# Rely/Guarantee Reasoning for Noninterference in Non-Blocking Algorithms

Nicholas Coughlin and Graeme Smith
*School of Information Technology and Electrical Engineering*
*The University of Queensland, Australia*
n.coughlin@uq.edu.au, smith@itee.uq.edu.au

***Abstract*—Noninterference characterizes a security property in which an attacker cannot determine the inputs to a system based on outputs of a lower classification. Value-dependent noninterference enables the analysis of systems in which these classifications may depend on the system's state and evolve throughout execution. Existing approaches to enforcing such a property for concurrent systems are constrained in their capability to express how the concurrent components modify shared variables and, therefore, the value-dependent classifications. Such approaches typically make use of externally verified annotations or coarse locking primitives to express limited constraints on variables, such as read and write permissions. Consequently, these techniques are insufficient for the analysis of programs that feature complex concurrent behaviours or require fine-grained synchronisation, as seen in non-blocking algorithms.**

**This paper presents a compositional logic for enforcing value-dependent noninterference properties for complex concurrent algorithms, including non-blocking algorithms. It uses rely/guarantee reasoning to establish how classifications may be modified by concurrent components. Additionally, the logic allows for the specification of security policies at a component level and ensures their valid composition. These results have been formalised in Isabelle/HOL.**

## I. Introduction

Approaches to demonstrating secure information flow have been researched extensively, due to their crucial nature in building secure systems capable of handling data of various sensitivities. *Noninterference* [1] formally characterizes an information flow security property in which an information source of a particular classification cannot influence sinks at lower classifications. As a result, a system exhibiting noninterference would prevent an attacker from determining secret values given a series of public outputs.

Type systems [2], [3] have been thoroughly explored as a method of demonstrating noninterference properties. For concurrent programs, these have utilised *rely/guarantee reasoning* [4] in which assumptions (or *rely* conditions) about a component's environment can be used in reasoning about that component provided all other components *guarantee* that the assumption holds. This provides a scalable analysis technique in which (sequential) components are considered in isolation and their results composed to demonstrate a noninterference property for the entire system. Existing approaches using rely/guarantee reasoning for non-interference, however, place significant restrictions on supported programs.

For example, Mantel et al. [5] propose a type system for concurrent programs which have static classifications for variables. This approach employs rely/guarantee reasoning by allowing a program to be annotated with read/write permissions for shared variables, either via a user or external process. These annotations, being limited to read/write permissions, are not sufficient for capturing the kinds of assumptions needed in many concurrent systems. Furthermore, while they simplify the type checking process, they do so at the cost of an additional proof obligation establishing the validity of the annotations and their compatibility between components.

Murray et al. [6] extend this approach to facilitate *value-dependent* noninterference in which a variable's classification may depend on the value of other variables, referred to as *control* variables. Such value-dependent properties are typically highly dependent on establishing functional correctness to accurately reason about control variables and their resulting classifications throughout execution [7]. While this approach extends Mantel et al.'s to a wider range of programs, it still suffers the same limitations: assumptions and guarantees are limited to read/write permissions, and their validity and compatibility between components needs to be established. These concerns have been partially addressed in later work by Murray et al. [8], [9] by allowing for more general rely/guarantee conditions coupled with locking primitives. The latter ensure assumptions that are made when a component holds a lock are guaranteed by all other components when they do not hold the lock. The fact that only one component can hold a given lock at a time greatly reduces the proof burden for the compatibility of annotations between components.

For performance, however, many applications utilise *non-blocking algorithms* which avoid the use of locks on shared variables and data structures [10]. Such algorithms are used extensively in operating systems and hence underlie all other applications. For this reason, it is crucial that we have means of detecting potential security leaks caused by them.

In this paper, we present a logic for the compositional verification of value-dependent noninterference, for concurrent programs with shared memory and fine-grained synchronisation, such as non-blocking algorithms. This is a notable improvement on other work in the field, as it supports general rely/guarantee conditions when establishing functional properties of the program; as well as component level security policies, facilitating non-trivial information flow between a program's components; and is entirely self-contained, avoiding the introduction of additional proof obligations exter-

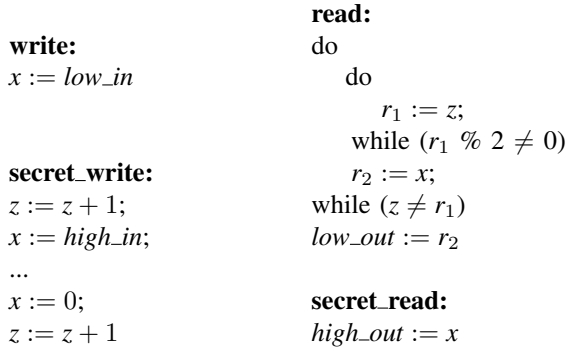


Fig. 1. Readers/writer example

nal to the logic. These results have been formalised in Isabelle/HOL [11]. Additionally, we introduce restrictions on these rely/guarantee conditions, assisting in automation of the logic. In Section II, we demonstrate, through a simple example, where existing logics for concurrent systems fail. In Section III, we provide an overview of our work, establishing our programming language and principal definitions. In Section IV, we detail the rely/guarantee theory with which we attain a compositional logic. In Section V, we establish our logic for demonstrating value-dependent noninterference on sequential components, and apply it to a simple application in Section VI. In Section VII, we describe modifications to simplify application of the logic. Finally, in Section VIII we detail related work and in Section IX we conclude the paper.

## II. Application

Consider the code in Figure 1 where variables $z$ and $x$ are shared between all concurrent components, and all other variables are local. A single writer component places information in a buffer $x$ using either **write** or, when the information is classified, **secret_write**. The latter operation increments a variable $z$ before placing information in $x$ and then increments $z$ again after the buffer has been read (the detection of which is elided in Figure 1) and the buffer cleared (by setting it to 0). Since $z$ is initially 0, this ensures that $z$ is odd whenever there is classified information in the buffer $x$.

The operation **secret_read** allows the buffer to be read at any time and can only be accessed by privileged components; all other components must read the buffer using **read**. We assume an attacker is only able to inspect the result of **read**, via *low_out*. This operation ensures that non-privileged components do not access classified information as follows. The value of $z$ is repeatedly read into a local variable $r_1$ until an even value is read. The information in the buffer is then read into a local variable $r_2$. Finally, a check is made that the value of $z$ has not changed since it was last read into $r_1$. If this is the case, the information in $r_2$ is not classified and hence can be output to the calling component. If, however, $z$ has changed, it is possible that the information in $r_2$ is classified and the operation restarts from the beginning.

The check that $z$ has not changed is made by comparing $z$ with $r_1$. This check is sufficient provided that $z$ always increases (something we know is true from **secret_write** which is the only operation to change $z$). To reason in isolation about a component calling **read** requires two things: a value-dependent security policy stating that $x$ does not contain classified information when $z$ is even, and an assumption that $z$ only increases. The former is not supported by the approach of Mantel et al. [5] which only allows static classifications of variables. It is supported by the approaches of Murray et al. [6], [8], [9]. However, none of the existing approaches support assumptions about the change of a variable: Mantel et al. and the early work of Murray et al. [6] are limited to assumptions on read/write permissions to a variable, and the later work of Murray et al. [8], [9] only supports assumptions on a single state (an assumption about the change of a variable needs to involve at least two states).

More importantly, the later work of Murray et al. requires that assumptions be associated with acquiring locks. The **secret_write** and **read** operations are based on a Linux read-write mechanism called seqlock [12], and is a typical example of a non-blocking algorithm. Such algorithms allow components to concurrently run the same code with no, or minimal, use of locking. The idea is that, for efficiency, the **secret_write** operation can operate without locking (and hence without delay), and multiple non-privileged components can call the **read** operation simultaneously: no component is ever blocked while waiting to obtain a lock. It is the purpose of this paper to provide support for reasoning about noninterference in such non-blocking algorithms.

## III. Overview

This section provides a high level description of the logic proposed in this paper.

### A. Language

We make use of a simple while-language for the purposes of our logic. Variables either belong to the set *Global* and are shared between all components, or the *Local* set. All components have the same set of *Local* variable names, however, they refer to distinct memory regions. Throughout examples, as seen in Figure 1, we refer to *Local* variables by $r_1, r_2, r_3$. Other variables are assumed to be *Global*. We refer to inputs and outputs of the system by ending their variable names in *_in* and *_out* respectively. We let $b$ refer to a boolean expression and let $e$ represent a value expression, with the constraint they are deterministic and their execution time is data-independent. Our language is defined as follows[1].

$$c \equiv x := e \mid \mathsf{skip} \mid c; c \mid \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ b\ \mathsf{do}\ c$$

[1]The $\mathsf{do}\ c\ \mathsf{while}\ (b)$ construct used in the code of Figure 1 is simply a shorthand for $c\ ;\ \mathsf{while}\ (b)\ \mathsf{do}\ c$.

$r_1 := z$
if $r_1 = 0$ then
    $r_2 := x$
else
    $r_2 := 0$
*low_out* $:= r_2$

Fig. 2. Conditional example

We restrict classifications to operate over a two-point security lattice, containing *High* and *Low*, structured such that *Low* $\sqsubseteq$ *High* and *High* $\not\sqsubseteq$ *Low*. Figure 2 demonstrates an example where a variable $x$ is read conditionally, based on the value of $z$, and the result written to an output *low_out*. In this example, we intend $x$ to have a value-dependent classification, such that it is *Low* when $z = 0$. Moreover, we consider the variable *low_out* to always be classified as *Low*. Hence, this example does not leak information, under the assumption that no other component changes the value of $z$, and subsequently the classification of $x$, after $z$ is read and before the read of $x$.

### B. Sequential Logic

Our logic operates over individual sequential components, with a context consisting of the tuple $P$ and $\Gamma$. Judgements are of the form $\vdash \{P, \Gamma\} \;\; c \;\; \{P', \Gamma'\}$ for a component $c$.

$P$ encodes a predicate over the current state, in a similar approach to a Hoare logic. In Figure 2, it would encode the test outcome in each respective branch. As a result, it would hold $z = 0$ in the then-branch, enabling reasoning about the classification of $x$ at the point it is read.

Value-dependent classifications are encoded as predicates, as in Murray et al. [6], [8]. $\Gamma$ encodes the local information flow context, retaining the classification of values held in variables based on local writes. To achieve this, it is structured as a mapping from variables to classification predicates. In Figure 2, it is necessary to establish that the data held in $r_2$ is of a *Low* classification to enable the final write to variable *low_out*. This is achieved via $\Gamma$, as both if-statement branches are able to demonstrate $r_2$ holds a *Low* value and update $\Gamma(r_2)$ to map to *True*, encoding *Low*.

