Compositional Security Definitions for Higher-Order Where Declassification

JAN MENZ, Max Planck Institute for Software Systems, Germany
ANDREW K. HIRSCH, University at Buffalo, SUNY, USA
PEIXUAN LI, Pennsylvania State University, USA
DEEPAK GARG, Max Planck Institute for Software Systems, Germany

To ensure programs do not leak private data, we often want to be able to provide formal guarantees ensuring such data is handled correctly. Often, we cannot keep such data secret entirely; instead programmers specify how private data may be *declassified*. While security definitions for declassification exist, they mostly do not handle higher-order programs. In fact, in the higher-order setting no compositional security definition exists for intensional information-flow properties such as *where* declassification, which allows declassification in specific parts of a program. We use logical relations to build a model (and thus security definition) of where declassification. The key insight required for our model is that we must stop enforcing indistinguishability once a *relevant declassification* has occurred. We show that the resulting security definition provides more security than the most related previous definition, which is for the lower-order setting.

CCS Concepts: • **Security and privacy** → **Formal security models**; **Information flow control**; • **Theory of computation** → Operational semantics.

Additional Key Words and Phrases: where declassification, logical relations, relevant declassification

ACM Reference Format:
Jan Menz, Andrew K. Hirsch, Peixuan Li, and Deepak Garg. 2023. Compositional Security Definitions for Higher-Order Where Declassification. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 89 (April 2023), 28 pages. https://doi.org/10.1145/3586041

1 INTRODUCTION

Language-based information-flow control guarantees that secret information does not flow to attacker-visible outputs of a program if such a flow would violate the security policy. The strictest form of such a guarantee is often formalized as *noninterference*, which says that a program is secure with respect to an attacker when any two runs of the program that differ only in secrets result in the same attacker-visible outcomes.

In practice, we often want less restrictive security policies since many programs intentionally reveal some secret information. In other words, secret information may not remain secret in its entirety forever; parts of it may need to be *declassified* at some points. Hence, there is need for program-security definitions that admit declassification of information in conformance with a given policy. While many such definitions exist for first-order programs—where functions cannot be passed to other functions and cannot be stored in memory [Askarov and Sabelfeld 2007a,b;

Authors' addresses: Jan Menz, Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, 66123, Germany, jmenz@mpi-sws.org; Andrew K. Hirsch, Department of Computer Science and Engineering, University at Buffalo, SUNY, 113M Davis Hall, Buffalo, New York, 14260-2500, USA, akhirsch@buffalo.edu; Peixuan Li, Pennsylvania State University, University Park, Pennsylvania, 16802, USA, pzl129@cse.psu.edu; Deepak Garg, Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, 66123, Germany, dg@mpi-sws.org.

2475-1421/2023/4-ART89
https://doi.org/10.1145/3586041

Banerjee et al. 2008; Broberg and Sands 2009, 2010; Kozyri and Schneider 2020; Mantel and Sands 2004; Sabelfeld and Myers 2003]—there are fewer such definitions for higher-order programs.

Most definitions for higher-order program security with declassification target *what declassification* [Cruz et al. 2017; Li and Zdancewic 2005; Ngo et al. 2020], wherein policies specify precisely what information may be revealed. For example, a policy might specify that a secret number may not be revealed in its entirety, but its parity—whether the number is even or odd—may be revealed. However, what declassification is purely extensional, i.e. based entirely on the input-output behavior, and therefore cannot specify temporal aspects – *where* and *when* in a program information can be declassified. In practice, this aspect is also important.

For where and when declassification, all security definitions supporting higher-order programs of which we are aware [Broberg and Sands 2006; Matos and Boudol 2005] target the so-called *store-based setting*. In this setting, security labels, which represent confidentiality policies and tell us how secret the data we are dealing with is, are attached to memory locations only. Unfortunately, definitions in the store-based setting give no meaningful notion of value security, even for those values—like function values—that have suspended computation in them. Instead, all values are usually assumed secure by virtue of not doing any computation and security is defined only for *whole-program* expressions.

As a result, existing definitions of higher-order program security with where/when declassification are not compositional. By *compositional* we mean "homomorphic with respect to the programming language's constructs", i.e., a program composed of secure partial programs should also be secure. For example, applying a secure function to a secure argument always result in a secure expression. However, the existing definitions in the store-based setting described above are not compositional in this sense. To see this, consider two locations $\mathtt{l}$ and $\mathtt{h}$ labeled public and secret, respectively, along with a function $\mathtt{f} \triangleq \lambda \mathtt{x}.(\mathtt{l} := !\,\mathtt{h})$, which copies $\mathtt{h}$ to $\mathtt{l}$ ignoring its argument $\mathtt{x}$. Clearly, $\mathtt{f}$'s body is insecure because $\mathtt{f}$ leaks $\mathtt{h}$ without declassifying it. However, store-based definitions deem all values secure and thus deem $\mathtt{f}$ secure. Next, consider the (trivially secure) argument 5. If the security definition were compositional, the expression $\mathtt{f}\ 5$ should also be secure. But, the expression $\mathtt{f}\ 5$ leaks $\mathtt{h}$ to $\mathtt{l}$, which is recognized as a security breach by the aforementioned store-based definitions!

As an immediate consequence of this lack of compositionality, programs cannot be verified secure against these definitions *modularly*. Rather, one must verify whole-program security. This constraint can be problematic. For example, libraries may be used in many different programs which do not yet exist. Hence, the authors of a library *cannot* verify the security of their library. Instead, each client must verify their program, including any library code they use, themselves. This may make some verification tasks—such as those for sufficiently large programs—infeasible. Using a compositional security definition instead, we can verify each library function *once*, reusing the security proof every time the library gets used.

Since type systems are usually compositional by construction, many papers side step this problem by using a type system to enforce a noncompositional security property. This allows programmers to reuse the proof that a well typed library function is secure, just as we advocated above. However, such type systems are necessarily conservative; they fail to recognize some secure programs as secure. For instance, imagine that we wish to verify the security of a software library used to build online shops. This library provides functions to calculate shipping costs. One function sets a new shipping cost and logs the fact that the shipping cost has been set (this log is for debugging):

$$\mathsf{SetShippingCosts} \triangleq \lambda \mathsf{cost}.\, \mathsf{shipping\_cost} := \mathsf{cost}; \mathsf{costs\_set} := \mathsf{true}$$

Another function calculates and sets the shipping costs based on whether the customer is living abroad or not:

```
CalculatePostage ≜ λisAbroad. λfee_local. λfee_abroad. λcustomer. if isAbroad customer
                                                                   then SetShippingCosts fee_abroad
                                                                   else SetShippingCosts fee_local
```

This function takes several parameters chosen by the client of the library and a customer of the online shop as inputs. Because the location of the customer is sensitive data, the policies of the result of the function parameter isAbroad and of the variable shipping_cost are more secret than the log variable costs_set. While no information about (isAbroad customer) and shipping_cost leaks to costs_set in the computation of CalculatePostage, the CalculatePostage function would be rejected as insecure by most security type systems. After all, it contains writes of less-secret data (the setting of costs_set) controlled by more-secret data (the result of isAbroad customer). The type systems cannot determine that the location of the customer is not leaked into costs_set via the writes in the two branches, so they reject the program, even though it is intuitively secure.

Because of this shortcoming, we cannot exclusively rely on type systems for compositional verification of our programs. Instead, we need more expressive security definitions that are compositional in their own right. Such definitions also have an additional benefit: they allow us to apply the powerful method of *semantic typing*. In this technique, a verification engineer proves the bulk of the program secure using the type system. Only small parts of the program that require security reasoning beyond the scope of the type system, such as the CalculatePostage function above, are proved secure directly against the security definition in the semantics. The proofs done in the type system and those done directly in the semantics are then composed using the compositionality of the security definition.

Our contribution. Our goal in this paper is to develop a security definition for *where declassification* that is *compositional* and *supports higher-order programs*. Since the store-based setting used by current definitions of where declassification leads to noncompositional definitions, we turn to an alternative setting, the *value-based setting*. In the value-based setting, values (and types) have associated security labels. We particularly work with a *fine-grained* value-based setting, where *every* value and type is associated with a security label.

Changing to the value-based setting allows us to define security via *logical relations*, creating a compositional-by-construction security definition. Existing logical-relations definitions of information-flow security define *extensional* security properties—i.e., properties where security only depends on the input-output behavior of the program [see e.g., Cruz et al. 2017; Frumin et al. 2021; Gregersen et al. 2021; Ngo et al. 2020; Rajani and Garg 2018]. For such properties, logical relations achieve compositionality by studying how the computation suspended in higher-order values can execute—the model simply evaluates those computations completely! For instance, given two functions, a logical-relations model of noninterference would say that those functions are indistinguishable if, given indistinguishable inputs, they give indistinguishable outputs. However, where and when declassification allow and disallow declassification based on internal state, such as the current state of the local memory ("when") or the subexpression being evaluated ("where"). These aspects cannot be observed from the outside by just analyzing the input/output behavior of the program, making them *intensional* security definitions. Consequently, depending on how the internal state changes during the execution, we can no longer expect two functions given indistinguishable inputs to produce indistinguishable results. Hence, **a security definition for where declassification must stop enforcing indistinguishability under some circumstances.**

To model this requirement, we identify and formalize what we call *relevant declassification steps*. Relevant declassification steps are those for which we both must stop enforcing indistinguishability *and* can do so securely. This work uses relevant declassification to build a logical relation. This insight—that we can stop enforcing indistinguishability after a relevant declassification—can, we believe, generalize to other notions of security for systems with intensional declassification. However, since we work in a setting with higher-order state and nondeterministic allocation of memory, this formalization is quite technical. Our main technical contribution is a description of such a formalization of relevant declassification, a logical relation built around this formalization, and a compositional security definition derived from the logical relation.

Concretely, our contributions are as follows: first, we present λ-WHR, a higher-order programming language and security type system allowing explicit where declassification. λ-WHR is based on a language and type system for noninterference called FG [Rajani and Garg 2018] and extends it with declassification policies in the style of Flow Locks [Broberg and Sands 2006, 2009]. In Flow Locks, distinguished variables called *locks* specify *when* (i.e., in which states) a value may be declassified to an actor. For example, the policy $\{\{\mathsf{allow}\} \Rightarrow \mathtt{alice}\}$ says "the actor alice can see whatever this policy protects if the lock allow is open." Policies are then used as security labels: the visibility of a value labeled with a policy is subject to that policy. This allows value protection to depend on locks, which can be opened and closed. Unlike Flow Locks, the lock-opening and -closing constructs of λ-WHR are syntactically scoped for technical convenience. Hence, whether a lock is open or not is clearly determined by the program subexpression being executed which means that λ-WHR policies model where declassification, rather than when declassification.

Second, we extend FG's compositional security definition to support where declassification. FG achieves compositionality by relying on a *binary logical relation*—a relational interpretation of labeled types defining when two values or expressions of a given type are indistinguishable to an adversary. The logical relation is indexed with a (labeled) type and a Kripke world [Ahmed et al. 2009], which specifies types of locations in the heap. This relation treats functions as indistinguishable exactly when they are extensionally indistinguishable, that is they produce indistinguishable results given indistinguishable inputs, forcing compositionality. A program is secure if it is indistinguishable from itself when run with different secrets. To handle declassification, we additionally index the relation with the open locks in the two programs and add a formalization of relevant declassification steps, which captures when we must stop enforcing indistinguishability.

Third, we prove λ-WHR's type system sound relative to this semantic model. This proof shows that every typing rule is semantically admissible. That is, our typing rules remain valid if we replace syntactic typing in premises and the conclusion with semantic typing based on the logical relation. This establishes that typing a program is a sound technique for proving it secure, and that mixing type-based and semantic proofs is sound. It also witnesses the compositionality of the security definition.

Finally, we connect our security definition to a prior security definition for Flow Locks [Broberg and Sands 2009]. This security definition is based on a different kind of security model: *attacker knowledge* [Askarov and Sabelfeld 2007a]. Briefly, this security model first defines what an attacker knows based on observations it has made during the execution of a program. Then, it deems a program secure if the attacker's knowledge does not increase except at relevant declassification events (i.e., declassification events that reveal new data to the attacker). Flow Locks' store-based security definition is not compositional and is limited to first-order state (where functions cannot be stored in the heap), while our definition is compositional and supports higher-order state. We prove that our security definition, restricted to first-order state, is stronger—i.e., less permissive—than the prior definition showing that compositionality does not weaken security.

