Backwards-directed information flow analysis for concurrent programs

Kirsten Winter, Nicholas Coughlin and Graeme Smith
Defence Science and Technology Group, Australia
School of Information Technology and Electrical Engineering, The University of Queensland, Australia

***Abstract*—A number of approaches have been developed for analysing information flow in concurrent programs in a compositional manner, i.e., in terms of one thread at a time. Early approaches modelled the behaviour of a given thread's environment using simple read and write permissions on variables, or by associating specific behaviour with whether or not locks are held. Recent approaches allow more general representations of environmental behaviour, increasing applicability. This, however, comes at a cost. These approaches analyse the code in a forwards direction, from the start of the program to the end, constructing the program's entire state after each instruction. This process needs to take into account the environmental influence on *all* shared variables of the program. When environmental influence is modelled in a general way, this leads to increased complexity, hindering automation of the analysis.**

In this paper, we present a compositional information flow analysis for concurrent systems which is the first to support a general representation of environmental behaviour and be automated within a theorem prover. Our approach analyses the code in a backwards direction, from the end of the program to the start. Rather than constructing the entire state at each instruction, it generates only the security-related proof obligations. These are, in general, much simpler, referring to only a fraction of the program's shared variables and thus reducing the complexity introduced by environmental behaviour. For increased applicability, our approach analyses value-dependent information flow, where the security classification of a variable may depend on the current state. The resulting logic has been proved sound within the theorem prover Isabelle/HOL.

I. Introduction

Confidentiality of data in computer applications can be achieved through access control mechanisms at the language [26], [38], operating system [12], [28], [40], or even hardware level [41]. Ultimately, however, confidentiality relies on a combination of such access control mechanisms and carefully designed code, for both their implementation and use. To fully support secure software development, we need to be able to detect vulnerabilities and information leaks that arise due to programmer oversights and errors.

Our work aims to provide such support for generic software components designed for combining applications running in different security domains [2]. Using such *cross-domain components* as the only connection between security domains, reduces the verification effort for confidentiality to that of these individual components. However, since such components form the basic infrastructure on which many applications will be built, they need to be designed to be highly efficient. For this reason, concurrency is important.

Over the past decade, a number of information flow logics have been developed to support security verification of concurrent code [11], [14], [20], [24], [30], [31], [36], [37]. Most of this work uses *rely/guarantee reasoning* [18] in which assumptions (or *rely* conditions) about a thread's environment are used in reasoning about that thread in isolation provided all other threads *guarantee* that the assumption holds. This provides a scalable analysis technique in which sequential analyses on individual threads are composed to demonstrate security for a concurrent program.

Each of the existing rely/guarantee-based approaches performs a *strongest postcondition* analysis of the code of each thread. Such an analysis moves forward through the code one instruction at a time to construct the strongest state that can result from the instruction's execution, and uses this to discharge the relevant security-related proof obligations. The calculation of the strongest state needs to take into account both the changes to the state resulting from the instruction and the changes to *all* shared variables that the thread's rely condition allows.

The recent approaches of Coughlin and Smith [11] and Schoepe et al. [36] allow the use of general rely (and guarantee) conditions, i.e., any predicate can be used to describe allowable state changes by the environment. In earlier approaches, rely conditions are either limited to read and write permissions on individual variables [24], [31], [37], or restricted to being associated with the acquisition and release of locks [30], limiting their applicability. For even wider applicability, the recent approaches (as well as most of the earlier ones) also support *value-dependent* information flow security in which the security classification of variables can change as the program executes [22], [29].

The complexity introduced by this combination of general rely conditions and value-dependent information flow inevitably affects the development of tool support. This has been addressed to some extent in [11] with the proposal of simplifications to the structure of rely conditions. These simplifications, however, add further burden on the developer to provide rely conditions in the required form, and also create further verification conditions for each instruction to be checked. The resulting type system that is encoded in the theorem prover Isabelle/HOL [33] can be used to establish information flow security, but requires a substantial amount of user interaction and hence limits its practicability. The work in [36], on the other hand, focusses on aspects of information flow security alone and assumes external tool

DOI 10.1109/CSF51468.2021.00017

support exists to provide program annotations required to support rely/guarantee reasoning.

In this paper, we provide an information flow logic for concurrent code which, like the work of Coughlin and Smith [11] and Schoepe et al. [36], uses general rely conditions and supports value-dependent information flow. Additionally, it is the first such logic for which there is a theorem prover encoding that can be used to establish information flow security automatically.