### C. Rely/Guarantee Reasoning

The system is assumed to consist of a static set of sequential components, each of which require a valid logic judgement. Rely/guarantee reasoning is then employed to compose these judgements and establish an information flow property over the entire system. To achieve this, two forms of rely/guarantee annotations are introduced for each component. The first enforces functional correctness properties, which we detail here, whilst the second describes information flow between components, which we provide an overview of in Section III-D.

For functional correctness, each component is coupled with a rely, $\mathcal{R}$, and guarantee, $\mathcal{G}$. These definitions follow the standard style of rely/guarantee reasoning, where $\mathcal{R}$ and $\mathcal{G}$ are relations on memories. A component assumes all others, collectively referred to as the environment, conform to the relation $\mathcal{R}$ and guarantees its own actions conform to $\mathcal{G}$.

Applying this to the sequential logic, $P$ and $\Gamma$ must be *stable* with respect to the rely condition $\mathcal{R}$, i.e., they cannot be invalidated by any interleaved actions from the environment. To illustrate, the predicate $z = 0$ would be stable only if $\mathcal{R}$ prevented the environment from modifying $z$. Additionally, the logic must ensure a component's actions conform to its own guarantee relation.

For rely/guarantee reasoning a proof of *compatibility* between the relations is necessary, such that, for any two components in the system, the rely relation of one contains the guarantee relation of the other. As the $\mathcal{R}$ and $\mathcal{G}$ relations apply to the entire execution of a component, this proof of compatibility is relatively straightforward.

Applying this to the running example, it is necessary to show the environment will not redefine $z$ between its read and the later read of $x$. Under one definition, the environment could be constrained by $z = z'$, a relational predicate encoding of $\mathcal{R}$, stating that $z$ is not modified. Consequently, it will preserve the value of $z$ under all conditions, resulting in the predicate $z = r_1$ being considered stable.

Alternatively, the constraints on the environment could be weakened to $z = 0 \Rightarrow z = z'$, stating that the environment will not modify $z$ once it is equal to 0. Hence, the predicate must be weakened to achieve stability whilst preserving information, attaining $r_1 = 0 \Rightarrow z = r_1$. It is then possible to demonstrate $r_1 = z$ and, subsequently, $x$'s classification as *Low* for the then-branch.

### D. Security Policies

In this work, information flow properties are encoded as security policies, mapping *Global* variables to classification predicates. Our approach supports the enforcement of a global security policy, $\mathcal{L}$, which is intended to hold throughout all stages of the system's execution. In the running example, this would encode the value-dependent classifications as $\mathcal{L}(\textit{low\_out}) \equiv \textit{True}$ and $\mathcal{L}(x) \equiv z = 0$.

This security policy serves two purposes, as it describes the classifications of inputs and outputs of the system, in addition to providing a means for components to coordinate their information flow. However, there are cases where a single global policy is insufficient.

For example, consider a privileged component, capable of storing *High* information to a variable, where all others can only store *Low*. It is not possible to encode this situation given a single policy, as a more restrictive policy must be enforced on the privileged component's environment. This can be seen in Figure 1, where only **secret_write** may write a *High* value.

To support such a scenario, the approach allows for the specification of two security policies for each component: $\mathcal{L}_G$, which constrains the component's own writes; and $\mathcal{L}_R$, which restricts the writes due to the environment. These policies are constrained to correspond to the desired global policy $\mathcal{L}$ along with proofs of compatibility, similar to the functional correctness rely/guarantee relations.

## IV. Compositional Noninterference

We first detail the semantics of our concurrent system, along with our definition of secure information flow at a global level. Following this, we describe our compositional technique, which extends prior work [6] to enable a wider variety of security policies in a self-contained logic. The definitions and theorems presented here can be considered a specialisation of the rely/guarantee parallel rule [4], describing how to appropriately compose the analysis of individual threads.

### A. Preliminaries

Let $\langle c, mem \rangle$ denote the configuration of a component, where $c$ is the executing code of the component and *mem* encodes memory as a mapping from variables to values. The memory in this configuration has a domain that includes *Local* and *Global* variables. We use $\rightarrow$ to denote a transition over local configurations using small step semantics, which are assumed to be deterministic.

Let $(comp, mem_G)$ denote the state of the system, where $mem_G$ encodes the memory for only *Global* variables and *comp* consists of a list of sequential components. These components are represented as a tuple, $(c, mem_L)$, such that $c$ is the program code and $mem_L$ encodes the memory for *Local* variables. We define transitions for global configurations as $\rightarrow_i$, where $i$ refers to the index of the local component performing an action.

$$\frac{\begin{array}{c} i < |comp| \\ comp[i] = (c, mem_L) \\ \langle c, mem_G + mem_L \rangle \rightarrow \langle c', mem'_G + mem'_L \rangle \end{array}}{(comp, mem_G) \rightarrow_i (comp[i \mapsto (c', mem'_L)], mem'_G)} \quad (1)$$

We use $m_1 + m_2$ to represent the merging of two mappings, under the assumption that their domains do not overlap. We also introduce $l[i]$ for accessing index $i$ of list $l$, $\#$ as list concatenation, and $|l|$ for the length of list $l$. Additionally, we define a list update as $l[i \mapsto e]$ which updates the $i$th position of $l$ to value $e$. The structure of our global transitions provides each component with a local memory and clearly enforces their separation.

In this work, we assume a fixed schedule, preventing execution behaviour from influencing later scheduling decisions. Moreover, we assume a fixed set of components. As a result, we can express the scheduler as a list of component indices and define its execution in terms of (1).

$$\frac{}{(comp, mem_G) \longrightarrow_{[]} (comp, mem_G)} \quad (2.1)$$

$$\frac{\begin{array}{c} (comp, mem_G) \rightarrow_i (comp', mem'_G) \\ (comp', mem'_G) \longrightarrow_t (comp'', mem''_G) \end{array}}{(comp, mem_G) \longrightarrow_{i\#t} (comp'', mem''_G)} \quad (2.2)$$

### B. Security Policies

To facilitate value-dependent classifications with compositionality, our approach employs several static security policies. These policies map variables to predicates stating the conditions under which the variable is considered to hold *Low* information. For example, a security policy mapping $x$ to $c = 0$ would require $x$ to hold *Low* information whenever $c$ is 0.

Similar to prior work [6], we introduce a global security policy $\mathcal{L}$, which the system conforms to at all stages throughout execution. However, to capture a wider variety of secure behaviours, we allow $\mathcal{L}$ to be broken down into further policies at the component level. Each component may introduce policies $\mathcal{L}_R$ and $\mathcal{L}_G$, named to parallel the rely/guarantee relations, such that $\mathcal{L}_G$ constrains information flow due the component's own writes, whilst $\mathcal{L}_R$ constraints those of other components. Moreover, to ensure components still conform to the global security policy $\mathcal{L}$, all component level policies must match the global, such that $\forall\, x \cdot \mathcal{L}(x) = \mathcal{L}_R(x) \wedge \mathcal{L}_G(x)$.

Noninterference is commonly established via *low equivalence* between two program configurations under a bisimulation. This property enforces equivalence between the two configurations for all variables classified as *Low*. Consequently, it is not possible to distinguish the two program configurations via inspection of only *Low* variables.

We introduce the following definition of low equivalence between two memories, given a security policy $l$ for a set of variables $V$.

$$\begin{array}{l} mem_1 =^{l,V} mem_2 \equiv \\ \quad \forall x \in V.\ \mathsf{eval}\ mem_1\ (l\ x) \vee \mathsf{eval}\ mem_2\ (l\ x) \Rightarrow \\ \quad\quad mem_1\ x = mem_2\ x \end{array} \quad (3)$$

where $\mathsf{eval}\ mem\ P$ is the evaluation of predicate $P$ on *mem*.

This representation of low equivalence allows for the two memories to disagree on the classification of a variable, however, it will always take the lowest classification of the two, preventing a potential leak from the perspective of both memories. It is a notable extension, compared to other work, as the variables referenced in the security policies are unconstrained. We use the shorthand $mem_1 =^l mem_2$ to represent low equivalence on all variables.

This definition may introduce a side-channel attack, depending on the model of the attacker and the system. For example, consider a system where an attacker is permitted to read variables with value-dependent classifications only when these classifications evaluate to *Low*, but is denied access otherwise. In the event that *High* information influences these permissions, an attacker may be capable of distinguishing two executions based on differences in permissions. We would consider this a leakage via this hypothetical permission system, rather than an issue with our definition of low equivalence. Consequently, we constrain the attacker to only accessing outputs of the system that are statically classified as *Low*. Variables with value-dependent classification are instead used for describing information flow between the system's components.

### C. Relations

To achieve a simple compositional approach, based on rely/guarantee reasoning, we introduce rely and guarantee relations that encapsulate the security policy definitions, $\mathcal{L}_R$ and $\mathcal{L}_G$, in addition to the relations $\mathcal{R}$ and $\mathcal{G}$ constraining program behaviour.

As these encapsulating relations, which we refer to as $\mathcal{R}_p$ and $\mathcal{G}_p$, include both bisimulation and behavioural properties, it is necessary to encode them as relations over pairs of bisimilar memories. To illustrate, given $((mem_1, mem_2), (mem'_1, mem'_2)) \in \mathcal{G}_p$, the component would transition $mem_1$ to $mem'_1$ and $mem_2$ to $mem'_2$ whilst potentially establishing bisimulation properties between $mem'_1$ and $mem'_2$ given suitable properties between $mem_1$ and $mem_2$.

$$\begin{aligned}\mathcal{R}_p \equiv \{((mem_1, mem_2), (mem'_1, mem'_2))\cdot \\ (mem_1, mem'_1) \in \mathcal{R} \wedge \\ (mem_2, mem'_2) \in \mathcal{R} \wedge \\ mem'_1 =^{\mathcal{L}} mem'_2 \wedge \\ mem'_1 =^{\mathcal{L}_R, V_d} mem'_2\}\end{aligned} \tag{4}$$

$$\begin{aligned}\mathcal{G}_p \equiv \{((mem_1, mem_2), (mem'_1, mem'_2))\cdot \\ (mem_1, mem'_1) \in \mathcal{G} \wedge \\ (mem_2, mem'_2) \in \mathcal{G} \wedge \\ mem'_1 =^{\mathcal{L}} mem'_2 \wedge \\ mem'_1 =^{\mathcal{L}_G, V_d} mem'_2\}\end{aligned} \tag{5}$$

where $V_d = \mathsf{diff}\ mem_1\ mem'_1\ \cup\ \mathsf{diff}\ mem_2\ mem'_2$.