Actors	a		
Locks	σ		
Lock Sets	Σ	$\in$	$\mathcal{P}(\text{Locks})$
Clauses	c	$::=$	$\Sigma \Rightarrow a$
Policies	p	$\in$	$\mathcal{P}(\text{Clauses})$
Types	A	$::=$	$\mathsf{unit} \mid \mathbb{N} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathsf{ref}\ \tau \mid \tau_1 \xrightarrow{\Sigma, p} \tau_2$
Annotated Types	τ, τ_1, τ_2	$::=$	A^p
Variables	x	$\in$	$\mathbb{V}$
Numbers	n	$\in$	$\mathbb{N}$
Locations	l	$\in$	$\mathcal{L}$
Expressions	e, e', e''	$::=$	$x \mid () \mid n \mid \lambda x.\, e \mid (e, e') \mid \mathsf{fst}\ (e) \mid \mathsf{snd}\ (e)$
		$\mid$	$\mathsf{inl}\ (e) \mid \mathsf{inr}\ (e) \mid \mathsf{case}\ e\ \mathsf{of} \mid \mathsf{inl}\ (x) \Rightarrow e' \mid \mathsf{inr}\ (y) \Rightarrow e''$
		$\mid$	$e\ e' \mid \mathsf{new}\ (e, \tau) \mid\ !\, e \mid e := e'$
		$\mid$	$\mathsf{open}\ \sigma\ \mathsf{in}\ e \mid \mathsf{close}\ \sigma\ \mathsf{in}\ e \mid l$
Values $\mathcal{V}$	v, v'	$::=$	$\lambda x.\, e \mid (v, v') \mid \mathsf{fst}\ (v) \mid \mathsf{snd}\ (v) \mid \mathsf{inl}\ (v) \mid \mathsf{inr}\ (v) \mid () \mid l \mid n$

Fig. 1. Syntax

For reasons of space and readability, proofs and some technical details are omitted from this paper. The accompanying technical appendix [Menz et al. 2023] contains these.

2 LANGUAGE

We base λ-WHR on FG [Rajani and Garg 2018], a simply typed λ-calculus with a unit type, natural numbers, sums, pairs, references, functions, and information-flow labels. We extend this core calculus with the declassification mechanism and the policy language of Flow Locks [Broberg and Sands 2006, 2009]. The syntax of this adapted language is shown in Figure 1. The accompanying technical appendix adds the possibility of branching on locks and for changes to locks to be observed, based on an extension to Flow Locks known as Paralocks [Broberg and Sands 2010]. Note that, while our language does not explicitly include booleans and if expressions, we will use them freely in examples via their usual encoding as sum types.

Flow Locks allows declassification via *locks*—usually denoted by σ, σ', σ_1, etc.—which can be opened and closed using the open and close constructs. We can think of these locks as boolean flags, denoting whether data is protected or not. In other words, locks placed on data, while closed, prevent certain actors from seeing that data. Opening the lock removes this protection. Note that, unlike Flow Locks, λ-WHR uses syntactically scoped open and close statements. This matches the functional nature of λ-WHR better than the unscoped, imperative setting of prior work.

To see how locks allow for declassification, consider the following example: a conference wants to publish the list of accepted papers, but only after the program committee has made all acceptance decisions. The conference software contains the following function, which publishes a paper to a publicly visible output out[1]:

$$\mathsf{publish} \triangleq \lambda x.\, \mathsf{out} := x$$

The list of accepted papers could then be published by

$$\mathsf{publishAll} \triangleq \mathsf{applyToList}\ \mathsf{publish}\ \mathsf{acceptedPapers}$$

[1]While a more realistic model would explicitly model input-output channels, we choose to minimize our calculus and use references for these purposes. This does not change the information-flow reasoning in any way.

where acceptedPapers is the list of accepted papers. While this correctly publishes the list of papers, it does not enforce our security policy, since nothing prevents this function from being run before the program committee has made its decision.

To prevent this leak, we introduce a lock published that protects papers from being published. Thus, we associate the Flow-Locks *policy* {{published} ⇒ pub} with each paper, where pub is an *actor* standing for the public. This association tells us that the public is only allowed to see any information derived from a paper if the lock published has been opened. In general, we associate each term with a Flow-Locks *policy* determining an information-flow policy. These *policies* are defined as sets of *clauses* of the form $\Sigma \Rightarrow \alpha$, where Σ is a set of locks and α is an actor [Broberg and Sands 2006, 2009]. Information labeled with a clause $\Sigma \Rightarrow \alpha$ may be seen by the actor α if all locks in Σ are open. A policy allows an information flow if it is allowed by any clause.

Since any data written to out can be observed by the public, we associate the data stored in out with the policy {∅ ⇒ pub}. With this association, we must ensure that whenever we use publish, the published lock is statically known to be open. To do so, we can change publishAll to check that the current date (stored in the variable today) is later than the publication date (stored in pubDate), and then open the published lock.

```
publishAll ≜ if today > pubDate
             then open published
                  in applyToList publish acceptedPapers
             else ()
```

Alternatively, we could change publish to check the policy and open the lock if appropriate. Remember that the opened lock is closed at the end of the scope of the open expression. Hence this approach opens (and closes) the lock for each assignment to out separately.

```
publish ≜ λx. if today > pubDate
              then open published
                   in out := x
              else ()
```

We will see in Section 3 that each implementation of this example gets a different type. We will also see how the type system uses those types to force the programmer to use the different functions correctly.

2.1 Operational Semantics

We use a small-step, call-by-value operational semantics. In order to define this semantics we need some knowledge about the *context* in which programs execute. The context contains a (heap) state S which maps finitely many locations to both the value currently stored at that location and the type of that value. If $(l \rightarrowtail (v, \tau)) \in S$ for a state S, we say that $S(l) = v$ and $\text{type}(S, l) = \tau$. Because execution steps can change the state, every execution step results in a new state in addition to the reduced term.

Flow Locks' attacker model [Broberg and Sands 2009], which we use, assumes that an attacker can observe changes to the state. To model this assumption, each reduction step also outputs an *observation* ω, which specifies what changes to the state an attacker can see due to the step. There are two types of observations (also from Flow Locks [Broberg and Sands 2009, 2010]): the empty observation ϵ and $l_\tau(v)$, which says that the value v was written to location l that should hold values of type τ. We define the policy of an observation as follows:

$$\text{pol}(\omega) = \begin{cases} p & \text{if } \omega = l_\tau(v) \text{ and } \tau = A^p \\ \text{undefined} & \text{if } \omega = \epsilon \end{cases}$$

$$\frac{\Sigma \cup \{\sigma\} \vdash e, S \xrightarrow{\omega;\Sigma'} e', S'}{\Sigma \vdash \mathsf{open}\ \sigma\ \mathsf{in}\ e, S \xrightarrow{\omega;\Sigma'} \mathsf{open}\ \sigma\ \mathsf{in}\ e', S'}\ \textsc{Eopened} \qquad \frac{}{\Sigma \vdash \mathsf{open}\ \sigma\ \mathsf{in}\ v, S \xrightarrow{\epsilon;\Sigma} v, S}\ \textsc{EopenedBeta}$$

$$\frac{\Sigma \setminus \{\sigma\} \vdash e, S \xrightarrow{\omega;\Sigma'} e', S'}{\Sigma \vdash \mathsf{close}\ \sigma\ \mathsf{in}\ e, S \xrightarrow{\omega;\Sigma'} \mathsf{close}\ \sigma\ \mathsf{in}\ e', S'}\ \textsc{Eclosed} \qquad \frac{}{\Sigma \vdash \mathsf{close}\ \sigma\ \mathsf{in}\ v, S \xrightarrow{\epsilon;\Sigma} v, S}\ \textsc{EclosedBeta}$$

$$\frac{(l \rightarrowtail (v, \tau)) \in S}{\Sigma \vdash !\,l, S \xrightarrow{\epsilon;\Sigma} v, S}\ \textsc{EDerefBeta} \qquad \frac{l \notin \mathrm{dom}(S)}{\Sigma \vdash \mathsf{new}(v, \tau), S \xrightarrow{l_\tau(v);\Sigma} l, S \cup \{l \rightarrowtail (v, \tau)\}}\ \textsc{ENewBeta}$$

$$\frac{l \in \mathrm{dom}(S) \qquad \mathrm{type}(S, l) = \tau}{\Sigma \vdash l := v, S \xrightarrow{l_\tau(v);\Sigma} (), S[l \rightarrowtail (v, \tau)]}\ \textsc{Eassign}$$

Fig. 2. Small-Step Reduction (Selected Rules)

Our operational semantics also tracks which locks are open when a step is taken. Since the language we present here does not allow executions to branch depending on the lock state, this may be surprising. However, these locks are used in our definition of security. Moreover, we allow branching on the lock state in the extended language of the technical appendix.

One might expect that we return a new lock set after every step, just as we do for the state. In particular, one might naïvely expect $\mathsf{open}\ \sigma\ \mathsf{in}\ e$ to reduce to e with a new lock set $\Sigma \cup \{\sigma\}$ when executed for one step in Σ. In fact, this is exactly how Broberg and Sands [2006, 2009, 2010] handle changes to the lock state. However, because in our setting open and close are scoped, we need to return to the original lock set after their scope has ended. If we use the naïve semantics, we lose vital information. To see this, consider the three programs $\mathsf{open}\ \sigma\ \mathsf{in}\ (e\ e')$, $(\mathsf{open}\ \sigma\ \mathsf{in}\ e)\ e'$, and $(\mathsf{open}\ \sigma'\ \mathsf{in}\ e)\ e'$. Assume that the first two programs are executed in lock set $\{\sigma'\}$ and the last one is executed in $\{\sigma\}$. With the semantics above all of them would reduce in one step to $e\ e'$ with a new lock set $\{\sigma, \sigma'\}$. We would thus lose the ability to distinguish them, so we would not know when the scope of the open ends and which locks to reset. To solve this problem, we reduce e *inside* the scope of the open. The lock σ is open within this scope, i.e., while e reduces. When e has reduced to a value v, we remove the open and return v.

Next, consider the program $\mathsf{open}\ \sigma\ \mathsf{in}\ \mathsf{new}\ (5, \mathcal{N}^{\mathsf{P}})$ with open locks Σ. The security of this program depends on the security of the subterm $\mathsf{new}\ (5, \mathcal{N}^{\mathsf{P}})$ with open locks $\Sigma \cup \{\sigma\}$. Since this subterm is the redex of the full term, we say that $\Sigma \cup \{\sigma\}$ is the *active* lock set in the reduction of the full term. In general, the active lock set of a reduction step is the lock set at the redex reduced by the step. We need to know this active lock set in order to determine whether the reduction step is secure. We therefore instrument our small-step relation to explicitly track the active lock set.

We now have the pieces to define reduction. A reduction step has the form $\Sigma \vdash e, S \xrightarrow{\omega;\Sigma'} e', S'$, which means that the expression e, in set of open locks Σ and state S, can take a step to the expression e', changing the state to S' and producing an observation ω. The lock set Σ' is the active lock set at the point of reduction. Selected reduction rules can be found in Figure 2. The full rules can be found in our technical appendix.

The reduction rules define a call-by-value semantics. The initial lock set is static and remains constant over several steps of execution, while the structure of the expression being executed

$$\frac{p \sqsubseteq p' \qquad A <: B}{A^{p} <: B^{p'}}\ \textsc{Sub-Policy} \qquad \frac{\tau_0 <: \tau_1 \qquad \tau_2 <: \tau_3 \qquad p' \sqsubseteq p \qquad \Sigma \subseteq \Sigma'}{\tau_1 \xrightarrow{\Sigma, p} \tau_2 <: \tau_0 \xrightarrow{\Sigma', p'} \tau_3}\ \textsc{Sub-Arrow}$$

$$\frac{\Gamma, x : \tau_1;\ \Sigma';\ \theta \vdash_{pc'} e : \tau_2}{\Gamma;\ \Sigma;\ \theta \vdash_{pc} \lambda x.e : (\tau_1 \xrightarrow{\Sigma', pc'} \tau_2)^{\bot}}\ \lambda$$

$$\frac{\Gamma; \Sigma \cup \{\sigma\}; \theta \vdash_{pc} e : \tau}{\Gamma; \Sigma; \theta \vdash_{pc} \mathsf{open}\ \sigma\ \mathsf{in}\ e : \tau}\ \textsc{Open}$$

$$\frac{\Gamma; \Sigma \setminus \{\sigma\}; \theta \vdash_{pc} e : \tau}{\Gamma; \Sigma; \theta \vdash_{pc} \mathsf{close}\ \sigma\ \mathsf{in}\ e : \tau}\ \textsc{Close}$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \theta \vdash_{pc} e : (\tau_1 + \tau_2)^{p} \\ p \sqsubseteq \tau \qquad \Gamma, x : \tau_1;\ \Sigma;\ \theta \vdash_{pc \sqcup p} e_1 : \tau \qquad \Gamma, y : \tau_2;\ \Sigma;\ \theta \vdash_{pc \sqcup p} e_2 : \tau\end{array}}{\Gamma;\ \Sigma;\ \theta \vdash_{pc} \mathsf{case}\ e\ \mathsf{of}\ |\ \mathsf{inl}\ (x) \Rightarrow e_1\ |\ \mathsf{inr}\ (y) \Rightarrow e_2 : \tau}\ \textsc{case}$$

$$\frac{\begin{array}{c}\Gamma;\ \Sigma;\ \theta \vdash_{pc} e_1 : (\tau_1 \xrightarrow{\Sigma', pc'} \tau_2)^{p} \\ \Gamma;\ \Sigma;\ \theta \vdash_{pc} e_2 : \tau_1' \qquad p \sqsubseteq \tau_2 \qquad pc \sqcup p \sqsubseteq pc' \qquad \tau_1' <: \tau_1 \qquad \Sigma \supseteq \Sigma'\end{array}}{\Gamma; \Sigma; \theta \vdash_{pc} e_1\ e_2 : \tau_2}\ \textsc{app}$$

$$\frac{\Gamma;\ \Sigma;\ \theta \vdash_{pc} e : (\mathsf{ref}\ \tau)^{p} \qquad p \sqsubseteq \tau' \qquad \tau <: \tau'}{\Gamma;\ \Sigma;\ \theta \vdash_{pc} !e : \tau'}\ \textsc{Deref}$$

$$\frac{\Gamma;\ \Sigma;\ \theta \vdash_{pc} e : \tau' \qquad pc \sqsubseteq \tau \qquad \tau'(\Sigma) <: \tau}{\Gamma;\ \Sigma;\ \theta \vdash_{pc} \mathsf{new}\ (e, \tau) : (\mathsf{ref}\ \tau)^{\bot}}\ \textsc{New} \qquad \frac{}{\Gamma;\ \Sigma;\ \theta \vdash_{pc} l : (\mathsf{ref}\ \theta(l))^{\bot}}\ \textsc{Loc}$$

$$\frac{\Gamma;\ \Sigma;\ \theta \vdash_{pc} e : (\mathsf{ref}\ \tau')^{p} \qquad \tau(\Sigma) <: \tau' \qquad \Gamma; \Sigma; \theta \vdash_{pc} e' : \tau \qquad pc \sqcup p \sqsubseteq \tau'}{\Gamma;\ \Sigma;\ \theta \vdash_{pc} e := e' : \mathsf{unit}^{\bot}}\ \textsc{Assign}$$

Fig. 3. Subtyping and Type System (Selected Rules)

determines the effective lock set at the point of reduction. Most rules are standard, apart from the lock set and observations. Most β-reduction steps are unobservable, and thus produce the observation ϵ. One example is **EDerefBeta** which reads a value stored in an existing location and silently outputs ϵ.