To facilitate this, our approach uses a backwards-directed *weakest precondition* analysis. Such an approach moves backwards through the code one instruction at a time generating the security-related proof obligations associated with each instruction, and simplifying (and ultimately discharging) proof obligations generated earlier in the analysis. Carrying only the unresolved proof obligations through the backwards analysis is less arduous in general than carrying the entire state forward as required in a strongest postcondition approach. Furthermore, the proof obligations generally range over only a fraction of the variables of the overall state space. Hence, taking into account the changes allowed by the rely condition (at *each* instruction) does not require all shared variables to be considered. Although, in general, weakest precondition analysis suffers from exponential growth of verification conditions for large or complex programs, Flanagan and Saxe [15] have shown that the complexity of verification conditions can be reduced to be *linear* in the size of the program in practice.

We are not the first to consider weakest precondition analysis for information flow. It was first proposed by Joshi and Leino [19], and has been adopted by Banajee et al. [3], [7], Barthe et al. [8], Scheben and Schmitt [35] and Balliu and Mastroeni [5], [6], among others. We are, however, the first to apply it for compositional (rely/guarantee) reasoning about concurrent programs.

The paper is structured as follows. Section II provides a motivating example which we use to illustrate concepts in the rest of the paper. Section III details the preliminary definitions used throughout. A core predicate transformer which is based on the notion of weakest preconditions, is defined in Section IV. This predicate transformer is extended to include rely/guarantee reasoning in Section V. The encoding in Isabelle and its automatic proof support is outlined in Section VI. In Section VII a formalisation of general security lattices for value-dependent information flow is proposed. Related work is surveyed in Section VIII after which the paper concludes with an outlook to further work in Section IX.

II. Motivating example

Consider the code for a concurrent object in Figure 1. The object provides synchronisation of multiple threads via a shared variable z, enabling communication of sensitive as well as non-sensitive data via a buffer variable x. We assume that trusted components can write sensitive data into the buffer variable, using the operation **sync_write** and read the buffer variable at any time using the operation **read**. Untrusted components, however, can only access the buffer variable's content via the operation **sync_read** and write to it using **write**.

```
initially  $z = 0 \wedge x = 0$

sync_write(dataType secret) :            sync_read : dataType
   while ($\neg$ CAS(z, 0, 1)) {            dataType y;
      while (z $\neq$ 0) {}                 if (CAS(z, 0, 2)) {
   }                                           y := x;
   x := secret;                                z := 0;
   ...                                         return y;
   x := 0;                                  }
   z := 0;

write(dataType data) :                   read : dataType
   x := data;                               return x;
```

Fig. 1. Spinlock-based reader/writer mechanism

The synchronisation mechanism encoded in both synchronised operations ensures that no sensitive information leaks to an untrusted component.

The encoding of the synchronisation mechanism is based on the spinlock algorithm [17] which utilises an atomic *compare-and-swap* (CAS) instruction. $\mathsf{CAS}(x, e_1, e_2)$ implements the conditional update $x := e_2$ in the case where $x = e_1$, otherwise, if $x \neq e_1$, x remains unchanged. In either case, the result of the test is returned. Note that the test and the (potential) update are executed atomically (in one step) and hence the environment cannot interfere between test and update.

Both synchronised operations attempt to set the synchronising variable z (using a CAS), which is only possible when $z = 0$. This gives them exclusive access to the buffer variable x. While the trusted writer keeps trying to set z, the untrusted reader has only one try and then gives up (similar to the try-acquire operation in the spinlock algorithm). Once z is set by the trusted writer, it can update x and (later) after clearing x's content, releases z. Similarly, when the untrusted reader sets z (to prevent concurrent updates to x), it reads x's content and releases z.

This provides us with an example for value-dependent security policies: x's security classification is dependent on z's value to enable it's use as both a sensitive and non-sensitive buffer. Furthermore, since variables x and z are shared between threads, the example requires an information flow logic which supports concurrency. In order to prove that no information leak can occur, both the trusted writer and untrusted reader need to know that variable z will only be updated by its environment when it evaluates to zero. Each of the threads needs to guarantee that this constraint on z is satisfied.