We define $\mathcal{R}_p$ and $\mathcal{G}_p$ for each component, based on the definitions above. These definitions first enforce the component's $\mathcal{R}$ and $\mathcal{G}$ relations across all transitions, ensuring rely/guarantee reasoning is fully supported by the logic. This is crucial to establishing a self-contained analysis capable of capturing a variety of synchronisation behaviours, in contrast to other approaches where annotations or locks are employed.

Two low equivalence properties are required to constrain information flow. The first, $mem'_1 =^{\mathcal{L}} mem_2$, ensures both the component and the environment preserve the global security policy $\mathcal{L}$ across all variables. Consequently, a local analysis can always assume this security property holds at a minimum. The remaining property enforces the more specific component level security policy, $\mathcal{L}_R$ or $\mathcal{L}_G$, restricted to only those variables that have been modified, $V_d$.

This enables information flow reasoning at a component level to distinguish between the environment and itself, as motivated in Section III-D. As a result, a component can guarantee a more restrictive classification $\mathcal{L}_G(x)$ when writing to $x$, in comparison with $\mathcal{L}(x)$, or assume a more restrictive policy on the environment's writes to $x$, $\mathcal{L}_R(x)$.

Note it would be possible to use the individual properties ($\mathcal{L}_R$, $\mathcal{L}_G$, $\mathcal{R}$, $\mathcal{G}$) rather than the relations $\mathcal{R}_p$ and $\mathcal{G}_p$ throughout the following definitions. However, merging them simplifies definitions and corresponding proofs as it assists in the reuse of standard rely/guarantee properties and theorems.

### D. Compositional Bisimulation

We can now define the bisimulation at both a global and component level. We use the syntax $(c_1, mem_1)\ \mathcal{B}\ (c_2, mem_2)$ to represent a bisimulation $\mathcal{B}$ within which component $c_1$ and memory $mem_1$ are related to component $c_2$ and memory $mem_2$. For a component with guarantee $\mathcal{G}_p$, we introduce the definition of a bisimulation extended to enforce its guarantee.

$$\begin{aligned}&\mathsf{bisim}\ \mathcal{B}\ \mathcal{G}_p \equiv \\ &\quad \mathsf{sym}\ \mathcal{B} \wedge \forall\, c_1\ mem_1\ c_2\ mem_2\ c'_1\ mem'_1. \\ &\qquad (c_1, mem_1)\ \mathcal{B}\ (c_2, mem_2) \Rightarrow \\ &\qquad\quad \langle c_1, mem_1\rangle \rightarrow \langle c'_1, mem'_1\rangle \Rightarrow \\ &\qquad\qquad (\exists\, c'_2\ mem'_2. \\ &\qquad\qquad\quad \langle c_2, mem_2\rangle \rightarrow \langle c'_2, mem'_2\rangle \wedge \\ &\qquad\qquad\quad ((mem_1, mem_2), (mem'_1, mem'_2)) \in \mathcal{G}_p \wedge \\ &\qquad\qquad\quad (c'_1, mem'_1)\ \mathcal{B}\ (c'_2, mem'_2))\end{aligned} \tag{6}$$

where $\mathsf{sym}\ \mathcal{B} \equiv \forall\, x\ y \cdot (x, y) \in \mathcal{B} \Rightarrow (y, x) \in \mathcal{B}$

Moreover, we introduce stability for the bisimulation, ensuring the relation is preserved across memory changes due to the environment, based on a component's rely $\mathcal{R}_p$.

$$\begin{aligned}&\mathsf{stable}\ \mathcal{B}\ \mathcal{R}_p \equiv \\ &\quad \forall\, c_1\ mem_1\ mem'_1\ c_2\ mem_2\ mem'_2. \\ &\qquad (c_1, mem_1)\ \mathcal{B}\ (c_2, mem_2) \Rightarrow \\ &\qquad\quad ((mem_1, mem_2), (mem'_1, mem'_2)) \in \mathcal{R}_p \Rightarrow \\ &\qquad\qquad (c_1, mem'_1)\ \mathcal{B}\ (c_2, mem'_2)\end{aligned} \tag{7}$$

We couple these definitions into a single property for a secure component, given its relations, $\mathcal{R}_p$ and $\mathcal{G}_p$, as well as initial conditions $P$, denoted as a set of memory pairs.

$$\begin{aligned}&\mathsf{secure}\ c\ P\ \mathcal{R}_p\ \mathcal{G}_p \equiv \\ &\quad \forall (mem_1, mem_2) \in P \cdot \exists\, \mathcal{B}. \\ &\qquad \mathsf{bisim}\ \mathcal{B}\ \mathcal{G}_p \wedge \mathsf{stable}\ \mathcal{B}\ \mathcal{R}_p \wedge \\ &\qquad (c, mem_1)\ \mathcal{B}\ (c, mem_2)\end{aligned} \tag{8}$$

Given such a bisimulation exists for all components, we can then compositionally establish a bisimulation over the entire system. It is necessary to express compatibility between components, such that given a component, all others will conform to its rely relation.

$$\begin{aligned}&\mathsf{compat}\ \mathcal{R}_{ps}\ \mathcal{G}_{ps} \equiv \\ &\quad \forall\, i < |\mathcal{R}_{ps}| \cdot\ \ \forall\, j < |\mathcal{G}_{ps}| \cdot\ \ i \neq j \Rightarrow \mathcal{G}_{ps}[j] \subseteq \mathcal{R}_{ps}[i]\end{aligned} \tag{9}$$

where we introduce the notation $\mathcal{R}_{ps}$ and $\mathcal{G}_{ps}$ to refer to lists of $\mathcal{R}_p$ and $\mathcal{G}_p$ respectively, such that $\mathcal{R}_{ps}[i]$ refers to the rely relation for the $i$th component. This definition of compatibility is standard for rely/guarantee reasoning.

Finally, we introduce the definition of a globally secure system, which preserves the global security policy $\mathcal{L}$ given the evaluation of any schedule $t$, including partial schedules.

$$\begin{aligned}&\mathsf{secure}_{\mathsf{G}}\ comp \equiv \\ &\quad \forall\, mem_1\ mem_2 \cdot\ \ mem_1 =^{\mathcal{L}} mem_2 \Rightarrow \\ &\qquad \forall\, t\ comp'\ mem'_1 \cdot \\ &\qquad\quad (comp, mem_1) \longrightarrow_t (comp', mem'_1) \Rightarrow \\ &\qquad\quad (\exists\, mem'_2 \cdot (comp, mem_2) \longrightarrow_t (comp', mem'_2) \wedge \\ &\qquad\qquad mem'_1 =^{\mathcal{L}} mem'_2)\end{aligned} \tag{10}$$

*Theorem 1:* Given a component, $c$, along with its individual rely/guarantee relations, $\mathcal{R}$ and $\mathcal{G}$, and its security policies, $\mathcal{L}_R$ and $\mathcal{L}_G$, it is possible to derive its rely/guarantee relations over paired memories, $\mathcal{R}_p$ and $\mathcal{G}_p$. Therefore, given a series of components, *comp*, and their derived rely/guarantee relations, $\mathcal{R}_{ps}$ and $\mathcal{G}_{ps}$, along with proofs that these components exhibit the desired bisimulation, secure, and compatibility properties, the global security property $\mathsf{secure}_G$ can be established.

$$\frac{\begin{array}{c}\mathsf{compat}\ \mathcal{R}_{ps}\ \mathcal{G}_{ps} \\ P \equiv \{(mem_1, mem_2) \cdot mem_1 =^{\mathcal{L}} mem_2\} \\ \forall\, i < |comp| \cdot\ \mathsf{secure}\ comp[i]\ P\ \mathcal{R}_{ps}[i]\ \mathcal{G}_{ps}[i]\end{array}}{\mathsf{secure}_{\mathsf{G}}\ comp}$$

Theorem 1 can be established using a similar approach to establishing the rely/guarantee parallel rule. We show the global property holds by defining a global bisimulation in terms of the local bisimulations provided in the premisses and inducting on the schedule $t$ in $\mathsf{secure}_{\mathsf{G}}$. The base case of an empty schedule is trivial, as the global security policy is known to hold initially.

For the inductive case, a component is selected and executed. It is necessary to establish the global security policy on the resulting memory state, as required by $\mathsf{secure}_{\mathsf{G}}$, and reestablish the secure properties for all components.

Given the bisim property for the selected component, we can show that the post configuration remains within its bisimulation, reestablishing its secure bisimulation property. Moreover, this transition is constrained by $\mathcal{G}_{ps}[i]$, and therefore enforces the global security policy $\mathcal{L}$, as required by $\mathsf{secure}_{\mathsf{G}}$. For all other components, we employ the compatibility premise to show the transition satisfies their rely relations. Therefore, it is possible to reestablish their secure bisimulation properties.

Consequently, a local analysis capable of establishing $\mathsf{secure}\ comp[i]\ P\ \mathcal{R}_{ps}[i]\ \mathcal{G}_{ps}[i]$ for all components in the system and corresponding proofs of compatibility can demonstrate a global security property.

## V. Sequential Logic

We introduce a logic based on this theory capable of demonstrating the desired local security property. As this logic operates on a single set of $\mathcal{R}$, $\mathcal{G}$, $\mathcal{L}$, $\mathcal{L}_R$ and $\mathcal{L}_G$ specifications we assume they are available throughout the following definitions.

### A. Logic Context

The logic encodes a forward pass over a component's code, maintaining a context across actions and environment steps throughout. This context consists of $P$, a predicate over the local and global memory, and $\Gamma$, a partial mapping from variables to the classification of the *values* they hold. Hence, $\Gamma$ tracks value classifications within a component, whilst the security policies $\mathcal{L}$, $\mathcal{L}_R$ and $\mathcal{L}_G$ define information flow at a global, compositional level. $P$ is required to enable context-aware reasoning over these value-dependent classifications.

We define the initial context as $P \equiv \mathit{True}$ and an empty mapping for $\Gamma$, encoding the weakest possible context. However, any wellformed context, which we define in Section V-C, would be acceptable.