Memory-manipulating reductions produce nontrivial observations. The rule **ENewBeta** allocates a new location l, storing the value v and the specified type τ in l. It produces the observation $l_\tau(v)$ and returns the new location l. **Assign** updates an existing location l of type τ in the state S to a value v (written $S[l \rightarrowtail (v, \tau)]$) and reduces to $()$ with observation $l_\tau(v)$.

Finally, the lock-manipulating rules determine our declassification mechanism. The rule **Opened** allows e to reduce with the lock σ open. Once e has reduced to a value v, **OpenedBeta** returns v with observation ϵ. The rules for close work similarly.

3 TYPE SYSTEM

Our type system extends the FG type system [Rajani and Garg 2018], which enforces noninterference without any declassification. Most rules remain unchanged, though we add rules for open and close and make small changes necessary for declassification to the rules manipulating state. We annotate every type with a Flow-Locks policy [Broberg and Sands 2006, 2009], including types contained within compound types.

We start by reviewing the properties of Flow-Locks policies as explored by Broberg and Sands [2006, 2009]. As is usual, policies form a join semilattice. A policy p is below a policy q, written $p \sqsubseteq q$, if policy p allows more actors to observe data labeled by it than does q. More specifically, this means that for every clause in q there is a less restrictive clause in p. A clause $\Sigma_1 \Rightarrow a$ is less restrictive than $\Sigma_2 \Rightarrow a$ if $\Sigma_1 \subseteq \Sigma_2$. The least restrictive policy $\bot \triangleq \{\emptyset \Rightarrow a \mid a \text{ is an actor}\}$ allows every actor to observe the labeled data in any lock state, while the most restrictive policy $\top \triangleq \emptyset$ does not allow any actor to observe the labeled data in any lock state. The general principle of secure information flow is that information should only flow upwards in the policy lattice: data labeled p may flow to a location labeled q if $p \sqsubseteq q$. For this reason, when $p \sqsubseteq q$, we also say that policy p *flows to* q.

The join of two policies p and q is $p \sqcup q \triangleq \{\Sigma_1 \cup \Sigma_2 \Rightarrow a \mid \Sigma_1 \Rightarrow a \in p \wedge \Sigma_2 \Rightarrow a \in q\}$. Finally, policies can be updated, or *specialized*, with regard to the current lock set. The specialization of p with Σ (written $p(\Sigma)$) represents the "effective" policy p when locks in Σ are assumed to be open: $p(\Sigma) \triangleq \{\Sigma_1 \backslash \Sigma \Rightarrow a \mid \Sigma_1 \Rightarrow a \in p\}$.

We lift the ordering on policies and policy specialization to policy-annotated types.

Definition 3.1.

$$A^p \sqsubseteq p' \triangleq p \sqsubseteq p'$$
$$p \sqsubseteq A^{p'} \triangleq p \sqsubseteq p'$$
$$A^p \sqsubseteq B^{p'} \triangleq p \sqsubseteq p'$$

$$\text{For } \tau = A^p \text{ we define } \tau(\Sigma) \triangleq A^{p(\Sigma)}.$$

Typing judgments have the form $\Gamma; \Sigma; \theta \vdash_{pc} e : \tau$, where the context Γ maps variables to their annotated types, the lock set Σ represents the open locks, the *state environment* θ gives types to locations, and pc is a policy. A judgment $\Gamma; \Sigma; \theta \vdash_{pc} e : \tau$ means that in environment Γ expression e has type τ if at least the locks in Σ are open and locations store data of the types specified by θ. The policy pc is a lower bound on the policies of observations produced by e.

Our type system enjoys subtyping, which we write $\tau_1 <: \tau_2$ meaning that τ_1 is a subtype of τ_2. As usual, this relationship implies that all terms of type τ_1 can be used as if they had type τ_2 (the restrictions on use posed by the policies in type τ_2 are stricter than those posed by τ_1). Selected typing and subtyping rules for our type system can be found in Figure 3, with the full type system found in the accompanying technical appendix. The remaining subtyping rules are entirely standard and unchanged from FG [Rajani and Garg 2018]. We highlight changes from FG [Rajani and Garg 2018] due to the addition of locks and declassification by giving the changes a yellow background. For instance, in addition to standard subtyping requirements, function types may require fewer locks than their supertypes.

The rules λ and **APP** deal with functions. In addition to annotating functions with a program-counter label pc', we also annotate function types with a set of locks Σ', representing the locks which must be open for the function body to run. To ensure that the body can be safely run if the locks in Σ' are open, the rule λ checks that the body type checks with Σ'.

The rule **app** ensures that a function call is safe by checking that the local pc flows to pc', and then checks that the open locks at the point of application are a superset of Σ'. Moreover, **app** checks that the policy p on the function type flows to the output of the function and to pc', ensuring that information about the function is not leaked via either the result or any effects in the function body. This reasoning also applies to the rules **deref** and **assign**, which manipulate references, ensuring that the label p flows to the type of data read from or stored in a reference, respectively.

The rules **open** and **close** simply check that the body of the respective construct is well typed when the lock being opened or closed is added to or removed from the original lock set.

The rule **case** ensures that the p labeling the data being case-analyzed is allowed to flow to the level of the result type. Additionally it prevents leaks via implicit information flows by checking the two branches with a pc that is at least p.

The typing rules **new** and **assign** for memory allocation and assignment allow for declassification by specialization with the current lock set. Hence, the set of current locks is taken into account when writing to memory. Note that this specialization is not done when checking for implicit information flows (e.g., in the rule for case or function application). To see why, consider typing the following expression in a lock set Σ which contains σ.

$$\begin{array}{l}\mathsf{case}\ x\ \mathsf{of}\\ |\ \mathsf{inl}\,(y) \Rightarrow \mathsf{close}\ \sigma\ \mathsf{in}\ l := \mathsf{inl}()\\ |\ \mathsf{inr}\,(z) \Rightarrow \mathsf{close}\ \sigma\ \mathsf{in}\ l := \mathsf{inr}()\end{array}$$

This expression leaks information about x to l. However, if we took the current lock set into account, we might allow branching on x because σ is open and, hence, x is declassified. But in the branches σ is closed, which could result in a leak that is not in compliance with the policy.

Consider again our example from Section 2. Recall that we had a list of papers acceptedPapers containing papers which were protected by policy $\{\{\mathsf{published}\} \Rightarrow \mathsf{pub}\}$ and a location out storing data with policy $\{\emptyset \Rightarrow \mathsf{pub}\}$. For simplicity, we give papers the type $\mathcal{N}$. Our original (insecure) implementation was the following:

$$\begin{array}{rl}\mathsf{publish} & \triangleq \lambda x.\, \mathsf{out} := x\\ \mathsf{publishAll} & \triangleq \mathsf{applyToList}\ \mathsf{publish}\ \mathsf{acceptedPapers}\end{array}$$

If we type publish as $\mathcal{N}^{\{\{\mathsf{published}\}\Rightarrow\mathsf{pub}\}} \xrightarrow{\{\mathsf{published}\},\{\emptyset\Rightarrow\mathsf{pub}\}} \mathsf{unit}^{\bot}$, then it may only be invoked where the lock published is open. Because this is not the case above, the type system rejects this implementation. In order for the implementation of publishAll to be accepted by the type system we can change it as follows:

$$\begin{array}{l}\mathsf{if}\ \mathsf{today} > \mathsf{pubDate}\\ \mathsf{then}\ \mathsf{open}\ \mathsf{published}\\ \qquad \mathsf{in}\ \mathsf{applyToList}\ \mathsf{publish}\ \mathsf{acceptedPapers}\\ \mathsf{else}\ ()\end{array}$$

Alternatively, we can check published in publish, leaving the client publishAll as is. In this case, publish would be defined as follows:

$$\lambda x.\ \begin{array}{l}\mathsf{if}\ \mathsf{today} > \mathsf{pubDate}\\ \mathsf{then}\ \mathsf{open}\ \mathsf{published}\\ \qquad \mathsf{in}\ \mathsf{out} := x\\ \mathsf{else}\ ()\end{array} : \mathcal{N}^{\{\{\mathsf{published}\}\Rightarrow\mathsf{pub}\}} \xrightarrow{\emptyset,\{\emptyset\Rightarrow\mathsf{pub}\}} \mathsf{unit}^{\bot}$$

As we can see from the type, this function can be invoked without any open locks because it opens the relevant lock itself. Note that the type system does not prevent the programmer from opening

published without checking that the publication date has passed, but by forcing the programmer to add code to open the lock, our type system prevents accidental declassification and forces the programmer to think about allowed information flows.

4 A COMPOSITIONAL DEFINITION OF SECURITY

We now turn to the main contribution of this paper: a compositional definition of security for λ-WHR. Intuitively, we want our security definition to say that programs provide noninterference until a *relevant declassification*, i.e., an allowed declassification that gives new information to an attacker. More precisely, if we have two low-equivalent runs — i.e., executions the attacker cannot tell apart — then every reduction step in either execution that is not a relevant declassification should preserve that low equivalence. However, once a relevant declassification has taken place the runs can diverge arbitrarily as that revealed information percolates through the execution. For example in the program $(\lambda x.\,\mathsf{if}\ !\,l\ \mathsf{then}\ f\ ()\ \mathsf{else}\ g\ ())\ (\mathsf{open}\ \sigma_h\ \mathsf{in}\ l := !\,h)$, which first declassifies the value stored in h to l and then branches on that value, either f or g gets executed depending on the declassified value.

In effect, the description above captures a step-by-step version of the requirements posed by Flow-Lock security [Broberg and Sands 2009], from which we take inspiration. In order to build a compositional definition of security that matches this intuition we use *logical relations*, a standard tool in the semantics of programming languages. We are not the first to use logical relations to define security compositionally. Indeed, our semantics is built on that of FG [Rajani and Garg 2018], which defines noninterference (without declassification) compositionally using logical relations. However, as far as we are aware ours is the first model of where declassification using logical relations. In fact, we believe this to be the first model of an intensional information flow property using logical relations.

Logical relations provide models for programming languages by associating each type with a relation on terms. Like most logical relations for information-flow security, our primary logical relation is *binary*, meaning that its constituting relations each relate pairs of terms. Intuitively, two terms are related if they are indistinguishable to a particular attacker $\mathcal{A}$ *until a relevant declassification occurs*. We also use a *unary* relation which tells us that programs do not perform low-visible effects in a high context. In other words, the unary relation is the semantic analogue of a *confinement* lemma. Since our unary relation is adapted nearly directly from FG—though changed to a small-step relation—we postpone the description of it to Section 4.4, where we will only cover the essentials.

Our binary relation—like most logical relations, including FG's—gives two interpretations of types: a *value* relation and an *expression* relation. The value relation is written $[\![\tau]\!]^{\mathcal{A}}_{\mathcal{V}}$, where $\mathcal{A}$ is an attacker (and $\mathcal{V}$ stands for "value"). Intuitively, the value relation tells us when two values are indistinguishable to an attacker. Importantly, if the values are functions, we check that calling those functions results in indistinguishable programs. This check provides the compositionality we set out to achieve.

The expression relation is written $[\![\tau]\!]^{\mathcal{A}}_{\mathsf{E}}$. Again, $\mathcal{A}$ is an attacker while E stands for "expression." Intuitively, the expression relation checks that two expressions evaluate to indistinguishable values and that information is never leaked to an adversary during the evaluation. If evaluating the expressions results in a relevant declassification, i.e., a pair of declassification steps that leads to diverging behavior, we no longer make any requirements on the expression. Importantly, this does not mean that our security definition does not detect information leaks beyond a fixed declassification point in the program. Our security definition considers *all pairs* of executions. Given any declassification point in a program, even if one pair of executions results in a relevant declassification because the two executions declassify different data at that point, in other pairs of

executions the two executions may declassify the same data at that point making the declassification non-relevant in those pairs of executions, and the security definition's requirements would continue to be imposed past that point for those pairs.

