familiar with those.

A. Model of double weakest preconditions

The model of the Iris logic [15], [16] 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. Figure 12 contains the formal definition of $\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}$, which captures that the expressions e_1 and e_2 are executed in a lock-step manner. This is done by considering two cases:

- 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.

Our definition of double weakest preconditions is inspired by the definition of ordinary weakest preconditions in Iris and the *product program* construction [33]. As such, 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.

B. Constructing a bisimulation

In order to prove that double weakest preconditions imply the security condition (Definition 2), we need to construct a scheduler specific probabilistic bisimulation (Definition 1). We construct such a bisimulation in the following steps:

1) First, we define a relation $\mathcal{R}$ that "lifts" double weakest preconditions out of the SeLoC logic into the metatheory (Definition 10).
2) We then show that the relation $\mathcal{R}$ satisfies a number of bisimulation-like properties (Lemma 12).
3) The relation $\mathcal{R}$ itself is not a bisimulation because it is not transitive. We show that the transitive closure $\mathcal{R}^*$ of $\mathcal{R}$ is a *strong low-bisimulation* (Lemma 13).
4) Finally, we use the method of Sabelfeld and Sands [5] to show that $\mathcal{R}^*$ is a ρ-specific probabilistic bisimulation for any scheduler ρ (Theorem 14).

Definition 10. We define the relation $\mathcal{R}$ on configurations of the same size to be the following:

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

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 bound the number of times the 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 11. If σ_1 and σ_2 are heaps such that $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$, and $I_{\mathfrak{L}} \vdash \mathsf{dwp}\ e\ \&\ s\ \{v_1\, v_2.\, v_1 = v_2\}$ is derivable in SeLoC, then $(e, \sigma_1)\ \mathcal{R}\ (s, \sigma_2)$.

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 12. 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 $(e_i, \sigma_1) \to_{\mathsf{t}} (e_i'\vec{e}, \sigma_1')$, then there exist an s_i', $\vec{s}$ and σ_2' such that:
   - $(s_i, \sigma_2) \to_{\mathsf{t}} (s_i'\vec{s}, \sigma_2')$;
   - $(e_0 \dots e_i'\vec{e} \dots, \sigma_1')\ \mathcal{R}\ (s_0 \dots s_i'\vec{s} \dots, \sigma_2')$.

By the above lemma, we now know that $\mathcal{R}$ has all the properties of a strong low-bisimulation (*c.f.* [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 $\mathcal{R}^*$.

Lemma 13. The relation $\mathcal{R}^*$ is *a strong low-bisimulation*. That is, the following properties hold:

1) $\mathcal{R}^*$ is a partial equivalence relation (*i.e.,* symmetric and transitive);

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

Figure 12. The model of double weakest preconditions.

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 $(e_0 \ldots e_i \ldots, \sigma_1)\ \mathcal{R}^*\ (s_0 \ldots s_i \ldots, \sigma_2)$ 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:
   - $(s_i, \sigma_2) \to_{\mathsf{t}} (s_i'\vec{s}, \sigma_2')$;
   - $(e_0 \ldots e_i'\vec{e} \ldots, \sigma_1')\ \mathcal{R}^*\ (s_0 \ldots s_i'\vec{s} \ldots, \sigma_2')$.

Finally, we arrive at the desired theorem (Theorem 14), which in combination with Proposition 11 implies the soundness of SeLoC (Theorem 6).

Theorem 14. The relation $\mathcal{R}^*$ is a ρ-specific probabilistic bisimulation for any scheduler ρ.

Proof. Let ρ be an arbitrary scheduler. The conditions 1, 2, and 3(a) hold by Lemma 13. In order to verify condition 3(b), we follow the approach of Sabelfeld and Sand [5, Proposition 3] and establish the equality of sums using a one-to-one correspondence between the underlying multisets. □

VIII. Mechanization in Coq

We have mechanized the definition of SeLoC, the type system, the soundness proof (with the exception of Theorem 14), and all examples and derived constructions in the paper and the appendix using the Coq proof assistant. The mechanization has been built on top of the mechanization of Iris [14]–[16], which readily provides, among others, the Iris base logic, the invariant mechanism, and the HeapLang language.

For the proof of soundness (§ VII), we have mechanized that double weakest preconditions give rise to the existence of a strong low-bisimulation $\mathcal{R}^*$ (Lemma 13), but omitted the last step showing that $\mathcal{R}^*$ is also a scheduler-specific probabilistic bisimulation (Theorem 14). We have omitted this step because the proof is based on a standard argument by Sabelfeld and Sands [5], which involves notions from probability theory.

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 [17], [34]. 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 (1740 line of Coq code), the type system (748 lines), and all the examples (1935 lines).

IX. Related work

A. Strong and probabilistic bisimulations

The security condition we use, a scheduler specific probabilistic bisimulation due to Sabelfeld and Sands [5], has been studied in a variety of related work.

In their original paper [5], Sabelfeld and Sands devised the notion of a *strong low-bisimulation*, and applied it to a first-order stateful language with concurrency. This notion is sound w.r.t. a scheduler specific probabilistic bisimulation, and can therefore be used to ease the construction of the latter. 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 flow-sensitive analyses. Our method is flow-sensitive because we compose 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 bisimiluations. 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], [35] 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).

Another variant of probabilistic bisimulations has been developed by Smith [36], who considered considered *weak probabilistic bisimulations* for a language with a fork construct, but without dynamic allocation. Contrary to our work, the security condition obtained that way is timing-insensitive.

B. Program logics for non-interference

Early work by Beringer and Hofmann [37] 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.