To manipulate the context, we introduce update functions for assignments and the environment. These updates require the introduction of temporary variables to retain information across reassignment and express complex rely conditions. To illustrate this, consider the implications of an assignment $x := e$. Any references to $x$ in $P$ and $\Gamma$ must be removed, and the new definition of $x$ added to $P$. To achieve this and retain existing information, we replace prior references to $x$ with a fresh temporary variable $t$. This is a slight modification of strongest postcondition for an assignment to appropriately handle the divided context of $P$ and $\Gamma$.

$$\{P, \Gamma\} + [x := e] \equiv \{P[t/x] \wedge x = e[t/x], \Gamma[t/x]\} \tag{11}$$

where $\Gamma[t/x]$ replaces all references to $x$ in $\Gamma$'s range with references to $t$.

### B. Classifications

Similar to prior work in value-dependent logics [6], [8], [9], variable classifications are expressed as predicates, such that a variable is considered *Low* if its classification predicate evaluates to true. Throughout the logic's rules, these classification predicates may be derived from $\mathcal{L}_R$, $\mathcal{L}_G$ or $\mathcal{L}$, depending on the context. Additionally, we enforce the constraint that a variable may not refer to itself in its classification, formally $x \notin \mathsf{vars}\ l$ where $l \in \{\mathcal{L}(x), \mathcal{L}_G(x), \mathcal{L}_R(x)\}$ and $\mathsf{vars}\ e$ denotes the free variables referenced in an expression or predicate. This simplifies the logic's rules without being overly restrictive.

When modifying a variable $x$, a component should consider $\mathcal{L}_G(x)$, the security policy governing its writes. However, it is not immediately obvious which security policy to use when reading $x$, as security policies $\mathcal{L}_R$ and $\mathcal{L}_G$ apply to writes from the environment and component respectively, and the most recent write could be derived from either. In this worst case, it is necessary to consider the global policy $\mathcal{L}(x)$ as this encompasses both situations.

It is possible to determine more accurate classifications via $\Gamma$, as it maps variables to their value's classification. Consequently, $\Gamma$ may hold a more specific classification for the read of a variable, such as capturing situations where a variable is considered to be *High* in $\mathcal{L}$ but has been written *Low* locally. To simplify the use of $\Gamma$, we introduce a total mapping $\Gamma\langle x \rangle$, which defaults to the worst case of $\mathcal{L}$ if necessary.

$$\Gamma\langle x \rangle \equiv \begin{cases} \Gamma(x) & \text{if } x \in \mathsf{dom}\ \Gamma \\ \mathcal{L}(x) & \text{if } x \in \mathit{Global} \\ \mathit{False} & \text{otherwise} \end{cases} \tag{12}$$

We treat *Local* variables as if they cannot be read by another component or attacker. As a result, they are able to hold values of any classification, encoded by the predicate *False*. Moreover, we introduce notation for determining the classification

of an expression, which evaluates to a conjunction over the classifications of its reads.

$$\Gamma \vdash e : t \;\equiv\; t = \bigwedge_{y \in \mathsf{vars}\ e} \Gamma\langle y \rangle \tag{13}$$

Additionally, we introduce comparisons on classifications via predicate entailment. For example, if it can be shown that $\mathcal{L}(y)$ is true in all contexts where $\mathcal{L}(x)$ is true, formally $\mathcal{L}(x) \Rightarrow \mathcal{L}(y)$, the classification of $y$ is considered lower than $x$. Consequently, it is always safe to perform the assignment $x := y$, as any context where $x$ is *Low* would imply $y$ is *Low*. Furthermore, these comparisons can be made context aware through the consideration of the current program state predicate $P$ using $P \wedge \mathcal{L}(x) \Rightarrow \mathcal{L}(y)$. For brevity, we employ the abbreviation $t \leq_{:P} t' \equiv P \wedge t' \Rightarrow t$.

### C. Wellformedness

We introduce a series of wellformedness properties for the logic context enforcing stability from the perspective of the rely specification $\mathcal{R}$ and $\mathcal{L}_R$. These allow us to establish the stable property defined in Section IV-D, and hence that environment steps cannot invalidate the context or the local logic's judgements. First, we require stability on our state predicate $P$ with respect to $\mathcal{R}$, in the traditional sense from rely/guarantee reasoning.

$$\begin{aligned}&\mathsf{stable\_P}\ P \equiv \forall\, mem\ mem'.\\ &\quad \mathsf{eval}\ mem\ P \wedge (mem, mem') \in \mathcal{R} \Rightarrow \mathsf{eval}\ mem'\ P\end{aligned} \tag{14}$$

Second, it is necessary to consider how the environment may invalidate $\Gamma$. Consider a mapping $\Gamma(x) = t$ stating that the variable $x$ holds a value with classification $t$. If the environment modifies $x$, its new value is constrained by the security policy $\mathcal{L}_R(x)$, regardless of $t$. Hence, the new classification of $x$'s value must be considered as $t \wedge \mathcal{L}_R(x)$, stating the value is *Low* if its value $t$ was *Low* prior to any potential environment steps and if the environment's writes were also *Low* values, $\mathcal{L}_R(x)$. Therefore, if it is possible to demonstrate $\mathcal{L}_R(x)$ is true in the current state $P$, the classification of $x$'s value can be considered unmodified. Furthermore, the classification of $x$'s value is trivially unmodified if the environment is prevented from writing to $x$ due to constraints in $\mathcal{R}$. We encode these properties as the set of variables low_or_eq, which forms the domain of a stable $\Gamma$.

$$\begin{aligned}&\mathsf{low\_or\_eq}\ P \equiv \{x \cdot P \Rightarrow \mathcal{L}_R(x) \vee\\ &\quad \forall\, mem\ mem' \cdot\\ &\qquad \mathsf{eval}\ mem\ P \wedge (mem, mem') \in \mathcal{R} \Rightarrow\\ &\qquad\quad mem\ x = mem'\ x\}\end{aligned} \tag{15}$$

Furthermore, the predicates in $\Gamma$'s range may refer to variables modified by an environment step. Consequently, the environment may modify the interpretation of a value's classification. To account for this, we introduce a stability property for $\Gamma$'s predicates.

$$\begin{aligned}&\mathsf{stable\_}\Gamma\ P\ \Gamma \equiv \forall\, mem\ mem'.\\ &\quad \mathsf{eval}\ mem\ P \wedge (mem, mem') \in \mathcal{R} \Rightarrow\\ &\qquad \forall x \in \mathsf{dom}\ \Gamma \cdot \mathsf{eval}\ mem\ \Gamma(x) = \mathsf{eval}\ mem'\ \Gamma(x)\end{aligned} \tag{16}$$

As the predicates in $\Gamma$ track classifications, they may not hold sufficient information to demonstrate restrictions on the environment steps. Hence, it is necessary to restrict the initial memory *mem* by $P$. Moreover, it is necessary to require equivalent interpretations of predicates in $\Gamma$ across an environment step rather than implication as seen in the stability property for $P$. This is required as it is possible that $\mathsf{eval}\ mem\ \Gamma(x)$ is false to encode a *High* value. If implication were used, the interpretation of $\Gamma(x)$ on the modified memory $mem'$ would, in this case, be unconstrained. These wellformedness properties are encapsulated in the definition $\mathsf{context\_wf}\ P\ \Gamma$.

$$\begin{aligned}&\mathsf{context\_wf}\ P\ \Gamma \;\equiv\\ &\quad \mathsf{stable\_}\Gamma\ P\ \Gamma \wedge \mathsf{stable\_P}\ P \wedge \mathrm{dom}\ \Gamma \subseteq \mathsf{low\_or\_eq}\ P\end{aligned} \tag{17}$$

### D. Rely/Guarantee Relations

To simplify the definition and application of the logic's rules, we introduce common constraints on the $\mathcal{R}$ and $\mathcal{G}$ relations. The first of these restricts $\mathcal{R}$ to exhibit transitivity and reflexivity. Transitivity allows for the consideration of multiple environment steps at once, whilst reflexivity encodes the possibility of no environment steps. $\mathcal{G}$ is restricted to exhibit reflexivity, avoiding constraints which would limit the current component's actions to those that mutate the global state.

Definitions thus far have employed the memory relation encodings of the rely/guarantee conditions, whilst the context encoding has employed predicates. To bridge this divide for concrete steps in the logic, we make use of the relational predicate encodings $R$ and $G$ for $\mathcal{R}$ and $\mathcal{G}$ respectively. For example, a relation predicate $z \leq z'$ specifies an equal or increasing $z$, where the primed variant refers to the modified variable.

Given $R$, it is possible to derive an update function for the context, ensuring the resulting state conforms to context_wf and, therefore, remains valid across any number of environment steps. To achieve this, we first define a mapping $m$ from *Global* variables to fresh temporary variables. Additionally, we define a mapping $m'$ from primed *Global* variables to their unprimed counterparts. Applying these mappings to $P$ and $R$ respectively, establishes a predicate that will be stable in $R$ as defined by stable_P.

$$P + R \;\equiv\; P[\mapsto m] \wedge R[\mapsto (m + m')] \tag{18}$$

where $P[\mapsto m]$ denotes mapping $m$ applied to predicate $P$.

$P + R$ is stable as $P[\mapsto m]$ will contain no references to *Global* variables and $R[\mapsto\ (m + m')]$ is stable due to the transitive property of $\mathcal{R}$.

To illustrate, consider the example of $R \equiv z \leq z'$ and $P \equiv r = z \wedge x = 0$. The mapping $m$ may be defined as $m \equiv [z \rightarrow t_1, x \rightarrow t_2]$. Therefore, an application of this update would produce $P + R = (r = t_1 \wedge t_2 = 0) \wedge (t_1 \leq z)$. The constraint on $r$ and $z$ is weakened across the update, from $=$ to $\leq$ whilst information regarding $x$ has been lost, due to a lack of restrictions in $R$.

This approach can also be applied to attain a stable $\Gamma$. Predicates within $\Gamma$ are updated using the mapping $m$, preserving

relationships between $P$ and $\Gamma$. The resulting predicates in $\Gamma$ are obviously stable as they do not refer to *Global* variables. Additionally, it is necessary to consider the domain of $\Gamma$. We employ the stable predicate $P + R$ to determine the set of low_or_eq variables and restrict the domain accordingly.

$$\{P,\Gamma\} + R \equiv \{P + R, \lambda x \in \mathsf{low\_or\_eq}\ (P+R) \cdot \Gamma(x)[\mapsto m]\} \tag{19}$$

The resulting context satisfies all three wellformedness properties, and is therefore stable across environment steps. However, these definitions have obvious flaws from the perspective of automation. This can be primarily seen in the modification of $P$, which will be extended by $R$ at every possible interleaving of environment steps. Given a large $R$, this quickly becomes prohibitively expensive without intervention to simplify the state. We employ this inefficient representation to simplify our soundness proof and retain a general logic. We introduce alternative definitions of $\{P,\Gamma\} + R$ to reduce this cost in the Section VII.