III. Security classification and levels

We let *Prog* represent the set of all programs (threads) executing instructions (skip, assignments, conditionals and loops) over program variables and expressions. Note that expressions include Boolean or arithmetic expressions as well as CAS constructs (as used in the motivating example). Variables of a program are either global and can be shared with other (concurrent) programs, or local, i.e., $Var = Global \cup Local$ where $Global \cap Local = \varnothing$. An evaluation of all variables provides a notion of *state*, $\sigma : Var \rightarrow Val$. We let Σ denote the overall state space.

Note that in our analysis we assume that assignments and the evaluation of expressions execute atomically (i.e., without interference from the environment). This can be achieved through code transformations to ensure that multiple global variables in an expression are loaded into local variables before being referenced.

For simplicity of presentation, we assume a two-valued security lattice, however, the treatment of general security lattices in the context of value-dependent information flow is outlined in Section VII. The two-valued security lattice $\mathcal{S}ec$ comprises the values *low* and *high* such that $low \sqsubseteq high$ and $high \not\sqsubseteq low$, the join operator $\sqcup$ such that $low \sqcup high = high$, and the meet operator $\sqcap$ such that $low \sqcap high = low$. The value *high* denotes information which is sensitive, and *low* denotes information which is not. Each value in a program has a *security level*, and each variable has a *security classification* capturing the highest level of information the variable is allowed to hold.

In the following we encode the two-valued security lattice as a Boolean lattice, where *true* represents low and *false* high, and represent security levels by Boolean variables, the join operator by conjunction and the meet operator by disjunction.

To model a security classification of a variable, a program has a function $\mathcal{L}$ (also referred to as the *security policy*) which maps variables to predicates:

$$\mathcal{L} : Var \rightarrow Pred$$

where *Pred* is the set of all predicates. The mapping to predicates allows value-dependent security classifications: to maintain information flow security, variable x can only hold low information in those states where $\mathcal{L}(x)$ evaluates to *true*, whenever $\mathcal{L}(x)$ evaluates to *false* x can hold both high and low information. Note that implementing value-dependent information flow security requires any program to always clear sensitive values from a variable prior to its classification falling from high to low. Our analysis is geared to find places in the program where this requirement is not met. For example, the security classification for the code in Figure 1 would prescribe that the variable x is low whenever variable z is not equal to 1, otherwise it is classified as high, i.e., $\mathcal{L}(x) = (z \neq 1)$.

For any variable that never changes its classification, either *true* (i.e., always low) or *false* (i.e., always high) may be used. In Figure 1, for example, $\mathcal{L}(z) = true$. We assume $\mathcal{L}(v) = false$ for all $v \in Local$ (as local variables are not readable by the environment). Note that, for analysis purposes, the statement *return y* in **sync_read** and *return x* in **read** are treated as assignments of y and x to global variables, with security classification *true* in the case of **sync_read** and *false* in the case of **read**.

The variables occurring in $\mathcal{L}(v)$ for some variable v, are called *control variables*, and the set of all control variables is denoted by $\mathcal{C}trl$. To enable communication of a shared variable's classification between threads, control variables are always global, i.e., $\mathcal{C}trl \subseteq Global$. A function $ctrled : Var \rightarrow \mathcal{P}Var$ provides, for each control variable, the set of *controlled* variables. Given the security classifications for the variables in Figure 1 above, we have $\mathcal{C}trl = \{z\}$ and $ctrled(z) = \{x\}$. Note that we assume that no variable controls its own security classification, i.e., $\forall x \in \mathcal{C}trl.\, x \notin ctrled(x)$.

The security level of a value held by or assigned to a variable has the type of the security lattice. This security level can be retrieved via a mapping $\gamma : Var \rightarrow Bool$, in our setting. For simplicity we use the notation Γ_x to denote $\gamma(x)$. Note that in contrast to much of the previous work on value-dependent information flow [11], [30], [31], [37], we treat this function as an *auxiliary* variable of the program's global state. This allows us to refer to it in predicates on the program's state, and allows it to be shared between threads. The result is a pure program logic which reasons over state alone (rather than a context of state/type pairs).

When a variable's value is updated, its security level is updated to reflect the security levels of the variables in the expression to which it is updated. We make the assumption at this point that all assignments are checked to never assign a *high* value to a variable with *low* classification (which is guaranteed by our predicate transformer defined in the following sections).

Let *Exp* denote the set of all expressions. The security level of an expression is defined by the function $\Gamma_{\mathrm{E}} : Exp \rightarrow Pred$ with

$$\Gamma_{\mathrm{E}}(e) \mathrel{\widehat{=}} \bigwedge_{v \in vars(e)} (\Gamma_v \vee \mathcal{L}(v))$$

where $vars : Exp \rightarrow \mathcal{P}Var$ returns the set of variables occurring in an expression and, as introduced above, $\Gamma_v = \gamma(v)$.