Attackers. In order to reason about what an attacker can see, we need to define attackers. Usually in information-flow–security definitions, an attacker is an information-flow label. However, λ-WHR uses Flow-Locks policies, which may contain more than one actor, rather than information-flow labels. In order to maintain the intuition of a single attacker, we follow Flow Locks [Broberg and Sands 2009, 2010] in defining an attacker $\mathcal{A}$ as a policy $\{\Sigma^{\mathcal{A}} \Rightarrow \mathfrak{a}\}$. That is, an attacker $\mathcal{A}$ is a pair $(\mathfrak{a}, \Sigma^{\mathcal{A}})$, of an actor $\mathfrak{a}$ and a lock set $\Sigma^{\mathcal{A}}$ called the *capability* of $\mathcal{A}$.

We can intuitively think of $\Sigma^{\mathcal{A}}$ as the set of locks that the attacker can open forcibly. Accordingly, a policy $\mathfrak{p}$ is visible to an attacker when $\mathfrak{p} \sqsubseteq \{\Sigma^{\mathcal{A}} \Rightarrow \mathfrak{a}\}$ (abbreviated $\mathfrak{p} \sqsubseteq \mathcal{A}$). This occurs if $\mathfrak{a}$ may learn information protected by $\mathfrak{p}$ under the assumption that the locks in $\Sigma^{\mathcal{A}}$ are open.

State indistinguishability. We want two expressions to be indistinguishable to an attacker $\mathcal{A}$ if, when started in $\mathcal{A}$-indistinguishable states, the two traces of observations produced by their execution and the final reduced values are also $\mathcal{A}$-indistinguishable. This requires us to determine when two states are $\mathcal{A}$-indistinguishable.

If λ-WHR had no way to allocate new memory cells, then we could simply say that two states are indistinguishable if every state cell contained indistinguishable values. More precisely, if θ was a state environment, and S_1, S_2 were two states of type θ, then S_1 and S_2 would be $\mathcal{A}$-indistinguishable if $(S_1(\mathfrak{l}), S_2(\mathfrak{l})) \in [\![\theta(\mathfrak{l})]\!]^{\mathcal{A}}_{\mathcal{V}}$ for every location $\mathfrak{l}$.

However, since λ-WHR has nondeterministic memory allocation, things get more complicated. For example, suppose we wish to prove that $e_1 \triangleq \mathsf{let}\ x = \mathsf{new}\ (1, \mathcal{N}^{\bot})\ \mathsf{in}\ !x$ is related to itself. The two copies of e_1 may allocate different locations, say $\mathfrak{l}_1$ and $\mathfrak{l}_2$, at the subexpression $\mathsf{new}\ (1, \mathcal{N}^{\bot})$. Consequently, one reduction step later, we would want to relate $!\,\mathfrak{l}_1$ and $!\,\mathfrak{l}_2$ in the starting states $\mathfrak{l}_1 \mapsto 1$ and $\mathfrak{l}_2 \mapsto 1$. These two states do not have the same locations, yet, somehow, we wish to say that they are indistinguishable.

In order to handle this, we adopt a standard solution: we allow each run to have different state environments, as long as there is a bijection between corresponding locations in the two runs. More precisely, we index all of our relations with Kripke *worlds* [Ahmed et al. 2009] (usually denoted by W, W', etc.), which are triples of the form $(\theta_1, \theta_2, \beta)$ where θ_1 and θ_2 are state environments and β is a bijection between their domains. The bijection β tells us how to match up locations between the runs. In our example above, when we get to relating $!\,\mathfrak{l}_1$ and $!\,\mathfrak{l}_2$, the bijection β would match $\mathfrak{l}_1$ and $\mathfrak{l}_2$, while θ_1 and θ_2 would assign $\mathfrak{l}_1$ and $\mathfrak{l}_2$, respectively, $\mathcal{N}^{\bot}$. Clearly, the expressions $!\,\mathfrak{l}_1$ and $!\,\mathfrak{l}_2$ will both evaluate to the same value in any two states that are indistinguishable up to this β, which justifies their relatedness.

So far, this definition implicitly assumes that when the program on the left-hand side of the relation allocates memory, so will the program on the right-hand side. After all, a bijection between two sets witnesses that they have the same size. This assumption does not hold, however, since secrets may determine whether a location is allocated. To see this, consider proving that $e_2 \triangleq \mathsf{if}\ !h\ \mathsf{then}\ \mathsf{let}\ x = \mathsf{new}\ (3, \mathcal{N}^{\top})\ \mathsf{in}\ !x\ \mathsf{else}\ 3$ is related to itself, where h is a high-security reference. Since h is high security, in one run it may store true while in the other run it stores false. Thus, after two steps of computation, we need to relate $\mathsf{let}\ x = \mathsf{new}\ (3, \mathcal{N}^{\top})\ \mathsf{in}\ !x$ with 3. On the left-hand side we allocate memory that we do not on the right. However, since this is a private memory cell, intuitively the adversary cannot see that allocation.

In order to accommodate situations like the one above, where differing numbers of private allocations occur in two runs, we relax the requirement that β be a bijection. Instead, we only

require that it be a *partial* bijection; that is, an injective partial function. We consider those locations matched by β to be "the same," while those not related to any location by β correspond to situations like that in e_2. Moreover, our binary relation requires that attacker-visible locations allocated in both branches be matched by β. Combining these facts, only private locations will fail to be related to any other location.

Formally, the $\mathcal{A}$-indistinguishability of states S_1 and S_2 at world W is denoted $(S_1, S_2, \boxed{m}) \overset{\mathcal{A}}{\triangleright} (W)$, and is defined in Definition 4.1. The part in gray, called the step index [Appel and McAllester 2001], can be ignored by most readers. We discuss it later.

Definition 4.1 (World-Indexed State Indistinguishability)**.**

We say that states S_1 and S_2 are *$\mathcal{A}$-indistinguishable in world W*, written $(S_1, S_2, \boxed{m}) \overset{\mathcal{A}}{\triangleright} W$, exactly when:

$$
\begin{array}{ll}
& \mathrm{dom}(W.\theta_i) \subseteq \mathrm{dom}(S_i) \\
\wedge & \forall l \in \mathrm{dom}(W.\theta_i).\, W.\theta_i(l) = \mathrm{type}(S_i, l) \\
\wedge & \beta \subseteq \mathrm{dom}(W.\theta_1) \times \mathrm{dom}(W.\theta_2) \\
\wedge & \left(\begin{array}{l} \forall (l_1, l_2) \in W.\beta. \\ \quad W.\theta_1(l_1) = W.\theta_2(l_2) \wedge \\ \quad (S_1(l_1), S_2(l_2), W, \boxed{m}) \in [\![W.\theta_1(l_1)]\!]^{\mathcal{A}}_{\mathcal{V}} \end{array} \right)
\end{array}
$$

The most important part of this definition is its last line, which says that for all locations l_1 and l_2 matched by β, the values in those locations must be indistinguishable at their common type. Again, we represent indistinguishability of values using the value interpretation $[\![W.\theta_1(l_1)]\!]^{\mathcal{A}}_{\mathcal{V}}$, but the astute reader will note that this interpretation seems to relate the *quadruple* $(S_1(l), S_2(l), W, \boxed{m})$ rather than the pair $(S_1(l), S_2(l))$. This is because the definition of our value interpretation $[\![W.\theta_1(l_1)]\!]^{\mathcal{A}}_{\mathcal{V}}$ also includes worlds. These worlds are necessary because function values may contain free locations, which could differ in the two runs. Roughly, $(v_1, v_2, W, \boxed{m}) \in [\![\tau]\!]^{\mathcal{A}}_{\mathcal{V}}$ means that v_1 and v_2 are $\mathcal{A}$-indistinguishable, up to renaming of locations under $W.\beta$. The full definition of $[\![\tau]\!]^{\mathcal{A}}_{\mathcal{V}}$ can be found in Section 4.2. Our notions of worlds and state indistinguishability presented above correspond to the ones in FG [Rajani and Garg 2018].

4.1 The Expression Relation

We now begin to explain the formal definition of indistinguishability through our logical relation. The complete definition of the expression relation can be found in Figure 4 (on the next page), while the value relation can be found in Figure 6 (in Section 4.2).

In order to explain the expression relation, which looks quite complicated, we break the explanation up into three parts. In the first part, we explain the parts of the relation relevant to noninterference. This part of the relation acts similarly to the relation for FG. However, in order to accommodate declassification, we have changed the presentation of the relation to use small-step operational semantics, rather than big-step. Then, we describe the parts of our relation that are relevant to declassification. Finally, we touch briefly on our use of step indexing [Appel and McAllester 2001], which is a technique for ensuring that logical relations are well defined in the face of higher-order state. In order to make the following sections easier to read, we have highlighted the parts of the relation related to declassification in yellow and those related to step indexing in gray. Thus, the first part of this subsection focuses on the *unhighlighted* parts of the expression relation. Then, we focus on the parts in yellow. Finally, we touch briefly on the parts in gray.

Relating expressions without declassification. The expression relation $[\![\cdot]\!]^{\mathcal{A}}_{\mathsf{E}}$ is a union of two relations: a relation $[\![\cdot]\!]^{\mathcal{A}}_{\mathsf{E}\,\beta}$ (from here on β-relation in the text) when at least one of the expressions

$$\llbracket\tau\rrbracket^{\mathcal{A}}_{\mathsf{E}} \triangleq \llbracket\tau\rrbracket^{\mathcal{A}}_{\mathsf{E}\beta} \cup \left\{ (v, v', W, \Sigma_{c_1}, \Sigma_{c_2}, m) \;\middle|\; (v, v', W, m) \in \llbracket\tau\rrbracket^{\mathcal{A}}_{\mathcal{V}} \right\} \text{ where:}$$

$$\llbracket\tau\rrbracket^{\mathcal{A}}_{\mathsf{E}\beta} \triangleq \left\{ (e_1, e_2, W, \Sigma, \Sigma', m) \;\middle|\; \begin{array}{l} \forall \Sigma_1, \Sigma_2, \Sigma \subseteq \Sigma_1 \wedge \Sigma' \subseteq \Sigma_2 \rightarrow \forall W', m', S_1, S_2. \\ m' < m \wedge W' \sqsupseteq W \wedge (S_1, S_2, m') \overset{\mathcal{A}}{\triangleright} W' \rightarrow (e_1, e_2) \in \\ \left(\begin{array}{l} C_{\text{PAR}} = \left\{ (e_1, e_2) \;\middle|\; \begin{array}{l} e_1 \notin \mathcal{V} \wedge e_2 \notin \mathcal{V} \wedge \\ \forall e_1', S_1', \Sigma_1', \omega, e_2', \Sigma_2', S_2', \omega'. \\ \Sigma_1 \vdash e_1, S_1 \xRightarrow{\omega; \Sigma_1'} e_1', S_1' \wedge \\ \Sigma_2 \vdash e_2, S_2 \xRightarrow{\omega'; \Sigma_2'} e_2', S_2' \rightarrow \\ \left[\begin{array}{l} \exists \mathfrak{l}, \nu, \tau', \mathfrak{l}', \nu'. \\ \omega = \mathfrak{l}_{\tau'}(\nu) \wedge \omega' = \mathfrak{l}'_{\tau'}(\nu') \wedge \\ \text{pol}(\omega) \sqsubseteq \mathcal{A} \wedge \text{pol}(\omega') \sqsubseteq \mathcal{A} \wedge \\ (\nu, \nu', W', m') \notin \llbracket\tau'\rrbracket^{\mathcal{A}}_{\mathcal{V}} \wedge \\ \Sigma_1' \not\subseteq \mathcal{A} \wedge \Sigma_2' \not\subseteq \mathcal{A} \end{array} \right] \vee \\ \exists W'' . W'' \sqsupseteq W' \wedge (S_1', S_2', m') \overset{\mathcal{A}}{\triangleright} (W'') \wedge \\ \omega \cong^{\mathcal{A}}_{(W'', m')} \omega' \wedge \\ (e_1', e_2', W'', \Sigma, \Sigma', m') \in \llbracket\tau\rrbracket^{\mathcal{A}}_{\mathsf{E}} \end{array} \right\} \cup \\ C_{\text{L}} = \left\{ (e_1, e_2) \;\middle|\; \begin{array}{l} e_1 \notin \mathcal{V} \wedge \forall e_1', S_1', \Sigma_1', \omega. \\ \Sigma_1 \vdash e_1, S_1 \xRightarrow{\omega; \Sigma_1'} e_1', S_1' \rightarrow \text{pol}(\omega) \not\sqsubseteq \mathcal{A} \wedge \\ (\exists W'' . W'' \sqsupseteq W' \wedge (S_1', S_2, m') \overset{\mathcal{A}}{\triangleright} (W'') \wedge \\ (e_1', e_2, W'', \Sigma, \Sigma', m') \in \llbracket\tau\rrbracket^{\mathcal{A}}_{\mathsf{E}}) \end{array} \right\} \cup \\ C_{\text{R}} = \left\{ (e_1, e_2) \;\middle|\; \begin{array}{l} e_2 \notin \mathcal{V} \wedge \forall e_2', S_2', \Sigma_2', \omega. \\ \Sigma_2 \vdash e_2, S_2 \xRightarrow{\omega; \Sigma_2'} e_2', S_2' \rightarrow \text{pol}(\omega) \not\sqsubseteq \mathcal{A} \wedge \\ (\exists W'' . W'' \sqsupseteq W' \wedge (S_1, S_2', m') \overset{\mathcal{A}}{\triangleright} W'' \wedge \\ (e_1, e_2', W'', \Sigma, \Sigma', m') \in \llbracket\tau\rrbracket^{\mathcal{A}}_{\mathsf{E}}) \end{array} \right\} \end{array} \right) \end{array} \right\}$$

Fig. 4. Binary Expression Relation

can take a step; and the value relation, when both of the expressions are values. Thus, the logical relation enforces progress: stuck programs are never in the expression relation. We describe the value relation in Section 4.2; the rest of this subsection will focus on the β relation.