When considering $G$, it is necessary to prove the component's actions conform to its guarantee. This can be done via predicate reasoning. Note that only a *Global* assignment has to be considered, as no other action will modify global memory.

$$\mathsf{guar}\ P\ (x := e) \equiv P \wedge x' = e \wedge (\forall y \cdot y \neq x \Rightarrow y = y') \Rightarrow G \tag{20}$$

### E. Rules

The SKIP and SEQ rules follow the standard structure seen in Hoare logic. For the CONSEQ rule, we introduce an ordering on contexts.

$$P,\Gamma \geq P',\Gamma' \equiv \mathsf{context\_wf}\ P\ \Gamma \Rightarrow \mathsf{context\_wf}\ P'\ \Gamma' \wedge \forall x.\ \Gamma\langle x\rangle \leq_P \Gamma'\langle x\rangle \wedge P \Rightarrow P' \tag{21}$$

The ordering enforces preservation of wellformedness across the states, therefore preserving wellformedness across the entire CONSEQ rule. Moreover, it enforces an ordering on $P$ via entailment, in a similar approach to the CONSEQ rule in Hoare logic. Finally, it enforces an ordering on $\Gamma$ via classification comparison ensuring the classifications in the stronger state are lower than those in the weaker. As a result, any valid information flow in the weaker state will be valid in the stronger.

The IF rule restricts the classification of the guard to being *Low*, therefore avoiding potential side-channels due to different branch outcomes or timing differences. The guard result is introduced into the respective branch, along with an application of $+R$ to enforce stability. A similar approach is used for the WHILE rule, establishing a *Low* guard and encoding the loop invariant. Both of these rules are not immediately applicable to automation in this form and can be specialized via combination with the CONSEQ rule as in prior work [6].

The simplest assignment rule, ASSIGNL, considers writes to *Local* variables. Due to their static *High* classification,

$$\textsc{Skip}\ \frac{}{\vdash \{P,\Gamma\}\ \mathsf{skip}\ \{P,\Gamma\}}$$

$$\textsc{Seq}\ \frac{\vdash \{P,\Gamma\}\ c_1\ \{P',\Gamma'\} \quad \vdash \{P',\Gamma'\}\ c_2\ \{P'',\Gamma''\}}{\vdash \{P,\Gamma\}\ c_1;\ c_2\ \{P'',\Gamma''\}}$$

$$\textsc{Conseq}\ \frac{P_2,\Gamma_2 \geq P_1,\Gamma_1 \quad \vdash \{P_1,\Gamma_1\}\ c\ \{P_1',\Gamma_1'\} \quad P_1',\Gamma_1' \geq P_2',\Gamma_2'}{\vdash \{P_2,\Gamma_2\}\ c\ \{P_2',\Gamma_2'\}}$$

$$\textsc{If}\ \frac{\Gamma \vdash b : t \quad P \Rightarrow t \quad \vdash \{(P \wedge b,\Gamma) + R\}\ c_1\ \{P',\Gamma'\} \quad \vdash \{(P \wedge \neg\, b,\Gamma) + R\}\ c_2\ \{P',\Gamma'\}}{\vdash \{P,\Gamma\}\ \mathsf{if}\ (b)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{P',\Gamma'\}}$$

$$\textsc{While}\ \frac{\Gamma \vdash b : t \quad P \Rightarrow t \quad \vdash \{(P \wedge b,\Gamma) + R\}\ c\ \{P,\Gamma\}}{\vdash \{P,\Gamma\}\ \mathsf{while}\ (b)\ \mathsf{do}\ c\ \{\{P \wedge \neg\, b,\Gamma\} + R\}}$$

$$\textsc{AssignL}\ \frac{x \notin \mathit{Global} \quad \Gamma \vdash e : t}{\vdash \{P,\Gamma\}\ x := e\ \{\{P,\Gamma[x \to t]\} + [x := e] + R\}}$$

$$\textsc{AssignG}\ \frac{t \leq_P \mathcal{L}_G(x) \quad x \in \mathit{Global} \quad \mathsf{fall}\ P\ \Gamma\ (x := e) \quad \Gamma \vdash e : t \quad \mathsf{guar}\ P\ (x := e)}{\vdash \{P,\Gamma\}\ x := e\ \{\{P,\Gamma[x \to t]\} + [x := e] + R\}}$$

Fig. 3. Rules of the logic.

it is not necessary to perform classification comparisons for these assignments. Hence, the context is updated with the new classification $t$ for the value in $x$, followed by updates for the assignment and environment.

The second assignment rule, ASSIGNG, considers writes to *Global* variables. As this operation is constrained by the guarantee properties and information flow, it is the most intricate. First, it is necessary to consider the information flow due to the assignment, achieved via a classification comparison between the expression's classification and the guaranteed classification, $\mathcal{L}_G$.

Second, it is necessary to consider the effects of this assignment on the classifications of other variables. Consider the case of $y := r;\ x := 0$, where $\mathcal{L}(y) \equiv x = 0$. After the execution of this snippet, the global security policy will claim $y$ holds *Low* information, as $\mathcal{L}(y)$ will be true. However, this may not be the case, as $r$ may have held *High* information. Consequently, it is possible to violate the security policy indirectly, due to changes in classifications, rather than direct information flow.

To account for this, we enumerate all variables whose global security policy may be influenced by $x := e$, achievable by inspecting the free variables referenced in their respective $\mathcal{L}$ predicates. Given an influenced variable $y$, we can establish a rising or equal classification across the assignment $x := e$ via

the comparison $\mathcal{L}(y) \leq_{:P} \mathcal{L}(y)[e/x]$. If this does not hold, the global classification of $y$ may be falling. Hence, it is necessary to restrict the classification of its value, corresponding to $r$ in the prior example. Given this classification is *Low* whenever the new classification of $y$ is *Low*, the indirect leak can be prevented. Again, this is established via a classification comparison $\Gamma\langle y\rangle \leq_{:P} \mathcal{L}(y)[e/x]$. We merge these comparisons, attaining the definition fall.

$$\begin{aligned} &\mathsf{fall}\ P\ \Gamma\ (x := e) \equiv \\ &\quad \forall y \cdot x \in \mathsf{vars}\ \mathcal{L}(y) \Rightarrow (\Gamma\langle y\rangle \vee \mathcal{L}(y)) \leq_{:P} \mathcal{L}(y)[e/x] \end{aligned} \tag{22}$$

The third proof obligation pertains to the component's guarantee $G$, which has been described in Section V-D.

### F. Soundness

The compositional theory in Section IV and the logic's rules have been encoded in Isabelle/HOL, building off prior work from [5] and [6]. The encoding and proof of soundness total ∼3K lines. We have shown the logic establishes a local bisimulation, which satisfies the constraints seen in secure. The bisimulation enforces a local security policy $\mathcal{L}_\Gamma \equiv \lambda x \cdot \mathcal{L}(x) \vee \Gamma\langle x\rangle$, such that $x$ is considered *Low* if it is required by the global security policy or can be shown locally via $\Gamma$.

$$\begin{aligned} &(c, mem_1)\ \mathcal{B}\ (c, mem_2) \equiv \\ &\quad \vdash \{P, \Gamma\}\ c\ \{P', \Gamma'\} \wedge \mathsf{context\_wf}\ P\ \Gamma \wedge \\ &\quad \mathsf{eval}\ mem_1\ P \wedge \mathsf{eval}\ mem_2\ P \wedge \\ &\quad mem_1 =^{\mathcal{L}_\Gamma} mem_2 \end{aligned} \tag{23}$$

The bisimulation is clearly symmetric, due to the symmetry of low equivalence. Moreover, it is possible to establish stability due to the wellformedness property. Establishing the bisimulation property is an involved process, consisting of an induction over the language's small-step semantics. The logic's proof obligations are then sufficient to establish the various intermediate states satisfy the component's guarantees and re-establish the bisimulation.

Therefore, the global security property can be established via analysis of individual components and proof of compatibility for the externally provided rely/guarantee conditions and security policies, coupled with Theorem 1. The full encoding is available at https://bitbucket.org/n_coughlin/rg-if/src/master/.

## VI. Application Revisited

As a demonstration of the logic, we apply it to the example from Section II, partially reproduced in Figure 4. Each operation is treated as being run by an individual component, with default initial conditions. We introduce an initial predicate $\exists n.\ z = 2 * n$ for **secret_write** only.

We first establish the security policy, along with rely and guarantee conditions. Data flows from one of the two inputs *high_in* or *low_in* to one of the two outputs *high_out* or *low_out*. The security policy should prevent the flow of information from *high_in* to *low_out*. To express this, we establish $\mathcal{L}(high\_in) \equiv \mathcal{L}(high\_out) \equiv False$ and $\mathcal{L}(low\_in) \equiv \mathcal{L}(low\_out) \equiv True$. Communication between the two threads occurs via $x$ and $z$, such that $x$ holds *High* information when $z$ is odd and *Low* otherwise. This can be expressed as $\mathcal{L}(x) \equiv \exists n.\ z = 2 * n$. We set $\mathcal{L}(z) \equiv True$, as it never holds *High* data.

```
write:                  read:
x := low_in
                        do
                            do
secret_write:                   r1 := z;
                            while (r1 % 2 ≠ 0)
z := z + 1;                 r2 := x;
x := high_in;           while (z ≠ r1)
...                     low_out := r2
x := 0;
z := z + 1              secret_read:
                        high_out := x
```

Fig. 4. Readers/writer example

For rely and guarantee conditions, we need to establish that $z$ is always increasing for **read** to function correctly. Moreover, it is evident that only **secret_write** will write a *High* value to $x$. As a result, we can restrict all other components from writing *High* values to $x$ in the security policy, simplifying the application of the logic to **secret_write**.

$$\begin{aligned} G_{secret\_write} &\equiv z' \geq z & \mathcal{L}_G(x)_{secret\_write} &\equiv \exists n.\ z = 2 * n \\ G_{otherwise} &\equiv z = z' & \mathcal{L}_G(x)_{otherwise} &\equiv True \\ R_{read} &\equiv z' \geq z & \mathcal{L}_R(x)_{secret\_write} &\equiv True \\ R_{secret\_write} &\equiv z' = z & \mathcal{L}_R(x)_{otherwise} &\equiv \exists n.\ z = 2 * n \\ R_{otherwise} &\equiv True \end{aligned}$$

It is straightforward to demonstrate compatibility between these specifications. For the rely/guarantee specifications, only $R_{read}$ introduces a restriction. This is obviously true given both possible $G$ specifications, as $z' \geq z \vee z = z' \Rightarrow z' \geq z$. The security policies only differ between components for $x$. In this case, it is obvious that $\mathcal{L}_R(x)_{secret\_write} \Rightarrow \mathcal{L}_G(x)_{otherwise}$ and $\mathcal{L}_R(x)_{otherwise} \Rightarrow \mathcal{L}_G(x)_{secret\_write}$. As security policies are shared for other variables, their proofs are trivial.