IV. Weakest precondition with security assurance

In our analysis we want to assert that high security values (which are sensitive) do not flow into variables with low security classifications (which may be publicly visible). The analysis follows the program code in a backwards fashion (i.e., starting at the end) and is based on computing weakest preconditions [13].

To achieve guarantees on the security of a concurrent program, an analysis needs to assure that no secret-dependent timing differences are present in the code. Otherwise, two threads which are deemed secure by themselves can still reveal secrets when combined (a detailed discussion can be found in [30], [37]). To uncover the possibility of secret-dependent timing differences, our analysis requires that branch conditions do not depend on high information. This can be achieved, if necessary, using program transformations (e.g., [1], [27]). Hence, we are not concerned in our analysis with the termination of loops. On the other hand, exceptions caused by evaluating an undefined expression may cause an information leak. For example, $x := y/z$ where x and z are high variables is secure when $z \neq 0$, but if $z = 0$ an attacker might be able to observe the change in behaviour due to the exception being thrown.

We base our transformer on a modified weakest precondition wp' which does not establish the termination of loops (via a loop variant), but does require expressions and branch conditions to be defined. Furthermore, the post-condition Q might contain references to the security level of the value in

x, Γ_x, which we have introduced as an auxiliary state variable, as well as references to x itself.

The modified transformer wp' is defined as follows. Let $\mathsf{ite}(b, c_1, c_2)$ abbreviate $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. and $\mathsf{while}(b, c)$ abbreviate $\mathsf{while}\ b\ \mathsf{do}\ c$.

$$\begin{aligned}
wp'(skip, Q) &\mathrel{\widehat{=}} Q \\
wp'(x := e, Q) &\mathrel{\widehat{=}} def(e) \wedge Q[x \leftarrow e, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(e)] \\
wp'(\mathsf{ite}(b, c_1, c_2), Q) &\mathrel{\widehat{=}} def(b) \wedge (b \Rightarrow wp'(c_1, Q)) \wedge \\
&\qquad (\neg b \Rightarrow wp'(c_2, Q)) \\
wp'(\mathsf{while}(b, c), Q) &\mathrel{\widehat{=}} Inv \wedge (\forall \sigma.(Inv \Rightarrow def(b)) \wedge \\
&\qquad (Inv \wedge b \Rightarrow wp'(c, Inv)) \wedge (Inv \wedge \neg b \Rightarrow Q)) \\
wp'(c1;\ c2, Q) &\mathrel{\widehat{=}} wp'(c1, wp'(c2, Q))
\end{aligned}$$

where the notation $Q[x \leftarrow e, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(e)]$ denotes that all free occurrences of x and Γ_x in Q are substituted by e and $\Gamma_{\mathrm{E}}(e)$. In the definition of the while statement, *Inv* is the loop invariant which needs to hold before the loop and has to imply the definedness of the loop condition in all states σ, be preserved by the loop body in all possible states (if the loop condition b is satisfied), and needs to imply the post-condition Q in all states when the loop is exited (i.e., when the loop condition does not hold). Intuitively, *Inv* can be used as a summary of the loop's behaviour, given that it can be shown that it is preserved during loop iteration and leads to the post-condition once the loop terminates.

Special consideration should be taken for CAS constructs, which are Boolean expressions with a side-effect: a test on a variable is atomically linked with its potential update (in case of a positive test). In particular, the atomicity of the construct will become relevant when interference between threads is considered.

CAS expressions can be used anywhere in the code where Boolean expressions are used because they return a Boolean value. Hence they can occur on the right-hand side of assignments, and also within guards in conditionals and loops.