In the β relation, we start by picking an arbitrary *future world* $W' \sqsupseteq W$ and two arbitrary states S_1 and S_2 which are $\mathcal{A}$-indistinguishable in W'. Intuitively, the choice of future world says that the relation must continue to hold even as the state evolves. This is an important ingredient of compositionality: functions that are considered equivalent when they are defined should continue to be equivalent when they are used, even if more locations have been allocated. By ensuring that our relation holds in an arbitrary future world, we make our relation *monotonic*: if e_1 and e_2 are related at some world W, then e_1 and e_2 remain related in any future world $W' \sqsupseteq W$. In the rest of this subsection we will see that ensuring such monotonicity properties is an important goal throughout the definition of the β relation, and in the definition of the logical relation as a whole.

Definition 4.2 (Future World [Rajani and Garg 2018])**.** Formally, we say W' *is a future world of* W (written $W' \sqsupseteq W$) whenever $W.\theta_i \subseteq W'.\theta_i$ for $i = 1, 2$ and $W.\beta \subseteq W'.\beta$. Thus, all allocated locations remain allocated with the same type, and locations which are matched across the two runs continue to be matched.

$$\frac{\mathrm{pol}(\omega) \not\sqsubseteq \mathcal{A} \qquad \mathrm{pol}(\omega') \not\sqsubseteq \mathcal{A}}{\omega \approx^{\mathcal{A}}_{(W,\,\mathfrak{m})} \omega'}\ \textsc{high} \qquad \frac{(v, v', W, \mathfrak{m}) \in [\![\tau]\!]^{\mathcal{A}}_{\mathcal{V}} \qquad (\mathfrak{l}, \mathfrak{l}') \in W.\beta}{\mathfrak{l}_\tau(v) \approx^{\mathcal{A}}_{(W,\,\mathfrak{m})} \mathfrak{l}'_\tau(v')}\ \textsc{extend-}\tau$$

Fig. 5. Observational Indistinguishability

Intuitively, our relation says that the attacker $\mathcal{A}$ cannot distinguish the executions of e_1 and e_2. We thus require that those execution steps of e_1 and e_2 that the attacker can observe be matched one-to-one. However, we allow e_1 and e_2 to take any number of invisible steps with no match on the other side. In order to formalize this intuition, we split the β relation into three parts: $C_{\textsc{Par}}$, C_L, and C_R. The $C_{\textsc{Par}}$ relation corresponds to the requirement that observable steps be matched one-to-one. The C_L and C_R relations correspond to e_1 and e_2, respectively, taking unmatched invisible steps.

The $C_{\textsc{Par}}$ subrelation requires that neither e_1 nor e_2 be values, since values cannot take any steps. Then, any steps of e_1 and e_2 result in new expressions e_1' and e_2' and new states S_1' and S_2'. In order for e_1 and e_2 to be indistinguishable, the adversary should not be able to distinguish S_1' and S_2'. However, the step we just took may have allocated more locations, so we cannot require that S_1' and S_2' be indistinguishable in W'. Thus, we require that there exist some future world $W'' \sqsupseteq W'$ such that S_1' and S_2' are indistinguishable.

Next we require that the observations ω and ω' generated by the steps above be indistinguishable to the attacker $\mathcal{A}$. Most security definitions just require that observations are equal. Because we are in the higher-order setting and wish to get a compositional definition, we need to be a bit more careful. We formally define observation indistinguishability in Figure 5. Intuitively, two observations are indistinguishable if the attacker cannot see either of them, as the rule HIGH expresses, or if they look the same to the attacker. If the attacker can see them, both observations are writes—i.e., we are trying to prove $\mathfrak{l}_\tau(v) \approx^{\mathcal{A}}_{(W,\,\mathfrak{m})} \mathfrak{l}_\tau(v')$. In this case EXTEND-$\tau$ applies. To ensure that both writes look the same to the attacker, the locations $\mathfrak{l}$ and $\mathfrak{l}'$ must be related by $W.\beta$, since they must represent "the same" location between the two runs. Additionally, v and v' must be indistinguishable to $\mathcal{A}$.

Returning to the definition of $C_{\textsc{Par}}$ in Figure 4, we finally require that the new programs e_1' and e_2' are indistinguishable in this new world W''. Thus, any further steps they take are again subject to the requirements of the β relation, ensuring that security holds for more than one step.

We now look at the left-hand relation C_L; the right-hand relation C_R is the dual, so we avoid explicitly describing it. Again, e_1 should not be a value, since values are not allowed to take steps. However, e_2 may now be a value, since we do not consider any steps it may take. In order to be in the left-hand relation, we require that e_1 only be able to take steps which are *not* visible to the attacker $\mathcal{A}$. In other words, every step that e_1 can take should yield an observation ω such that $\mathrm{pol}(\omega) \not\sqsubseteq \mathcal{A}$. (Note that ϵ is one such observation.) When e_1 takes such a step, yielding a new expression e_1' and a new state S_1', intuitively the attacker should not be able to tell that e_1 took a step. To formalize this, we require that S_1' is indistinguishable from S_2 at some future world $W'' \sqsupseteq W'$. We also require that e_1' be indistinguishable from e_2, again ensuring that security holds for more than one step.

Note that our relation enforces a *termination insensitive* indistinguishability relation: we consider a program e_t that terminates to be indistinguishable from another program e' that continues executing after e_t's termination but has been indistinguishable from e_t up until this point. We choose a termination insensitive definition because our type system only enforces termination insensitive security. Termination-sensitive type systems are notoriously restrictive and difficult to work with. We could turn our logical relation into a termination-sensitive definition by enforcing in

C_{PAR} that any step in the left execution is matched by a step in the right execution and by enforcing that C_L and C_R can only be used a finite number of times before another clause is used.

Handling where declassification. We now revisit the definition of the expression relation, considering the parts in yellow which correspond to reasoning about declassification. Most of the changes involve carrying around two sets of open locks Σ_1 and Σ_2, one for each expression. Notably, the value relation does not require us to reason about open locks, since the only way values can contain declassifications is inside functions, which carry their open-lock sets with them. Thus, values are related at all pairs of lock sets if they are related at all. With this insight, we can again focus our attention on the β relation.

We begin by picking supersets Σ_1 and Σ_2 of our open lock sets Σ and Σ'. This corresponds to the intuition that a term that is deemed secure starting from a certain set of open locks should also be deemed secure starting from a larger set of open locks (since having more open locks can only make more, not fewer, declassifications policy compliant). Again, this leads to a *monotonicity* property: if two terms are related with lock sets Σ and Σ', then they are related with any supersets of those lock sets.

Once we consider lock sets, we must index every step of evaluation with the current open locks and with the active lock set of the reduction. In C_L and C_R, these play no further role in the logical relation; since lock sets are static, the original lock sets are reused in recursive calls. However, in C_{PAR} there is an additional clause which needs explanation.

The key difference between noninterference and declassification is that two executions of noninterfering programs are *always* indistinguishable, while **executions of programs with declassification may stop being indistinguishable at some point.** To see this, consider two executions of a program that differ only in secret inputs. Once a secret is declassified, the two executions now differ in both public and secret data. Thus, they may no longer behave the same and may return distinguishable outputs. As long as the declassification that caused the executions to get out of sync was allowed according to policy, this is not a security breach.

We call a declassification that can cause the two executions to legally get out of sync a *relevant declassification*. Because declassification happens during the execution of the program, rather than as part of a value, the formalization of relevant declassification appears as part of the expression relation. Once a relevant declassification appears, we need to stop requiring that the programs act in lockstep. It is precisely to formalize this that we have based our logical relation on small-step reduction, considering each step in isolation in order to establish a lockstep bisimulation. When formalizing an extensional property like noninterference, this is unnecessary; thus, the structure of our expression relation is significantly different from FG's logical relation.

We formalize relevant declassifications as part of the C_{PAR} clause in the expression relation. There are four major requirements in order for a step to be a relevant declassification, corresponding to the four lines of the large yellow conjunction in the definition of C_{PAR}.

First, the two steps must write to memory, providing something that the adversary could possibly see. We enforce this by requiring that the observations must be write observations.

Second, those writes must be to a location that the attacker can see; we formalize this by requiring that the policy of the observation flow to the attacker. Not only does this make intuitive sense (if the attacker cannot see a location, writing data to that location does not release any information to the attacker), but it is also vital for enforcing security. To see this, consider the program $(\mathsf{open}\ \sigma_h\ \mathsf{in}\ m := h); l := h$ where l is not protected by locks, while m and h are protected by locks σ_m and σ_h, respectively. This program first declassifies h to m, and then *leaks* h to l. Since any attacker which can see neither m nor h sees new information when h is assigned to l, our logical relation correctly rules out this program. If we considered the declassification of m

to $\mathtt{h}$ to be relevant, however, we would stop requiring that the programs be indistinguishable after $\mathtt{m} := \mathtt{h}$ and thus incorrectly deem this program secure.

Third, the attacker must be able to distinguish the observations in the two executions. Because our notion of *in*distinguishability is being related in the logical relation, we formalize distinguishability by requiring that the two values being written are *not* related by the logical relation.

It may be surprising that we do not consider observations that write the same value to different locations (i.e., locations that are not related by the partial bijection β) distinguishable. This has the consequence that our definition deems certain programs insecure that we would consider secure if we did take the locations into account. For example if the location $\mathtt{h}$ is protected by a lock $\sigma_{\mathtt{h}}$ and $\mathtt{l}, \mathtt{l}'$ are publicly visible, then the program $\mathsf{if}\ !\mathtt{h}\ \mathsf{then}\ (\mathsf{open}\ \sigma_{\mathtt{h}}\ \mathsf{in}\ \mathtt{l} := 5)\ \mathsf{else}\ (\mathsf{open}\ \sigma_{\mathtt{h}}\ \mathsf{in}\ \mathtt{l}' := 5)$ is deemed insecure by our definition since the writes of 5 to $\mathtt{l}$ and $\mathtt{l}'$ are not deemed relevant declassification but the program still leaks the value of $\mathtt{h}$.

The obvious question is why our definition of relevant declassification chooses to ignore the relatedness of locations that were written. The reason is there is no world whose β we can use to check the relatedness of the locations correctly across all programs in this situation. If we were to use the world prior to the parallel steps (i.e., W'), a pair of locations that is newly allocated during the step would not be related under this world's β when they should be. On the other hand, we cannot yet extend to a future world, since there will always be a future world in which the newly allocated locations are not related, so any pair of steps allocating new locations would constitute relevant declassification.

We could solve this problem by only taking into account locations already matched by β and ignoring all other locations when checking relatedness, but we choose not to do so here to avoid complicating our definition further. Note that our current choice is "safe" in the sense that it only deems fewer, not more, programs secure. It also does not cause any trouble in proving our type system sound since this situation can only occur if the program has previously branched on a secret value and the type system, like most other information flow type systems, rules out public writes in branches influenced by non-public values (as in the example above). In particular, the example would not type in our – and most existing – type systems.

Fourth, the currently open locks must *allow* declassification, otherwise a difference in observed behavior would just be a leak. We ensure that declassification is allowed by checking that the open locks at the point of the declassification are not all contained in the attacker's capability. Indeed, if the open locks were contained in the attacker's capability, then the attacker should not learn anything new from the declassification so the declassification would not be relevant. On first glance just requiring that some lock is open that is not part of the attacker's capability might seem too weak. Intuitively one might expect that we require that the correct locks needed to declassify the value in question, namely all locks in the *difference* of the lock set that must be open for the attacker to see the value that the step writes and the lock set that must be open for the attacker to see the location that is written, are actually open. Without this condition, the declassification is illegal and should not be considered relevant. The reason it is sound to omit this condition here is that our program security definition (Definition 4.3) requires security against *all* attackers. If a lock σ required for the flow to be legal is not actually open, we can choose a different attacker $\mathcal{A}_\sigma$ whose capability excludes σ but includes all locks needed to view the location (i.e., the observation of the step). For this attacker, the step will not be relevant declassification and our security definition will deem the program insecure.