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 [26] devised a capabilities-based type system for *observational determinism* [38]. Observational determinism is a formulation non-interference for concurrent programs that is substantially different from the probabilistic 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 [39], [40] 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 [41] 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 [17]–[19], [24], [25], 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. Notably, Jung *et al.* [18] use it to reason about unsafe code in Rust, and Krogh-Jespersen *et al.* [24] 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 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 demonstrated this ability by building an information-flow aware type system on top of SeLoC, which can be used to combine type checked code with manually verified code, and by verifying non-interference of a variety of challenging examples. 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 this 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 [30], [31], [42], [43].

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] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] ——, "Coq mechanization of SeLoC," 2019, available online at https://cs.ru.nl/~dfrumin/arch/seloc-0.1.tgz.

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

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

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

[24] 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.

[25] 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, Dec. 2017.

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

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

[28] 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.

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

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

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

[32] 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.

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

[34] 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.

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

[36] G. Smith, "Probabilistic noninterference through weak probabilistic bisimulation," in *CSFW*, 6 2003, pp. 3–13.

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

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

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

[40] 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.

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

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

[43] 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.

Appendix A
Type system

We give the full definition of the type system described in § V. *Types* are inductively defined as:

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

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

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

Figure 13. The subtyping rules.

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

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

$$\frac{\chi \sqsubseteq \mathbf{L} \qquad \Gamma \vdash e : \mathtt{bool}^{\chi} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : (\tau \to \tau')^{\chi} \qquad \Gamma \vdash s : \tau}{\Gamma \vdash e\ s : \tau' \sqcup \chi} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{fork}\ \{e\} : \mathtt{unit}}$$

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

Figure 14. The typing rules.

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
$$\frac{i_1, i_2 \in \mathbb{Z} \qquad \chi \not\sqsubseteq \mathbf{L}}{\mathsf{dwp}\ i_1\ \&\ i_2\ \{[\![\mathtt{int}^{\chi}]\!]\}}$$

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

LOGREL-BOOL
$$\frac{b_1, b_2 \in \mathbb{B} \qquad \chi \not\sqsubseteq \mathbf{L}}{\mathsf{dwp}\ b_1\ \&\ b_2\ \{[\![\mathtt{bool}^{\chi}]\!]\}}$$

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

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

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

LOGREL-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-SEQ
$$\frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\} \qquad \mathsf{dwp}\ s_1\ \&\ s_2\ \{\Psi\}}{\mathsf{dwp}\ e_1; s_1\ \&\ e_2; s_2\ \{\Psi\}}$$

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

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

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

Figure 15. The compatibility rules.

The *level stamping* function is defined as:

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

The subtyping and typing rules can be found in Figure 13 and Figure 14. The compatibility rules can be found in Figure 15.

APPENDIX B
HOCAP-STYLE MODULAR SPECIFICATIONS

We provide modular logically atomic specifications for the module of dynamically classified references (§ VI-B) in Figure 16. These specifications are stronger than the ones given in Figure 16 in the sense that they are *logically atomic*, *i.e.,* they allow one to open invariants around operations. This is achieved using the HOCAP [32] approach to logical atomicity. More information about HOCAP-style specifications in Iris can be found in [23, Chapter 10]. Note that the weaker specifications in Figure 16 can be derived from the rules in Figure 16.

APPENDIX C
DISCUSSION

We discuss some miscellaneous topics that are not required for an overall comprehension of the paper.

A. 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], [26]. These annotations are used to prevent leaks via aliasing, while allowing more programs to be typed. On the contrary, 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 (like the notion of scheduler specific bisimulations that we use). For example, termination-insensitive type systems usually accept the following program as secure:

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

where h is a high-sensitivity Boolean, and f and g are functions of type $(\mathtt{unit} \to \mathtt{unit})^{\mathbf{L}}$. 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 a sensitive information leakage via reference aliasing is the following program p_1:

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

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 $(\mathtt{ref}\ \mathtt{int}^{\mathbf{L}})^{\mathbf{H}}$ 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.,*

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

Proving this proposition, would in particular require proving $\mathsf{dwp}\ r_1\ \&\ s_2\ \{[\![\mathtt{ref}\ \mathtt{int}^{\mathbf{L}}]\!]\}$, 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:

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

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:

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

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 $\mathsf{dwp}\ p_2\ r_1\ s_1\ \textbf{true}\ \&\ p_2\ r_2\ s_2\ \textbf{true}\ \{[\![\mathtt{unit}]\!]\}$, $\mathsf{dwp}\ p_2\ r_1\ s_1\ \textbf{true}\ \&\ p_2\ r_2\ s_2\ \textbf{false}\ \{[\![\mathtt{unit}]\!]\}$, *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 $\textbf{let}\ x$-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:

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

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

$$\frac{\mathsf{class\_auth}_{(r_1,r_2)}(\chi) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, 1)}{\Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi') * \mathsf{class}_{(r_1,r_2)}(\chi', 1)}$$

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

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

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 16. HOCAP-style specifications for dynamically classified references.

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 $!\,x$.

B. Generalized rule for branching

The notion of security that we use (scheduler specific bisimulations) allows for branching on high-sensitivity data, provided that the timing behavior of the branches is indistinguishable. Due to relational nature of our logic, we can verify such programs to be secure, even though type systems do not allow for branching on high-sensitive data. In § V we presented a structural rule that links together two if-expressions if the branching is done on a low-sensitivity data. A sound typing rule for branching on high-sensitivity data is hard to conceive: if we branch on a high-sensitivity Boolean then it is insufficient to verify that the 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 structural rule LOGREL-IF in Figure 15. We can speak of two different branches being indistinguishable because we have moved from a unary typing system to a binary logic.

Recall, that an inference rule is 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.