We will first consider the **write** component. As $x$ is a global, it is necessary to apply AssignG. These examples feature various uses of this rule, so we introduce the following structure to its proof obligations:

- Information flow: $\Gamma \vdash e : t$, $P \wedge \mathcal{L}_G(x) \Rightarrow t$
- Falling classifications: $\mathsf{fall}\ P\ \Gamma\ (x := e)$
- Guarantee: $\mathsf{guar}\ P\ (x := e)$

For $x := low\_in$, these proof obligations are trivial. The classification of the expression *low_in* is *True* via a consultation of $\mathcal{L}$. As a result, $P \wedge \mathcal{L}_G(x) \Rightarrow True$ can be discharged. As $x$ cannot influence classifications, it cannot cause a classification to fall, eliminating this proof obligation. Additionally, the guarantee for the **write** component only constrains $z$, so a

write to $x$ can be ignored. As there are no further instructions in this component, it has been verified.

We will now consider **secret_write**. As this component is more complex, we will establish the logic context between each line. We use the shorthand $x :: t$ to represent mappings in $\Gamma$.

$\{\exists n.\ z = 2 * n\}$
$z := z + 1$
$\{\exists n.\ z = 2 * n + 1 \wedge z :: \mathit{True}\}$
$x := \mathit{high\_in}$
$\{\exists n.\ z = 2 * n + 1 \wedge z :: \mathit{True} \wedge x :: \mathit{False}\}$
...
$x := 0$
$\{\exists n.\ z = 2 * n + 1 \wedge z :: \mathit{True} \wedge x :: \mathit{True}\}$
$z := z + 1$

Again, we can establish the local security property for this component through application of the AssignG rule. Considering the first assignment to $z$, we must work through the three proof obligations outlined above. For the information flow test, $\mathcal{L}(z) \equiv \mathit{True}$, resulting in a trivially *Low* expression. For the falling classification obligation, it is necessary to show the classification of $x$ is not falling. As we know $\exists n.\ z = 2*n$, we can show $\mathcal{L}(x)$ is true prior to the assignment, indicating it is not falling. Finally, we can show the value of $z$ is increasing, satisfying the guarantee. We then compute a post state, with the new value of $z$. Due to the rely condition, we can establish that no other component will modify $z$, allowing for the preservation of all information across environment steps.

For the first assignment to $x$, we only have to consider the information flow test, for reasons outlined prior in **write**. This introduces the proof obligation $\exists n.\ z = 2 * n + 1 \wedge \exists n.\ z = 2 * n \Rightarrow \mathit{False}$, as $\mathcal{L}(\mathit{high\_in}) \equiv \mathit{False}$ and $\mathcal{L}(x) \equiv \exists n.\ z = 2 * n$. This is true, due to the contradiction on the left side of the implication. The post state for this action introduces an entry for $x$ in $\Gamma$. As all other components are restricted to writing *Low* values to $x$, evident in $\mathcal{L}_R(x) \equiv \mathit{True}$, it's $\Gamma$ mapping can be preserved.

For the next assignment to $x$, we have a trivially *Low* expression. As a result, all proof obligations are straightforward to discharge. In the post state, the mapping for $x$ in $\Gamma$ is updated to *True* accordingly. Again, this mapping can be preserved due to $\mathcal{L}_R(x) \equiv \mathit{True}$.

Finally, we have the second assignment to $z$. Similar to the first assignment, it is trivial to demonstrate the information flow check and guarantee specification. However, this assignment will result in the classification of $x$ falling, as it is *High* in the pre state and *Low* in the post. Therefore, it is necessary to show $\Gamma\langle x\rangle \leq_{:P} \mathcal{L}(x)[z{+}1/z]$ which corresponds to $P \wedge \mathcal{L}(x)[z{+}1/z] \Rightarrow \Gamma\langle x\rangle$ or $(\exists n.\ z = 2{*}n{+}1) \wedge (\exists n.\ z{+}1 = 2 * n) \Rightarrow \mathit{True}$ in the current state, which is obviously true. Note that this is only possible due to the preservation of $x$ in the domain of $\Gamma$ due to the component specific security policy.

For **secret_read** we apply the AssignG rule to $\mathit{high\_out} := x$. This case is trivial, as the classification of *high_out* is *False*, discharging the proof obligation. Similar to prior cases, we can ignore the other two proof obligations.

The **read** component introduces the most complexity. It relies on the increasing value of $z$ to detect an interleaving with **secret_write** to roll back and re-attempt the read.

We introduce the following rule for do loops, which is a composition of Seq and While.

$$\textsc{Do}\ \frac{\begin{array}{cc}\Gamma' \vdash b : t & \vdash \{P, \Gamma\}\ c\ \{P', \Gamma'\} \\ P' \Rightarrow t & \vdash \{\{P' \wedge b, \Gamma'\} + R\}\ c\ \{P', \Gamma'\}\end{array}}{\vdash \{P, \Gamma\}\ \mathsf{do}\ c\ \mathsf{while}\ (b)\ \{\{P' \wedge \neg\, b, \Gamma'\} + R\}}$$

Note that, given $\{P' \wedge b, \Gamma'\} + R \geq P, \Gamma$ and the first proof over $c$, $\vdash \{P, \Gamma\}\ c\ \{P', \Gamma'\}$, it is possible to establish the second proof over $c$ via the Conseq rule. This will be the case for both loops in **read**, as they have initial conditions that correspond to the weakest logic context, in which $P$ is *True* and $\Gamma$ contains no mappings.

do
  do
    $r_1 := z$
    $\{r_1 \leq z \wedge r_1 :: \mathit{True}\}$
  while $(r_1 \% 2 \neq 0)$
  $\{r_1 \leq z \wedge r_1 \% 2 = 0 \wedge r_1 :: \mathit{True}\}$
  $r_2 := x$
  $\{r_1 \leq t_z \wedge t_z \leq z \wedge r_2 :: \exists n.\ t_z = 2 * n \wedge \cdots\}$
while $(z \neq r_1)$
$\{r_1 \leq t_z \wedge t_z \leq t'_z \wedge t'_z = r_1 \wedge t'_z \leq z \wedge r_1 \% 2 = 0 \wedge \cdots\}$
$\{r_1 \leq t_z \wedge t_z \leq r_1 \wedge r_1 \% 2 = 0 \wedge r_2 :: \exists n.\ t_z = 2 * n\}$
$\mathit{low\_out} := r_2$

For $r_1 := z$, we make use of AssignL to compute the classification of $z$ and the post state. As $\mathcal{L}(z) \equiv \mathit{True}$, the appropriate $\Gamma$ mapping for $r_1$ is introduced. Moreover, as we can only establish $z$ is incrementing, our post state $r_1 = z$ is weakened to $r_1 \leq z$. We can then discharge the remaining proof obligations for this first loop, as $\Gamma' \vdash r_1 \% 2 \neq 0 : \mathit{True}$, based on the recently introduced mapping for $r_1$. The negation of this guard is then added to our post state.

We then apply AssignL to $r_2 := x$. As $\mathcal{L}(x) \equiv \exists n.\ z = 2{*}n$, this predicate is added to $\Gamma$ for $r_2$. To enforce wellformedness over this context, whilst retaining sufficient information for the example, $+R$ must introduce a temporary variable $t_z$ to encapsulate the value of $z$ during this operation. This $t_z$ replaces references to $z$ in $P$ and $\Gamma$, including the new entry for $r_2$. Additionally, the temporary value is constrained based on the rely by introducing $t_z \leq z$. The resulting predicate is stable under an increasing $z$.

We retain $r_1 :: \mathit{True}$ in our context, resulting in $\Gamma' \vdash z \neq r_1 : \mathit{True}$, solving the remaining proof obligations for the outer loop. In the resulting post state, we gain $z = r_1$, however, this will be modified by $+R$ to enforce stability, resulting in the introduction of a new temporary $t'_z$. We simplify these redundant temporary variables for the purposes of presentation.

We can now apply the AssignG rule to $\mathit{low\_out} := r_2$. Similar to prior applications, it is not necessary to consider

the falling case or the guarantee. However, it is necessary to demonstrate $P \wedge \mathcal{L}(low\_out) \Rightarrow \Gamma\langle r_2 \rangle$. Given the current context, this will be $r_1 \leq t_z \wedge t_z \leq r_1 \wedge r_1 \% 2 = 0 \wedge True \Rightarrow \exists n.\ t_z = 2{*}n$. As the antecedent implies $r_1 = t_z$, this simplifies to $r_1 \% 2 = 0 \Rightarrow \exists n.\ r_1 = 2 * n$ which is true.

As a result, all components of the system exhibit the local security property and the rely/guarantee specifications compose. Therefore, the global security property holds.

## VII. Automation

Type system approaches to establishing noninterference benefit from a high degree of automation. Context-aware value-dependent variants do not present trivial implementations due to the necessity in establishing and maintaining a predicate $P$ throughout the analysis. Prior approaches have employed interactive reasoning in theorem provers [6] and automation via symbolic execution coupled with SMT solvers [9], such as Z3 [13], to account for this complexity. We outline a series of simplifications to facilitate similar automation.

### A. Restrictions on Variables

A significant barrier to automation is the environment step function $+R$, due to increases in $P$'s size and complexity in the form of temporary variables and addition of $R$. To compensate for this, we structure the relational predicate $R$ by introducing the set $R_{var}$. This set consists of elements of the form $(x, c, r)$, where $x$ is a variable, $c$ is a predicate and $r$ a relation on values, stating that the environment is constrained to modifying $x$ in accordance with $r$ if $c$ holds.

$$R \equiv \bigwedge_{(x,c,r) \in R_{var}} c \Rightarrow (x, x') \in r \tag{24}$$

This structure enables reasoning about the effects of the environment at an individual variable level. Therefore, we are able to modify $+R$ to only introduce temporary variables and expand $P$ on a variable-by-variable basis. Moreover, the current implementation of $+R$ enforces stability given any context, disregarding the fact that the pre-context was stable. Therefore, by inspecting the changes to the context throughout the logic's rules, it is possible to reduce the overhead of $+R$.