It might also be surprising that our relevant declassification condition occurs only in the $\mathsf{C}_{\mathrm{PAR}}$ clause. Indeed, one might expect that a program that only declassifies data in one execution also exhibits relevant declassification. However, considering such a situation relevant declassification would incorrectly classify insecure programs as secure. To see this, recall that we start executions

$$
\begin{array}{lll}
\llbracket A^{p} \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ (v, v', W, \boxed{m}) \;\middle|\; p \sqsubseteq \mathcal{A} \wedge (v, v', W, \boxed{m}) \in \llbracket A \rrbracket^{\mathcal{A}}_{\mathcal{V}} \right\} \\
 & & \cup \left\{ (v, v', W, \boxed{m}) \;\middle|\; p \not\sqsubseteq \mathcal{A} \wedge (v, W.\theta_1, \boxed{m}) \in \lceil A \rceil_{\mathcal{V}} \wedge (v', W.\theta_2, \boxed{m}) \in \lceil A \rceil_{\mathcal{V}} \right\} \\
\llbracket \mathsf{unit} \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ ((), (), W, \boxed{m}) \right\} \\
\llbracket \mathbb{N} \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ (n, n, W, \boxed{m}) \;\middle|\; n \in \mathbb{N} \right\} \\
\llbracket \tau_1 \times \tau_2 \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ ((v_1, v_2), (v_1', v_2'), W, \boxed{m}) \;\middle|\; (v_1, v_1', W, \boxed{m}) \in \llbracket \tau_1 \rrbracket^{\mathcal{A}}_{\mathcal{V}} \wedge (v_2, v_2', W, \boxed{m}) \in \llbracket \tau_2 \rrbracket^{\mathcal{A}}_{\mathcal{V}} \right\} \\
\llbracket \tau_1 + \tau_2 \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ (\mathsf{inl}\,(v), \mathsf{inl}\,(v'), W, \boxed{m}) \;\middle|\; (v, v', W, \boxed{m}) \in \llbracket \tau_1 \rrbracket^{\mathcal{A}}_{\mathcal{V}} \right\} \\
 & & \cup \left\{ (\mathsf{inr}\,(v), \mathsf{inr}\,(v'), W, \boxed{m}) \;\middle|\; (v, v', W, \boxed{m}) \in \llbracket \tau_2 \rrbracket^{\mathcal{A}}_{\mathcal{V}} \right\} \\
\llbracket \mathsf{ref}\ \tau \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ (l, l', W, \boxed{m}) \;\middle|\; W.\theta_1(l) = \tau = W.\theta_2(l') \wedge (l, l') \in W.\beta \right\} \\
\llbracket \tau_1 \xrightarrow{\boxed{\Sigma}, \mathsf{pc}} \tau_2 \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ (\lambda x.\, e_1, \lambda x.\, e_2, W, \boxed{m}) \;\middle|\; \begin{array}{l} (\forall W'.W' \sqsupseteq W \to \boxed{\forall m'.m' < m} \to \\ \forall v_1, v_2.(v_1, v_2, W', \boxed{m'}) \in \llbracket \tau_1 \rrbracket^{\mathcal{A}}_{\mathcal{V}} \to \boxed{\forall \Sigma_1, \Sigma_2.\Sigma_1 \supseteq \Sigma \subseteq \Sigma_2} \to \\ (e_1[x \mapsto v_1], e_2[x \mapsto v_2], W', \boxed{\Sigma_1, \Sigma_2}, \boxed{m'}) \in \llbracket \tau_2 \rrbracket^{\mathcal{A}}_{\mathsf{E}}) \wedge \\ (\lambda x.\, e_1, W.\theta_1, \boxed{m}) \in \lceil \tau_1 \xrightarrow{\boxed{\Sigma}, \mathsf{pc}} \tau_2 \rceil_{\mathcal{V}} \wedge \\ (\lambda x.\, e_2, W.\theta_2, \boxed{m}) \in \lceil \tau_1 \xrightarrow{\boxed{\Sigma}, \mathsf{pc}} \tau_2 \rceil_{\mathcal{V}} \end{array} \right\} \\
\llbracket \Gamma \rrbracket^{\mathcal{A}}_{\mathcal{V}} & \triangleq & \left\{ (\gamma, W, \boxed{m}) \;\middle|\; \mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\gamma) \wedge \forall x \in \mathrm{dom}(\Gamma).(\gamma_1(x), \gamma_2(x), W, \boxed{m}) \in \llbracket \Gamma(x) \rrbracket^{\mathcal{A}}_{\mathcal{V}} \right\}
\end{array}
$$

Fig. 6. Binary Value Relation

in indistinguishable states. Thus, if a declassification happens in one execution but not the other, that difference must come from branching on some secret value h. Therefore, if there was a declassification in one execution, the absence of a similar declassification in the other execution would leak information about h.

Note that we use the formalization of relevant declassification in C_{PAR} as part of a disjunction[2]. In other words, every parallel step must show either that the step is a relevant declassification or that the bisimulation continues in lockstep. This matches the key intuition for declassification that we gave earlier: declassification allows us to *stop* requiring that two executions be bisimilar, while noninterference requires bisimilarity for the entirety of the traces.

Step indexing. We now turn to the last part of our logical relation, the part of the relation in gray. Since our logical relation is defined recursively, in order for the relation to be well defined the recursion must be well founded. Like FG [Rajani and Garg 2018], we use a standard trick to ensure wellfoundedness, *step indexing* [see e.g., Ahmed 2004; Ahmed et al. 2009; Appel and McAllester 2001]. Essentially, step indexing forces us to define a well founded recursion over the natural numbers. By only making recursive calls at strictly smaller indices, we can be sure that we do not encounter any circularity. This is similar to the trick commonly used when defining a terminating function: by supplying an extra (natural number) *fuel* parameter to the function, we can ensure that the function terminates.

Compositionality again demands that we make the relation monotonic along decreasing m. We do this in the β relation by choosing an $m' < m$ when we want to show that e_1 and e_2 are related at the step index m. Note that, unlike when we chose a future world or extended lock state, we use *strict* inequality here. This guarantees that we only recursively check the relation for smaller step indices, ensuring wellfoundedness.

4.2 The Value Relation

We now consider the value relation, which can be found in Figure 6. Here is where our relationship to the logical relation of FG [Rajani and Garg 2018] stands out the most. Unlike the expression relation, which needed significant changes to accommodate declassification, the value relation needs mostly cosmetic changes to account for declassification. As before, we highlight these changes in yellow. We also highlight, step indexes in gray again. As before, a reader not familiar with step indexing may ignore them without missing any important insights.

We start with a description of the first clause in Figure 6. This clause defines the value relation for annotated types τ, which always have the form A^p. First, we ensure that the policy p is respected, and then we use the type A to ensure indistinguishability. To ensure that p is respected, we check whether $p \sqsubseteq \mathcal{A}$. If not, then $\mathcal{A}$ cannot see data labeled with p, so we simply check that the two expressions are independently in the unary relation (defined in Section 4.4), which ensures that any functions nested inside the value do not contain implicit leaks. This acts as a confinement lemma. If $p \sqsubseteq \mathcal{A}$ then $\mathcal{A}$ can see data labeled with p, so we check that the values are indistinguishable at the type A.

For the base types unit and $\mathbb{N}$, indistinguishability is equality on values of the type. For product types $\tau_1 \times \tau_2$, we check that both values are pairs. Then, we compare the left and right projections of the values at their respective types. For sum types $\tau_1 + \tau_2$, we need to check that either both values are inl or both values are inr before checking at the respective summand type.

Two values of a reference type ref τ are related if they are locations related by $W.\beta$ and their types match those in W.

Finally, two functions of type $\tau_1 \xrightarrow{\Sigma, pc} \tau_2$ are indistinguishable if, when applied to two indistinguishable arguments, they produce indistinguishable results; this definition is the secret sauce for a compositional definition of security. We thus pick two related—and hence indistinguishable—values of type τ_1, called v_1 and v_2. We then check that the bodies e_1 and e_2 of the functions, after substituting v_1 and v_2 for x, are related in the *expression relation* at τ_2. This check requires that we know the open locks of both expressions. While we do not know what the open locks will be when these functions are called, we know that they will be at least Σ, the lock set given on the function type. We similarly choose a future world and smaller step index, ensuring that our relation is monotonic. Finally, in order to prevent functions that have side effects below pc from being related, we explicitly check that both functions are in the unary relation.

4.3 Definition of Security

Now that we have defined when an attacker $\mathcal{A}$ cannot distinguish two programs up to relevant declassification, we use that definition to define when a program is secure. Intuitively, a program is secure when no attacker can distinguish it from itself when given different secret inputs but the same public inputs, where secret and public respectively refer to values the chosen attacker cannot see and can see. Since λ-WHR is a stateful functional language, secret inputs can be both the contents of secret locations of the initial state *and* substitutions of free variables with secret types. While we have discussed how we ensure that we consider programs with different secret parts of the state, we still need to substitute free variables.

To do so, we first define binary substitutions: a binary substitution $\gamma \triangleq (\gamma_1, \gamma_2)$ is a pair of substitutions γ_1 and γ_2. Given a context Γ, we say that a binary substitution γ is *suitable for* Γ using the logical relation $[\![\Gamma]\!]^{\mathcal{A}}_{\mathcal{V}}$ defined in Figure 6. Intuitively, a pair of substitutions (γ_1, γ_2) is suitable for Γ if they substitute every variable $x : \tau \in \Gamma$ with a pair of values $(\gamma_1(x), \gamma_2(x))$ which

[2]The technical appendix actually uses an equivalent implication instead of a disjunction, as it is more convenient for formal reasoning. We chose to explain the equivalent formulation with a disjunction here for readability's sake.

are related at τ. Since the value relation allows arbitrarily different values for secret types, this correctly allows differing secrets in different runs.

Given a program e and a suitable binary substitution γ, the substituted programs $\gamma_1(e)$ and $\gamma_2(e)$ represent e in two different runs with different secret inputs. In order to deem e secure, we ensure both that these two programs are always indistinguishable from themselves and that $\gamma_1(e)$ and $\gamma_2(e)$ are in the unary logical relation. This guarantees that the type and pc are respected.

Definition 4.3 (Security)**.** Let e be a program with free variables in Γ and locations in θ. Then we say that e is *secure at type* τ *with program counter* pc *under open locks* Σ if, for every attacker $\mathcal{A}$, step index m, world $W \sqsupseteq (\theta, \theta, \mathrm{id}_{\mathrm{dom}(\theta)})$, and binary substitution γ such that $(\gamma, W, m) \in [\![\Gamma]\!]^{\mathcal{A}}_{\mathcal{V}}$,

$$(\gamma_1(e), \gamma_2(e), W, \Sigma, \Sigma, m) \in [\![\tau]\!]^{\mathcal{A}}_{\mathsf{E}} \wedge$$
$$(\gamma_1(e), W.\theta_1, m) \in \lceil\tau\rceil^{\mathsf{pc}}_{\mathsf{E}} \wedge (\gamma_2(e), W.\theta_2, m) \in \lceil\tau\rceil^{\mathsf{pc}}_{\mathsf{E}}$$

To see why this corresponds to our intuitive notion of security, imagine that program e contains an illegal declassification. In particular, imagine that some lock σ is required for the declassification, which happens without σ open. Then e will not be related to itself for the attacker $(a, \{\sigma' \mid \sigma' \neq \sigma\})$.

Also note that this definition finds information leaks that happen after a legitimate declassification. To see why, consider the following program:

$$(\mathsf{open}\ \sigma\ \mathsf{in}\ l := !\,h); l' := !\,h'$$

Assume that h and h' are protected by the lock σ whereas l and l' are public. One might be concerned that the legitimate declassification of h to l might mask the leak of h' to l' later in the program because we stop checking indistinguishability after relevant declassification. However, our security property considers all pairs of states that are compatible with the initial world. This includes, in particular, pairs of states S and S' in which the values of h are the same but the values of h' are different. Assuming that h and h' point to integers, one might for example have $S(h) = 1$, $S(h') = 2$ and $S'(h) = 1$, $S'(h') = 0$. For these S and S', the write of h to l would not constitute relevant declassification because the attacker cannot distinguish the two writes. Hence, we will continue to enforce indistinguishability and find the leak of h' via l'. [3]

λ-WHR types enforce security. Now that we have formally defined when a program is secure, we would like to prove that our type system enforces program security. This corresponds exactly to a standard theorem for logical relations, the *fundamental theorem*. The fundamental theorem says that well typed programs are self-related. Since we defined security as self-relation, in our case the fundamental theorem tells us that all well typed programs are secure. (The fundamental theorem for the unary logical relation, which implies only that the type system enforces the meaning of pc correctly, can be found in Section 4.4.)

Theorem 4.1 (Binary Fundamental Theorem)**.**
If $\Gamma; \Sigma; \theta \vdash_{\mathsf{pc}} e : \tau$, $(\gamma, W, m) \in [\![\Gamma]\!]^{\mathcal{A}}_{\mathcal{V}}$, $W.\theta_1 \sqsupseteq \theta$, $W.\theta_2 \sqsupseteq \theta$, and for every location $l \in \mathrm{dom}(\theta)$, we have $(l, l) \in W.\beta$, then $\forall \mathcal{A}, m.(\gamma_1(e), \gamma_2(e), W, \Sigma, \Sigma, m) \in [\![\tau]\!]^{\mathcal{A}}_{\mathsf{E}}$.

We prove this theorem by induction on the given typing derivation of e. As a consequence of this proof, we can use the logical relation for *semantic typing*, which allows a programmer to compose a program which they have proven secure directly using the logical relation with a program proven secure using the type system, and obtain a guaranteed-secure composed program.