For sequential programs, the atomicity of the CAS test and update does not affect the weakest precondition computation. An assignment of a CAS expression can hence be treated similar to a conditional update of two variables (in case of a positive test) or one variable (in case of a negative test). For conditionals and loops using CAS expressions, the transformer can be defined similarly.

$$\begin{aligned}
&wp'(y := \mathsf{CAS}(x, e_1, e_2), Q) \mathrel{\widehat{=}} \\
&\qquad wp'(\mathsf{ite}(x = e_1, (x := e_2;\ y := true), y := false), Q) \\
&wp'(\mathsf{ite}(\mathsf{CAS}(x, e_1, e_2), c_1, c_2), Q) \mathrel{\widehat{=}} \\
&\qquad wp'(\mathsf{ite}(x = e_1, (x := e_2;\ c_1), c_2), Q) \\
&wp'(\mathsf{while}(\mathsf{CAS}(x, e_1, e_2), c), Q) \mathrel{\widehat{=}} \\
&\qquad wp'(\mathsf{while}(x = e_1, (x := e_2;\ c)), Q)
\end{aligned}$$

Some special attention needs to be paid to instructions that refer to the negation of a CAS instruction, since the updating behaviour remains the same but the return value is negated.

$$\begin{aligned}
&wp'(y := \neg\mathsf{CAS}(x, e_1, e_2), Q) \mathrel{\widehat{=}} \\
&\qquad wp'(\mathsf{ite}(x = e_1, (x := e_2;\ y := false), y := true), Q) \\
&wp'(\mathsf{ite}(\neg\mathsf{CAS}(x, e_1, e_2), c_1, c_2), Q) \mathrel{\widehat{=}} \\
&\qquad wp'(\mathsf{ite}(x = e_1, (x := e_2;\ c_2), c_1), Q) \\
&wp'(\mathsf{while}(\neg\mathsf{CAS}(x, e_1, e_2), c), Q) \mathrel{\widehat{=}} \\
&\qquad wp'((\mathsf{while}(x \neq e_1, c);\ x := e_2), Q)
\end{aligned}$$

For presentation purposes, in the rest of this paper we assume that all expressions are defined, i.e., $def(e) = true$ in the rules above. In practice, definedness can be checked statically as is done in other program analysis tools (e.g., [21]).

A. A predicate transformer for information flow security

In our context, we additionally generate proof obligations at each step ensuring that information flow security is not violated by the currently analysed instruction. The additional proof obligations are predicates which are conjoined with the weakest precondition. We define a new transformer which is based on wp', called the *weakest precondition for information flow* (*wpif*). For an instruction, c, $wpif(c, Q)$ is of the form

$$PO(c) \wedge wp_Q(c, Q)$$

where $PO(c)$ is an additional proof obligation that needs to be resolved by an earlier instruction, and $wp_Q(c, Q)$ is a modified transformation of the post-condition Q as defined below.

B. Assignment

For a simple assignment instruction, $x := e$, the generated proof obligation distinguishes whether the variable x is global (and hence accessible by other threads) and whether it is a control variable.

For global variables the proof obligation ensures that the security level of the value to be assigned, $\Gamma_{\mathrm{E}}(e)$, does not exceed the security classification of the variable it is assigned to, $\mathcal{L}(x)$. Since both are predicates which represent low when they are *true*, this holds when the latter implies the former. For control variables a further proof obligation checks if the assignment causes any information leaks from the variables *controlled by* the updated variable.

We define the predicate transformer for assignments as follows.

$$\begin{aligned}
&wpif(x := e, Q) \mathrel{\widehat{=}} PO(x := e) \wedge wp_Q(x := e, Q) \\
&\text{where} \\
&PO(x := e) \mathrel{\widehat{=}} (x \in Global \Rightarrow (\mathcal{L}(x) \Rightarrow \Gamma_{\mathrm{E}}(e))) \wedge \\
&\qquad\qquad (x \in \mathcal{C}trl \Rightarrow secureUpd(x := e)) \\
&wp_Q(x := e, Q) \mathrel{\widehat{=}} wp'(x := e, Q)
\end{aligned}$$

The second constraint of the proof obligation concerns the update of control variables. This constraint is key to *value-dependent* information flow security in which a variable's classification can be modified during execution of the program. The analysis needs to ensure that whenever such a change in classification occurs, no *high* value is exposed via a now *low* classified, hence observable, variable. The predicate *secureUpd* formalises such a check.

$$\begin{aligned}
&secureUpd(x := e) \mathrel{\widehat{=}} \\
&\qquad \forall\, y \in ctrled(x).\, \mathcal{L}(y)[x \leftarrow e] \Rightarrow \Gamma_y \vee \mathcal{L}(y)
\end{aligned}$$

This definition ensures that for each variable y controlled by x, if y's security classification after the update is *low* then either y's security level is *low* or its classification was *low* before the update (in which case it is ensured that it holds a *low* value).