Rules introduce a small set of variables to the context throughout the logic. For example, the If and While rules introduce variables referenced in their boolean expressions, $\mathsf{vars}\ b$. For an assignment $x := e$, it is trivial to see the rules introduce references to variables $\{x\} \cup \mathsf{vars}\ e$. Additionally, the classification, derived via $\Gamma \vdash e : t$, may introduce additional references in $\Gamma$, in situations where free variables in $e$ resolve their classification based on $\mathcal{L}$. We refer to this set of new variable references as $\mathsf{new\_var}$.

$$\begin{aligned} &\mathsf{new\_var}\ b \equiv \mathsf{vars}\ b \\ &\mathsf{new\_var}\ (x := e) \equiv \\ &\quad \{x\} \cup \mathsf{vars}\ e \cup \bigcup_{y \in \mathsf{vars}\ e \setminus \mathsf{dom}\ \Gamma} \mathsf{vars}\ (\mathcal{L}(y)) \end{aligned} \tag{25}$$

In addition to introducing new variable references, assignments may reduce the constraints on the environment by negating the conditional $c$ portions of $R_{var}$ properties. We can determine this set of variables by comparing the conditions before and after the assignment, such that, given $c$ holds prior to the assignment, it should hold after. This approach is similar to the $\mathsf{fall}$ property seen in Section V-E.

$$\begin{aligned} \mathsf{weaker}\ P\ (x := e) \equiv \{y \cdot \exists\, c\ r \cdot \\ (y, c, r) \in R_{var} \wedge \neg\, (P \wedge c \Rightarrow c[e/x])\} \end{aligned} \tag{26}$$

For all variables not in $\mathsf{new\_var}$ and $\mathsf{weaker}$, we can therefore show properties constraining their values in $P$ and $\Gamma$ are stable in the pre-context and their relations still hold. Given these relations are transitive, as required by the constraint on $\mathcal{R}$, the stable properties in the pre-context will continue to hold on the post-context. Consequently, we do not have to introduce new temporary variables or extend the predicate $P$ with the constrained relations for these variables. It is only necessary to consider those variables in $\mathsf{new\_var}$ and $\mathsf{weaker}$ to maintain wellformedness.

Moreover, we introduce a special case for the variables unmodified by the environment, under the current predicate $P$. These variables do not require new temporaries or condition relations, as they are trivially equivalent.

$$\mathsf{equal}\ P \equiv \{y \cdot \exists\, c \cdot (y, c, \mathcal{I}) \in R_{var} \wedge P \Rightarrow c\} \tag{27}$$

where $\mathcal{I}$ is the identity relation.

Given these definitions, we restrict the domain of $m$, a mapping from *Global* variables to fresh temporaries, to those in $\mathsf{new\_var}$ and $\mathsf{weaker}$, but not $\mathsf{equal}$. This mapping is then used in the definition of the $+R$ operation, as defined in Section V-D. Moreover, we restrict the expansion of $P$ to only those conditional relations that are required.

$$P + R \equiv P[\mapsto m] \wedge \bigwedge_{(x,c,r) \in R'_{var}} c \Rightarrow (m\ x, x) \in r \tag{28}$$

where $R'_{var} \equiv \{(x, c, r) \cdot (x, c, r) \in R_{var} \wedge x \in \mathsf{dom}\ m\}$

While this approach presents some expansion of the context, it can be fine tuned on a variable-by-variable basis. We do not explore such optimisations.

### B. Rely/Guarantee Invariants

We introduce support for $R_{inv}$, an invariant maintained throughout execution. The conjunction of this predicate and the prior $R_{var}$ therefore forms the full relation predicate $R$. Introducing this to $P$ at the application of $+R$ will obviously result in dramatic increases to predicate size. Instead, we modify all applications of $P \Rightarrow Q$ and $t \leq_{:P} t'$ to consider $P \wedge R_{inv}$. As this approach fails to preserve restrictions due to the invariant on variable definitions prior to reassignment, this optimisation may result in their rejection, where they would have been accepted by the general logic. In these cases, $R_{inv}$ may be injected into $P$ via the Conseq rule.

### C. Security Policies

Due to the verbosity of two security policies per component, in addition to a global security policy, it can be overwhelming to specify the system's security properties and demonstrate

the compatibility requirements. This can be simplified by having an external source provide only the global security policy $\mathcal{L}$ initially, and defaulting both $\mathcal{L}_G$ and $\mathcal{L}_R$ to use this mapping. Moreover, this default policy immediately satisfies compat_sec, discharging the compatibility proof.

## VIII. Related Work

Several prior works have established sound type systems for demonstrating noninterference, starting with Volpano and Smith [2]. Moreover, many of these techniques have been employed for systems exhibiting concurrency [3] with various scheduler constraints [14], [15]. Our proposed logic enforces *timing-sensitive* noninterference, as detailed in [8]. Consequently, we do not consider the probabilistic schedulers explored in these works.

Mantel et al. [5] proposed a type system capable of enforcing a static security policy via compositional analysis. To support a greater number of secure programs, this approach facilitated read/write annotations on variables. These annotations, whilst trivial to apply, introduced a complex proof obligation to establish their compatibility. Later work has explored automatic proof of this property, via guarantee generation coupled with synchronisation via locks [16]. Additionally, Li et al. [17] used rely/guarantee to enable compositional reasoning about information flow in a message passing system.

Multiple authors have explored techniques for establishing value-dependent noninterference, with a particular focus on dependent type systems. This has included encodings withing existing dependent type systems [18] [19], as well as approaches specialised to information flow [20] [21].

Murray et al. [6] introduced a compositional dependent-type system for verifying value-dependent noninterference, initially employing read/write annotations and later coupling these permissions with locking primitives in Covern [8], removing the need for a complex external proof of compatibility. Moreover, this work allowed for rely/guarantee invariants to be coupled with locks, restricted to predicates over a single state. Hence, it is not possible to express information flow between components, beyond a global security policy, or general rely/guarantee conditions.

Ernst and Murray later introduced SecCSL [9], a concurrent separation logic for proving value-dependent information flow targeting low-level programs. Building on separation logic, it is capable of reasoning about pointers and arrays. Additionally, it introduces a relational semantics for its assertions, allowing for the coupling of classification and state predicates. Coupling these assertions with locking primitives allows for the specification of information flow properties between components. However, similar to Covern, this work is constrained to synchronisation on these lock primitives. An interesting feature of this work is its automation via symbolic execution coupled with SMT solvers. This approach has been used by other projects in enforcing information flow properties [22] [23], however, these do not reason about concurrency.

Schoepe et al. [7] have recently detailed an alternative approach to establishing value-dependent noninterference in Veronica, which decouples the functional correctness of a program from its information flow analysis. Consequently, external systems for analysis and program reasoning can introduce behaviour annotations throughout a program, which a simple information flow analysis can then exploit to demonstrate noninterference. The technique has been applied with a coarse locking approach, based on Owicki-Gries and rely/guarantee reasoning. Consequently, its application to non-blocking algorithms has not been clearly explored, with questions as to how appropriate annotations may be generated and verified in such a scenario.

The RGSim framework [24] defines a similar compositional theory to the one detailed in this paper, merging a simulation definition with rely/guarantee relations over pairs of states. Consequently, they are able to demonstrate the preservation of properties across program transformations. This technique has been extended by Murray et al. [6] to demonstrate the preservation of noninterference properties across refinement.

Gordon et al. [25] proposed a similar approach for statically associating rely/guarantee conditions to variables, in RGRef. This work couples rely/guarantee conditions with references, encoding rely/guarantee relations over their objects, which are then enforced via a type system. Such an approach provides a potential alternative to the rely/guarantee conditions we introduce.

## IX. Conclusion

We have presented a logic for the compositional verification of value-dependent noninterference, for concurrent programs with shared memory and fine-grained synchronisation, such as non-blocking algorithms. The logic is sufficiently general to capture complex rely/guarantee conditions. Moreover, it supports security policies at a component level, capturing information flow within a system to establish a global security property. Additionally, the logic is self-contained, requiring only external proofs of compatibility for its rely/guarantee conditions. Finally, we introduce optimisations for automation, which reduce the complexity of the logic, at the expense of rely/guarantee generality.

This work does not consider the implications of pointers in concurrent algorithms, essential when considering realistic programs. Such issues have been captured in related work via separation logic [26]. Coupling these two approaches may lead to techniques applicable to realistic implementations of non-blocking algorithms.

It is well known that compilers may not preserve noninterference properties shown at the language level [27], [28]. Additionally, modern architectures are capable of invalidating noninterference properties due to their weak memory models, capable of reordering memory operations [29]–[31]. Consequently, there is also potential to explore extensions or applications of the logic for more realistic execution environments.

## Acknowledgment

This work was supported by Australian Research Council Discovery Grant DP160102457.

## References


[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, 1982*, pp. 11–20, IEEE Computer Society, 1982.

[2] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, p. 167–187, Jan. 1996.

[3] G. Smith and D. M. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (D. B. MacQueen and L. Cardelli, eds.), pp. 355–364, ACM, 1998.

[4] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, pp. 321–332, 1983.

[5] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011*, pp. 218–232, IEEE Computer Society, 2011.

[6] T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *IEEE 29th Computer Security Foundations Symposium, CSF 2016*, pp. 417–431, IEEE Computer Society, 2016.

[7] D. Schoepe, T. Murray, and A. Sabelfeld, "VERONICA: expressive and precise concurrent information flow security (extended version with technical appendices)," *CoRR*, vol. abs/2001.11142, 2020.

[8] T. C. Murray, R. Sison, and K. Engelhardt, "COVERN: A logic for compositional verification of information flow control," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pp. 16–30, IEEE, 2018.

[9] G. Ernst and T. Murray, "SecCSL: Security concurrent separation logic," in *Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings, Part II* (I. Dillig and S. Tasiran, eds.), vol. 11562 of *Lecture Notes in Computer Science*, pp. 208–230, Springer, 2019.

[10] M. Moir and N. Shavit, "Concurrent data structures," in *Handbook of Data Structures and Applications*. (D. P. Mehta and S. Sahni, eds.), Chapman and Hall/CRC, 2004.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[12] H. Boehm, "Can seqlocks get along with programming language memory models?," in *Proceedings of the 2012 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '12* (L. Zhang and O. Mutlu, eds.), pp. 12–20, ACM, 2012.

[13] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[14] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00*, pp. 200–214, IEEE Computer Society, 2000.