[3] This explanation contains a slight simplification. Technically we *reset* the state after each step. This means we would even disallow a leak of h, even if h was correctly declassified before. Consequently, our definition is not flow sensitive. We will briefly discuss this aspect in the conclusion.

$$
\begin{array}{ll}
\lceil \tau \rceil^{pc}_{E} & \triangleq \lceil \tau \rceil^{pc}_{E\beta} \cup \lceil \tau \rceil_{\mathcal{V}} \\
\lceil \tau \rceil^{pc}_{E\beta} & \triangleq \left\{ (e, \theta, m) \;\middle|\; \begin{array}{l} e \notin \mathcal{V} \wedge \forall S, \theta', m', e', S', \omega, \Sigma, \Sigma' . \theta' \sqsupseteq \theta \wedge m' < m \wedge (S, m') \triangleright (\theta') \rightarrow \\ \Sigma \vdash e, S \xrightarrow{\omega; \Sigma'} e', S' \rightarrow (\omega = \epsilon \vee pc \sqsubseteq pol(\omega)) \wedge \\ (\exists \theta'' . \theta'' \sqsupseteq \theta' \wedge (S', m') \triangleright (\theta'') \wedge (e', \theta'', m') \in \lceil \tau \rceil^{pc}_{E}) \end{array} \right\} \\
\lceil unit \rceil_{\mathcal{V}} & \triangleq \left\{ (e(), \theta, m) \right\} \\
\lceil \mathcal{N} \rceil_{\mathcal{V}} & \triangleq \left\{ (n, \theta, m) \;\middle|\; n \in \mathbb{N} \right\} \\
\lceil \tau_1 \times \tau_2 \rceil_{\mathcal{V}} & \triangleq \left\{ ((v_1, v_2), \theta, m) \;\middle|\; (v_1, \theta, m) \in \lceil \tau_1 \rceil_{\mathcal{V}} \wedge (v_2, \theta, m) \in \lceil \tau_2 \rceil_{\mathcal{V}} \right\} \\
\lceil \tau_1 + \tau_2 \rceil_{\mathcal{V}} & \triangleq \left\{ (inl\,(v), \theta, m) \;\middle|\; (v, \theta, m) \in \lceil \tau_1 \rceil_{\mathcal{V}} \right\} \cup \left\{ (inr\,(v), \theta, m) \;\middle|\; (v, \theta, m) \in \lceil \tau_2 \rceil_{\mathcal{V}} \right\} \\
\lceil \tau_1 \xrightarrow{\Sigma, pc} \tau_2 \rceil_{\mathcal{V}} & \triangleq \left\{ (\lambda x. e, \theta, m) \;\middle|\; \begin{array}{l} \forall \theta', v, m' . \theta' \sqsupseteq \theta \wedge m' < m \wedge \\ (v, \theta', m') \in \lceil \tau_1 \rceil_{\mathcal{V}} \rightarrow (e[x \mapsto v], \theta', m') \in \lceil \tau_2 \rceil^{pc}_{E} \end{array} \right\} \\
\lceil ref\ \tau \rceil_{\mathcal{V}} & \triangleq \left\{ (l, \theta, m) \;\middle|\; \theta(l) = \tau \right\} \\
\lceil A^{p} \rceil_{\mathcal{V}} & \triangleq \lceil A \rceil_{\mathcal{V}} \\
\lceil \Gamma \rceil_{\mathcal{V}} & \triangleq \left\{ (\delta, \theta, m) \;\middle|\; dom(\Gamma) \subseteq dom(\delta) \wedge \forall x \in dom(\Gamma) . (\delta(x), \theta, m) \in \lceil \Gamma(x) \rceil_{\mathcal{V}} \right\}
\end{array}
$$

Fig. 7. Unary Relation

4.4 The Unary Relation

Our unary relation largely follows the work of Rajani and Garg [2018]. Here, we discuss the relation briefly. For readers familiar with FG, the differences from FG's unary logical relation are that: (1) our relation is based on our small-step semantics while FG's relation uses a big-step semantics, and (2) our relation adds locks sets at relevant places to match the revised syntax of types and the operational semantics, but this addition is only cosmetic since these sets are irrelevant for the unary relation. Other than this, the unary relations are essentially the same.

The definition of the unary relation can be found in Figure 7. Again, everything related to step-indexes is highlighted in gray. The additions for lock sets are marked in yellow.

Much like the binary relation, the unary relation consists of an expression relation and a value relation. Both relations are step- and world-indexed but the worlds are much simpler since we consider only one run at a time. The simplified worlds are just state environments—θ in our notation. A state environment θ' is a future world of θ, written $\theta' \sqsupseteq \theta$, if $\theta \subseteq \theta'$.

Definition 4.4 (State Well-Formedness [Rajani and Garg 2018])**.** We say that the state S is well-formed in the state environment θ at step index m, written $(S, m) \triangleright (\theta)$, if:

$$
\begin{array}{l}
\quad dom(\theta) \subseteq dom(S) \\
\wedge\ (\forall l \in dom(\theta) . (S(l), \theta, m) \in \lceil \theta(l) \rceil_{\mathcal{V}}) \\
\wedge\ \forall l \in dom(\theta) . \theta(l) = type(S, l)
\end{array}
$$

The expression relation is again a union of a β relation and the value relation. The β relation, and thus the expression relation as a whole, is indexed by a program-counter label pc. The β relation acts primarily as a confinement lemma, ensuring that any observation made during the execution of an expression has a policy above pc, i.e., $pc \sqsubseteq pol(\omega)$. A program e is in the relation if for any starting state S, e can take a step to some e' and a new state S', where the new state is correct for some future world θ' and the observation produced by the step, if non-trivial, has a policy above pc.

The value relation ignores the policy annotation on the type and ensures that values have the "right" shapes for their respective types. Thus, values of type unit and $\mathcal{N}$ are simply $()$ and natural

numbers, respectively, sums and products recursively check constituting values against constituting types, and values of a reference type $\mathsf{ref}\ \tau$ are locations that have type τ in the current world. A function is in the relation of its type if it yields an expression of the result type when applied to any value of the input type. Again, this makes the unary type relation compositional.

We prove the fundamental theorem for the unary logical relation as well.

Theorem 4.2 (Unary Fundamental Theorem)**.**
If $\Gamma; \Sigma; \theta \vdash_{\mathsf{pc}} \bar{e} : \tau$ and $\theta' \sqsupseteq \theta$ and $(\delta, \theta', m) \in \lceil\Gamma\rceil_{\mathcal{V}}$, then $(\delta(\bar{e}), \theta', m) \in \lceil\tau\rceil_{\mathsf{E}}^{\mathsf{pc}}$.

5 COMPARISON TO FLOW-LOCK SECURITY

We use the declassification mechanism of Flow Locks [Broberg and Sands 2009] and our security definition is inspired by Flow-Lock security. It is therefore reasonable to ask if the two definitions are also semantically related and, if so, whether this relationship can be made mathematically precise.

Upfront, the two definitions are *not equivalent*. Our definition is motivated by the desire for a compositional security property, which means that we consider some functions to be insecure. Since Flow Locks considers all functions secure, our security definition is more restrictive. However, our security definition *implies* Flow-Lock security.

In order to compare our guarantees to those of Flow Locks, we have to translate the security definitions of Flow Locks to λ-WHR. Because our language is more complex than that of Flow Locks, this translation reasonably extends the definitions of Flow Locks to cover our language constructs. We then use our logical relation to show that secure λ-WHR programs are also Flow-Lock secure. However, even the extended Flow-Lock security definition does not apply to programs with higher-order state, so we must restrict the formal theorem relating the two definitions (Theorem 5.1) to λ-WHR programs without higher-order state.

*Flow Locks defines security using **attacker knowledge**.* Here is how Flow-Lock security works intuitively. Consider two executions that start in attacker-indistinguishable states and have attacker-indistinguishable observations up to some point in the execution. Then, at the next step, either there is a relevant declassification or the attacker *learns nothing new about the initial state*, i.e., the observations remain attacker-indistinguishable. The rest of this section formalizes this intuitive understanding of Flow-Lock security and explains the relation between that and our security as defined in Section 4.

We already have a way to determine when two states are indistinguishable to an attacker $\mathcal{A}$ up to relevant declassification, given via the logical relation in Definition 4.1. However, in order to match the development of Flow Locks, we need to be more precise. We consider states S_1 and S_2 with the same set of low locations (i.e., locations which $\mathcal{A}$ can see). More precisely, we assume that we have some state environment θ which agrees with S_1 and S_2 on the types of those low locations, so $\mathrm{dom}(\theta)$ is the set of low locations in S_1 and S_2. Then, we say that S_1 and S_2 are *low equivalent*, written $S \approx_{\mathcal{A}} S'$, if for every step index m, $(S_1, S_2, m) \overset{\mathcal{A}}{\triangleright} (\theta, \theta, \mathrm{id}_{\mathrm{dom}(\theta)})$. Note that this definition imposes no requirements on the high locations.

We refer to lists of observations which result from consecutive steps as *traces*. For example, if e steps to e' generating observation ω and e' steps to e'' generating observation ω', then the resulting trace is ω, ω'. We further write $\Sigma \vdash e, S \overset{\Omega;\Sigma'}{\Longrightarrow}{}^{*}_{\mathcal{A}} e', S'$ when e reduces to e' in state S and lock state Σ over several steps, resulting in new state S' with active lock set Σ' in the final step, allowing attacker $\mathcal{A}$ to make observations Ω. Note that such traces only contain attacker visible observations, so in particularly they do not explicitly contain ϵ-observations. We lift observation indistinguishability (Figure 5) to trace indistinguishability pointwise.

Finally, we can define the *knowledge* of an attacker [Askarov and Sabelfeld 2007a]. Intuitively, if an attacker observes the execution of some expression e, then the knowledge of the attacker is the set of possible states that the execution could have started in. Assume that the starting state is low equivalent to some state L which contains only low locations. Further assume that a state environment θ_L agrees with L on the types of every location. We can then define attacker $\mathcal{A}$'s knowledge as follows:

$$k_{\mathcal{A}}(e; \Omega; L; \Sigma) \triangleq \left\{ S \;\middle|\; \begin{array}{l} S \approx_{\mathcal{A}} L \\ \wedge\ \Sigma \vdash e, S \xrightarrow{\Omega';\Sigma'}{}^{*}_{\mathcal{A}}\ e', S' \\ \wedge\ \exists W \supseteq (\theta_L, \theta_L, \mathrm{id}_{\mathrm{dom}(L)}) \\ \quad \text{such that } \forall m.\ \Omega \approx^{\mathcal{A}}_{(W,m)} \Omega' \end{array} \right\}$$

Note that we use the logical relations based indistinguishability on observations here. This definition coincides with the definition of indistinguishability used in Flow locks [Broberg and Sands 2009] for those observations present in [Broberg and Sands 2009]. Additionally, our definition extends to other types in our language in a way that is consistent with our treatment of indistinguishability.

The knowledge definition above is intuitive, but it behaves strangely if there is higher-order state, which allows storing nonterminating functions in memory. To see this, consider the program $l := !\,l$ where l is a low location. Since this assigns a low value to a low location, it should intuitively be safe. Moreover, our logical relation correctly deems this program secure. However, knowledge-based security with the knowledge definition above would consider this program insecure.

This discrepancy comes from an assumption implicit in the definition of knowledge-based security: that the indistinguishability relation used on observations is an equivalence relation. However, because our logical relation represents termination-*in*sensitive indistinguishability, diverging functions are related to many functions that are not necessarily related to each other. For example, every silently diverging function is indistinguishable from every other function. So the knowledge (set) of $l := !\,l$ contains $\lambda x.5$ when starting in a state that maps l to a silently nonterminating function f, but not when starting in a state that maps l to $\lambda x.6$, even though f and $\lambda x.6$ are indistinguishable.

If we had defined a termination-sensitive logical relation and additionally proved it an equivalence relation[4], we would not have run into this limitation of knowledge-based definitions. However, we choose to stay with a termination-insensitive logical relation because: (a) It is standard in literature. In particular, the two security definitions of Flow Locks in [Broberg and Sands 2009] are termination-insensitive, and (b) Any type system that is sound with respect to a termination-sensitive logical relation must also establish program termination, which is difficult in practice.

Hence, to get a meaningful comparison between Flow-Lock security and our security definition, we only consider programs with lower-order state, i.e., programs that do not store functions in memory. This eliminates the problem outlined above. This setting is still more general than that of Flow-Lock security, which does not consider any higher order programs, excluding even those higher-order programs that never store functions in memory. With this restriction in place, we can prove that our lifting of Flow-Lock security Broberg and Sands [2009, 2010] is implied by our notion of security (for λ-WHR).

Theorem 5.1 (Our security implies Flow-Lock security [sketch only; see the technical appendix for details])**.** Let e be a program that does not use higher-order state, θ an arbitrary state environment, L a low state, and θ^L the extension of θ so that it agrees with L on the types of all locations in L. Imagine that $(e, e, (\theta^L, \theta^L, \mathrm{id}_{\mathrm{dom}(\theta^L)}), \Sigma, \Sigma, m) \in [\![\tau]\!]^{\mathcal{A}}_E$ at all step indexes m and that an attacker $\mathcal{A}$ executes e in two starting states that $\mathcal{A}$ cannot distinguish from L. This results in two $\mathcal{A}$-visible traces, Ω, ω and Ω', ω'. Let Σ' and Σ'' be the active lock sets in the steps producing ω and ω', respectively.

[4]Proving a step-indexed logical relation is an equivalence relation is generally very difficult even if the relation is termination sensitive. Methods to do this in the general case are not known yet.