[15] G. Boudol and I. Castellani, "Noninterference for concurrent programs and thread systems," *Theor. Comput. Sci.*, vol. 281, no. 1-2, pp. 109–130, 2002.

[16] H. Mantel, M. Müller-Olm, M. Perner, and A. Wenner, "Using dynamic pushdown networks to automate a modular information-flow analysis," in *Logic-Based Program Synthesis and Transformation* (M. Falaschi, ed.), pp. 201–217, Springer International Publishing, 2015.

[17] X. Li, H. Mantel, and M. Tasch, "Taming message-passing communication in compositional reasoning about confidentiality," in *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017* (B. E. Chang, ed.), vol. 10695 of *Lecture Notes in Computer Science*, pp. 45–66, Springer, 2017.

[18] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010* (A. D. Gordon, ed.), vol. 6012 of *Lecture Notes in Computer Science*, pp. 529–549, Springer, 2010.

[19] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," *J. Funct. Program.*, vol. 23, no. 4, pp. 402–451, 2013.

[20] H. Chen, A. Tiu, Z. Xu, and Y. Liu, "A permission-dependent type system for secure information flow analysis," in *31st IEEE Computer Security Foundations Symposium, CSF 2018*, pp. 218–232, IEEE Computer Society, 2018.

[21] A. Nanevski, A. Banerjee, and D. Garg, "Dependent type theory for verification of information flow and access control policies," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 2, pp. 6:1–6:41, 2013.

[22] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019* (T. Brecht and C. Williamson, eds.), pp. 225–242, ACM, 2019.

[23] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang, "Nickel: A framework for design and verification of information flow control systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 287–305, USENIX Association, Oct. 2018.

[24] H. Liang, X. Feng, and M. Fu, "A rely-guarantee-based simulation for verifying concurrent program transformations," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012* (J. Field and M. Hicks, eds.), pp. 455–468, ACM, 2012.

[25] C. S. Gordon, M. D. Ernst, and D. Grossman, "Rely-guarantee references for refinement types over aliased mutable data," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13* (H. Boehm and C. Flanagan, eds.), pp. 73–84, ACM, 2013.

[26] V. Vafeiadis and M. Parkinson, "A marriage of rely/guarantee and separation logic," in *CONCUR 2007 – Concurrency Theory* (L. Caires and V. T. Vasconcelos, eds.), pp. 256–271, Springer Berlin Heidelberg, 2007.

[27] V. D'Silva, M. Payer, and D. X. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015*, pp. 73–87, IEEE Computer Society, 2015.

[28] R. Sison and T. Murray, "Verifying that a compiler preserves concurrent value-dependent information-flow security," in *10th International Conference on Interactive Theorem Proving, ITP 2019* (J. Harrison, J. O'Leary, and A. Tolmach, eds.), vol. 141 of *LIPIcs*, pp. 27:1–27:19, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[29] J. A. Vaughan and T. D. Millstein, "Secure information flow for concurrent programs under Total Store Order," in *25th IEEE Computer Security Foundations Symposium, CSF 2012* (S. Chong, ed.), pp. 19–29, IEEE Computer Society, 2012.

[30] H. Mantel, M. Perner, and J. Sauer, "Noninterference under weak memory models," in *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pp. 80–94, IEEE Computer Society, 2014.

[31] G. Smith, N. Coughlin, and T. Murray, "Value-dependent information-flow security on weak memory models," in *Formal Methods – The Next 30 Years* (M. H. ter Beek, A. McIver, and J. N. Oliveira, eds.), pp. 539–555, Springer International Publishing, 2019.

## APPENDIX A
## SOUNDNESS PROOF

We provide a high level description of the soundness proof discussed in Section V-F, with more detail included in the Isabelle/HOL encoding. This proof demonstrates that the sequential logic establishes the relation defined in (23) (reiterated below) and that this relation satisfies the bisimulation properties of a secure component, defined as secure in (8).

$$
\begin{aligned}
&(c, mem_1)\ \mathcal{B}\ (c, mem_2) \equiv \\
&\quad \vdash \{P, \Gamma\}\ c\ \{P', \Gamma'\} \wedge \\
&\quad \mathsf{context\_wf}\ P\ \Gamma \wedge \\
&\quad \mathsf{eval}\ mem_1\ P \wedge \mathsf{eval}\ mem_2\ P \wedge \\
&\quad mem_1 =^{\mathcal{L}_\Gamma} mem_2
\end{aligned}
$$

It is necessary to show the three components of secure given the relation $\mathcal{B}$:

- The relation must be symmetric.
- The relation must be stable given the component's rely $\mathcal{R}_p$, as defined in (7).
- The relation must be a valid bisimulation that conforms to the component's guarantee $\mathcal{G}_p$, as defined in (6).

We will focus on each of these cases individually.

### A. Symmetry

The simplest property is symmetry. Given $(c, mem_1)\ \mathcal{B}\ (c, mem_2)$ and the definition of the bisimulation above, it is trivial to establish the necessary properties for $(c, mem_2)\ \mathcal{B}\ (c, mem_1)$. Only the proof of $mem_1 =^{\mathcal{L}_\Gamma} mem_2 \Rightarrow mem_2 =^{\mathcal{L}_\Gamma} mem_1$ introduces some complexity, however, this is obviously the case when considering the expansion of (3).

### B. Stability

Next, we consider stability. Expanding (7), we must demonstrate $(c, mem_1')\ \mathcal{B}\ (c, mem_2')$ given $(c, mem_1)\ \mathcal{B}\ (c, mem_2)$ for some $mem_1'$ and $mem_2'$ such that $((mem_1, mem_2), (mem_1', mem_2')) \in \mathcal{R}_p$.

The first two properties of the relation are preserved across this memory change as they are not dependent on $mem_1$ or $mem_2$. Due to context_wf, the predicate $P$ is known to be stable in $\mathcal{R}$. Therefore, it is straightforward to establish eval $mem_1'\ P$ and eval $mem_2'\ P$, based on the definition of stability.

To demonstrate the low equivalence property $mem_1' =^{\mathcal{L}_\Gamma} mem_2'$, where $\mathcal{L}_\Gamma \equiv \lambda x \cdot \mathcal{L}(x) \vee \Gamma\langle x \rangle$, we first unfold the definitions of low equivalence. This simplifies to a proof of $mem_1'\ x = mem_2'\ x$ given $\mathcal{L}_\Gamma(x)$ for any $x$.

Eliminating the disjunction in $\mathcal{L}_\Gamma$, we first consider the case where $\mathcal{L}(x)$ is true. Therefore, the variable $x$ is known to be *Low* via the global security policy $\mathcal{L}$, which $\mathcal{R}_p$ enforces on the primed memories. Consequently, we can establish equivalence between the two primed memories for $x$.

In the alternative case, $\Gamma\langle x \rangle$ is true. If $x$ is not in the domain of $\Gamma$, then $\Gamma\langle x \rangle = \mathcal{L}(x)$ and the prior case proof holds. Otherwise, $x$ is in the domain of $\Gamma$ and, therefore, in the set low_or_eq $P$ based on the wellformedness property. Moreover, as we known $\Gamma(x)$ and $mem_1 =^{\mathcal{L}_\Gamma} mem_2$, we can show $mem_1\ x = mem_2\ x$. Therefore, either the environment does not modify $x$, in which case $mem_1\ x = mem_2\ x \Rightarrow mem_1'\ x = mem_2'\ x$, or the environment guarantees to only write *Low* information to $x$, which can be rephrased as $mem_1'\ x = mem_2'\ x$.

As a result, all properties necessary to establish $(c, mem_1')\ \mathcal{B}\ (c, mem_2')$ can be shown and the relation $\mathcal{B}$ can be considered stable under $\mathcal{R}_p$.

### C. Bisimulation

Finally, it is necessary to show that the relation is a bisimulation, such that a step from one configuration implies a step from the other and the resultant states remain within the relation. Additionally, it is necessary to show that this transition conforms to the guarantee $\mathcal{G}_p$.

This is achieved by inducting over the definition of the language semantics for the known configuration step. A majority of these cases are trivial, with the exception of assignments. For such instructions, it is necessary to:

1) Show the result of computing the strongest post-condition of $P$ corresponds to the new memories.
2) Show the application of $+R$ results in a context that is wellformed.
3) Show that the security policy $\mathcal{L}_\Gamma$ holds between the two modified memories.
4) Show the security policy $\mathcal{L}_G$ holds for the written variable.
5) Show the assignment conforms to the guarantee $\mathcal{G}$.

The first properties are straightforward, as the strongest post-condition operation follows the standard approach and the application of $+R$ has been detailed in Section V-D. The proof obligations for the ASSIGNG and ASSIGNL rules provide sufficient information to demonstrate the remaining properties. For example, the classification comparisons ensure a *High* expression is never written to a *Low* variable. Moreover, a *Low* expression must have the same result given either configuration. Therefore, if the written variable is *Low*, the written value must be equal between the resultant memories. This corresponds to item (4) in the list, as well as trivial cases for item (3). To fully demonstrate (3), it is necessary to consider cases of a classification change, which fall addresses. Additionally, the guar proof obligation directly demonstrates item (5).

A similar approach is taken for other instructions, such as guards. Given the above properties are shown, the bisimulation can be reestablished on the new memories and the transition can be shown to conform to $\mathcal{G}_p$.

### D. Composition

Given these proofs, the relation $\mathcal{B}$ can be considered a bisimulation. It is also necessary to demonstrate that the component is initially within the bisimulation. This is achieved by using the weakest initial conditions for the logic context and assuming the memories are initially equal. Therefore, a logic

judgement is sufficient to establish a component is within its bisimulation.

$$\{P_0, \Gamma_0\}\ c\ \{P, \Gamma\} \Rightarrow (c, mem)\ \mathcal{B}\ (c, mem)$$

where $P_0 \equiv True$ and $\Gamma_0$ is an empty map.

Composing these results with Theorem 1, it is possible to directly relate all significant components of the logic.

*Theorem 2:* Given sequential logic judgements for all components and a proof of compatibility between their rely/guarantee relations $\mathcal{R}_p$ and $\mathcal{G}_p$, the global information flow property can be established.

$$\frac{\mathsf{compat}\ \mathcal{R}_{ps}\ \mathcal{G}_{ps} \qquad \forall\, i < |comp| \cdot\ \ \vdash \{P_0, \Gamma_0\}\ comp[i]\ \{P_i, \Gamma_i\}\ \text{for}\ \mathcal{R}_{ps}[i], \mathcal{G}_{ps}[i]}{\mathsf{secure}_{\mathsf{G}}\ comp}$$