If the two initial traces are indistinguishable to $\mathcal{A}$—i.e., if $\forall \mathfrak{m}.\ \Omega \approx^{\mathcal{A}}_{((\theta,\theta,\mathrm{id}_{\mathrm{dom}(\theta)}),\mathfrak{m})} \Omega'$—and either $\Sigma' \sqsubseteq \mathcal{A}$ or $\Sigma'' \sqsubseteq \mathcal{A}$, then $\mathrm{k}_{\mathcal{A}}(e; \Omega, \omega; L; \Sigma) = \mathrm{k}_{\mathcal{A}}(e; \Omega', \omega'; L; \Sigma)$.

6 RELATED WORK

Flow Locks and Paralocks. Flow Locks and Paralocks form the basis of the declassification mechanism we present here. They also inspired our security definition. The first work on Flow Locks [Broberg and Sands 2006] introduces an unscoped version of the open and close constructs. There are two very different security definitions for Flow Locks: a bisimulation-based definition [Broberg and Sands 2006] and a knowledge-based definition [Broberg and Sands 2009] (which we build on). Our security definition generalizes Flow Locks' security definition to make it compositional, and to lift it to the higher-order setting with some technical changes, e.g., our open and close constructs are scoped.

Paralocks [Broberg and Sands 2010] is an extension of Flow Locks which both generalizes Flow-Locks policies in several ways and adds a when construct that allows programs to test whether a lock is open. Defining security for a language with when requires making locks observable. In order to focus on the key features of higher-order where declassification, we did not consider Paralocks' additions. However, the language of the accompanying technical appendix covers the when construct.

Higher-Order where declassification. We are only aware of two other works on where declassification for higher-order languages: the original Flow-Locks paper [Broberg and Sands 2006] and Matos and Boudol's work on the non-disclosure policy [Matos and Boudol 2005]. Both papers give a bisimulation-based definition in the store-based setting which only guarantees the lack of information leaks during the computation of an expression to a value, without giving any guarantees about later uses of the resulting value. This makes the security definitions noncompositional.

Broberg and Sands [2006] do not give security guarantees to their surface language. Instead, they translate all programs to a restricted language where all programs are in A-normal form, and then give security guarantees for *that* language. They state that their original language "[is] not well suited when defining the semantic security property," suggesting that their security property does not generalize.

Matos and Boudol [2005] use scoped flow constructs, similar to λ-WHR's open and close. This seems to be one of the key reasons that non-disclosure is usually defined as a where property, while Flow-Lock and Paralock security are usually defined as *when* properties. However, they treat functions imprecisely in at least two ways. First, they require that the output of a function be at least as sensitive as all of its inputs, even those on which the output does not depend, creating label creep. Second, they define function indistinguishability syntactically, which leads to noncompositionality in their definition.

Logical relations and information flow. We build directly on the language, associated type system, and logical relation for FG [Rajani and Garg 2018]. FG does not allow any declassification. Our work can be viewed as an extension of FG with where declassification, with policies specified in the style of Flow Locks.

Gregersen et al. [2021] use logical relations to provide a noninterference guarantee to a language with higher-order state and polymorphism. Frumin et al. go in a different direction with SeLoC [Frumin et al. 2021] by including concurrency. Both papers use Iris [Jung et al. 2018, 2015], a Coq [The Coq Development Team 2022] framework for reasoning about software using separation logic, for their proofs and definitions.

While we believe that we are the first to use logical relations to reason about *where* declassification, logical relations have been used to reason about *what* declassification. Cruz et al. [2017] propose type-based relaxed noninterference, an object-oriented version of relaxed noninterference [Li and Zdancewic 2005]. They use a logical relation to make type-based relaxed noninterference compositional, thus allowing relaxed noninterference to be lifted to object-oriented programming, which is inherently higher-order. Likewise, Ngo et al. [2020] use a logical relation to adapt relaxed noninterference to the simply typed lambda calculus.

Relevant declassification. While we are the first to use relevant declassification as the basis of a model of types—and the first to name it—the insight that in the presence of (intensional) declassification we cannot always require indistinguishability is not new. Our concrete definition of relevant declassification is inspired by the definitions of Flow-Lock and Paralock security. Specifically, the proofs in the appendixes and full developments of the respective papers [Broberg and Sands 2009, 2010] contain ideas that influenced various aspects of relevant declassification.

The closest analogues to our relevant declassification condition can be found in bisimulation-based definitions for declassification, which only require that the second execution match the first if some precondition is satisfied. Often, this precondition requires that the memories the two programs are executed in are indistinguishable given the currently active declassification policies (in our case, the open locks). This is, for example, the case in the nondisclosure policy [Matos and Boudol 2005] and the original Flow-Locks security definition [Broberg and Sands 2006]. Note that, unlike our definition, this condition does *not* require any difference in the observed behavior.

Our formalization of relevant declassification is also similar to conditions in the definition of strong D-bisimulations by Mantel and Sands [2004]. Mantel and Sands instrumented their operational semantics to distinguish steps in which declassification is allowed from other execution steps. They then define a bisimulation which, similar to our logical relation, requires that declassifications happen in lockstep. Then, if a declassification step satisfies a condition very similar to our relevance condition, they do not require the resulting memories to be related. Concretely, their condition checks that (a) the downgrade is allowed by the declassification policy, (b) that the downgrade is visible at the currently considered security level (i.e, to the attacker), and (c) that the starting memories differ when viewed from the security level the declassification starts at. Notably, in case of relevant declassification they still require the resulting programs to be indistinguishable from each other. We cannot do this without breaking compositionality.

Unlike our security condition, strong D-bisimulations are a store-based approach to security that cannot handle dynamic allocation of memory locations. They were applied to a first-order imperative language only.

The security definitions for escape-hatch style what-declassification mechanisms à la delimited release [Sabelfeld and Myers 2003] contain a clause that might look superficially similar to our relevant declassification mechanism. The clause ensures that indistinguishability is only enforced if the declassified expressions are indistinguishable in the two executions under consideration. This condition is, however, fundamentally different from our notion of relevant declassification in that it can be checked *once and for all* right at the beginning of the program. This works because what declassification is an extensional property. We have to make the relevance check at all writes during the program's executions because, in our setting, the interpretation of the policy and, hence, relevance, depends on the part of the program that is executing.

Small-step logical relations. In order to be able to give security guarantees for partial traces and diverging programs, we chose to give security guarantees for individual steps of computation, requiring that the resulting programs be related in the expression relation again. While this *small-step* approach was inspired by bisimulations, it is also standard in logical-relations models. Notably,

most logical relations built in Iris [Jung et al. 2018, 2015] are defined using a weakest-precondition predicate defined over small-step reductions. Most closely related to our definition are the logical relations of Simuliris [Gäher et al. 2022]—which sets up a one-directional simulation between executions—and the previously mentioned SeLoC [Frumin et al. 2021]—which sets up a strong lockstep bisimulation for noninterference, similar to our C_{PAR} clause (but without declassification).

Knowledge-based security definitions. Askarov and Sabelfeld [2007a] first proposed knowledge-based security guarantees for *gradual release*, which allows declassification via specific commands. First, they give a knowledge-based definition of noninterference: attacker knowledge does not increase during program execution. They then contrast it with a guarantee for gradual release itself: attacker knowledge may increase only at specific declassification commands.

Since then, knowledge-based definitions have been used by many researchers. Importantly, Broberg and Sands [2010] replaced their original bisimulation-based security definition for Flow Locks [Broberg and Sands 2006] with a *flow-sensitive* knowledge-based definition. While our security definition is not flow sensitive, we believe that higher-order flow-sensitive security definitions should be possible through more advanced logical relations technology [Ahmed et al. 2009; Hur et al. 2012].

All knowledge-based security definitions we are aware of apply to lower-order languages only. As we noted in Section 5, knowledge-based security definitions assume that indistinguishability of observations is an equivalence relation. It is unclear whether one can satisfy this assumption to get a termination-insensitive security definition in the presence of higher-order state.

7 CONCLUSION

We have defined λ-WHR, a higher-order language based on FG [Rajani and Garg 2018] with where-declassification policies based on Flow Locks [Broberg and Sands 2006, 2009]. We were able to give a *compositional* definition of security for λ-WHR, the first such definition for higher-order where declassification. To achieve this, we used a logical relation, which deems two functions indistinguishable if indistinguishable inputs lead to indistinguishable results. This definition is compositional on the language syntax. Relevant declassification enabled us to reason about the intensional aspects of where declassification in our logical relation. Formalizing and incorporating relevant declassification into the logical relation forms the core technical contribution of our work.

Although we are not the first to notice that logical relations are particularly well-suited to defining compositional security properties—they have been used both for noninterference [Frumin et al. 2021; Gregersen et al. 2021; Rajani and Garg 2018] and what declassification [Cruz et al. 2017; Ngo et al. 2020], we have demonstrated a new use of logical relations by showing that they can be used to reason about intensional security properties.

Our security definition is based on the lower-order definition of [Broberg and Sands 2009]. We lift their definition to the higher-order setting by embedding the key conditions in the expression relation of our logical relation. We believe that this approach can be applied to other declassification mechanisms and styles as well. For example, we believe that lifting bisimulation-based security definitions for other declassification mechanisms like those of Matos and Boudol [2005] and Mantel and Sands [2004] to the higher-order setting should pose few further technical challenges. As noted in Section 6, many bisimulation-based security definitions rely on conditions similar our condition defining relevant declassification. Our expression relation essentially *is* a bisimulation that puts additional requirements on values via the value logical relation. We therefore expect that we could similarly extend other bisimulation-based definitions to the higher-order setting by extending them with a value relation and replacing their notion of indistinguishability with relatedness in the logical relation, thus obtaining compositionality.

Furthermore, a logical-relations definition of security for *when* declassification based on our relation should be straightforward.

Even though our definition of security is not flow-sensitive, this is not inherent in our approach. By tracking the *contents* of the state in the world, as is routine in Kripke logical relations [Ahmed et al. 2009], we would obtain a flow-sensitive security definition. Frumin et al. [2021] and Gregersen et al. [2021] already incorporate this in their logical relations for noninterference.

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. Princeton University. https://www.ccs.neu.edu/home/amal/ahmedthesis.pdf

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 340–353. https://doi.org/10.1145/1480881.1480925

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Aslan Askarov and Andrei Sabelfeld. 2007a. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*. IEEE Computer Society, 207–221. https://doi.org/10.1109/SP.2007.22

Aslan Askarov and Andrei Sabelfeld. 2007b. Localized delimited release: combining the what and where dimensions of information release. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007*, Michael W. Hicks (Ed.). ACM, 53–60. https://doi.org/10.1145/1255329.1255339

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2008. Expressive Declassification Policies and Modular Static Enforcement. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 339–353. https://doi.org/10.1109/SP.2008.20

Niklas Broberg and David Sands. 2006. Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 180–196. https://doi.org/10.1007/11693024_13

Niklas Broberg and David Sands. 2009. Flow-sensitive semantics for dynamic information flow policies. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, Stephen Chong and David A. Naumann (Eds.). ACM, 101–112. https://doi.org/10.1145/1554339.1554352

Niklas Broberg and David Sands. 2010. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 431–444. https://doi.org/10.1145/1706299.1706349

Raimil Cruz, Tamara Rezk, Bernard P. Serpette, and Éric Tanter. 2017. Type Abstraction for Relaxed Noninterference. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.7

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. Compositional Non-Interference for Fine-Grained Concurrent Programs. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1416–1433. https://doi.org/10.1109/SP40001.2021.00003

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434291

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 59–72. https://doi.org/10.1145/2103656.2103666

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Elisavet Kozyri and Fred B. Schneider. 2020. RIF: Reactive information flow labels. *J. Comput. Secur.* 28, 2 (2020), 191–228. https://doi.org/10.3233/JCS-191316

Peng Li and Steve Zdancewic. 2005. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 158–170. https://doi.org/10.1145/1040305.1040319

Heiko Mantel and David Sands. 2004. Controlled Declassification Based on Intransitive Noninterference. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings (Lecture Notes in Computer Science, Vol. 3302)*, Wei-Ngan Chin (Ed.). Springer, 129–145. https://doi.org/10.1007/978-3-540-30477-7_9

Ana Almeida Matos and Gérard Boudol. 2005. On Declassification and the Non-Disclosure Policy. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society, 226–240. https://doi.org/10.1109/CSFW.2005.21

Jan Menz, Andrew Hirsch, Peixuan Li, and Deepak Garg. 2023. *Compositional Security Definitions for Higher-Order Where Declassification - Technical Appendix*. Technical Report. https://gitlab.mpi-sws.org/Quarkbeast/lambda-where-fullproofs/-/raw/main/Technical_Appendix.pdf

Minh Ngo, David A. Naumann, and Tamara Rezk. 2020. Type-Based Declassification for Free. In *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12531)*, Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony (Eds.). Springer, 181–197. https://doi.org/10.1007/978-3-030-63406-3_11

Vineet Rajani and Deepak Garg. 2018. Types for Information Flow Control: Labeling Granularity and Semantic Models. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 233–246. https://doi.org/10.1109/CSF.2018.00024

Andrei Sabelfeld and Andrew C. Myers. 2003. A Model for Delimited Information Release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers (Lecture Notes in Computer Science, Vol. 3233)*, Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki (Eds.). Springer, 174–191. https://doi.org/10.1007/978-3-540-37621-7_9

The Coq Development Team. 2022. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.5846982

Received 2022-10-28; accepted 2023-02-25