C. Conditionals and loops

For a conditional statement if b then c_1 else c_2, the *wpif* rule has a similar structure to that of wp': in the case that b holds *wpif* of the first statement c_1 is computed, and in the case where $\neg b$ holds *wpif* of the second statement $c2$ is computed. The definition additionally asserts as a proof obligation that the security level of the branch condition is low to avoid secret-dependent timing differences.

$$\begin{array}{l} wpif(\mathsf{ite}(b,c_1,c_2),Q)) \mathrel{\hat{=}} \\ \quad PO(\mathsf{ite}(b,c_1,c_2)) \wedge wp_Q(\mathsf{ite}(b,c_1,c_2),Q) \\ \text{where} \\ PO(\mathsf{ite}(b,c_1,c_2)) \mathrel{\hat{=}} \Gamma_E(b) \\ wp_Q(\mathsf{ite}(b,c_1,c_2),Q) \mathrel{\hat{=}} \\ \quad (b \Rightarrow wpif(c_1,Q)) \wedge (\neg b \Rightarrow wpif(c_2,Q)) \end{array}$$

Similarly, a while loop is treated analogously to the standard wp' computation with an additional assertion on the security level of the loop condition which is to be implied in all states by the loop invariant, i.e., *Inv* guarantees a low security level. As a result, when the loop condition is true, the loop body not only maintains *Inv* but also the low security level of the loop condition.

$$wpif(\mathsf{while}(b,c),Q) \mathrel{\hat{=}} PO(\mathsf{while}(b,c)) \wedge wp_Q(\mathsf{while}(b,c))$$

where

$$\begin{array}{l} PO(\mathsf{while}(b,c)) \mathrel{\hat{=}} Inv \wedge (\forall\, \sigma.\, Inv \Rightarrow \Gamma_E(b)) \\ wp_Q(\mathsf{while}(b,c)) \mathrel{\hat{=}} \forall\, \sigma.\, (Inv \wedge b \Rightarrow wpif(c,Inv)) \wedge \\ \qquad\qquad\qquad\qquad \forall\, \sigma.\, (Inv \wedge \neg b \Rightarrow Q) \end{array}$$

D. Instructions with CAS expressions

The above definitions feed straightforwardly into the definition of the transformer for instructions with CAS expressions, which may be assignments, conditionals or loops.

As is argued for conditionals, a proof obligation on the security level of the guard is required. Since the instruction $\mathsf{CAS}(x,e_1,e_2)$ introduces the test $x = e_1$ we need to assure $\Gamma_E(x = e_1)$. The recursive call to the transformer *wpif* will generate proof obligations for the remainder of the instruction.

In the case of a CAS assignment this remainder constitutes the update of the CAS variable and the assignment of the test outcome to y.

$$\begin{array}{l} wpif(y := \mathsf{CAS}(x,e_1,e_2)) \mathrel{\hat{=}} \Gamma_E(x = e_1) \wedge \\ \quad (x = e_1 \Rightarrow wpif(x := e_2, wpif(y := true, Q))) \wedge \\ \quad (x \neq e_1 \Rightarrow wpif(y := false, Q)) \end{array}$$

which is equivalent to

$$wpif(\mathsf{ite}(x = e_1, (x := e_2;\ y := true), y := false), Q)$$

For conditionals with a CAS expression as a guard, the remainder constitutes either the sub-instructions c_1 or c_2, depending on the outcome of the CAS test, where c_1 is preceded by the CAS update.

$$\begin{array}{l} wpif((\mathsf{ite}(\mathsf{CAS}(x,e_1,e_2),c_1,c_2),Q) \mathrel{\hat{=}} \Gamma_E(x = e_1) \\ \quad (x = e_1 \Rightarrow wpif(x := e_2, wpif(c_1,Q))) \wedge \\ \quad (x \neq e_1 \Rightarrow wpif(c_2,Q)) \end{array}$$

which is equivalent to

$$wpif(\mathsf{ite}(x = e_1, (x := e_2;\ c_1), c_2), Q)$$

For a loop with a CAS expression, we need to show that the loop invariant implies $\Gamma_E(x = e_1)$. The execution of the loop body depends on the CAS test and is preceded by the CAS update.

$$\begin{array}{l} wpif(\mathsf{while}(\mathsf{CAS}(x,e_1,e_2),c),Q) \mathrel{\hat{=}} Inv \wedge \\ \quad \forall\, \sigma.\, (\,(Inv \Rightarrow (\Gamma_E(x = e_1))) \wedge \\ \qquad (Inv \wedge x = e_1 \Rightarrow wpif(x := e_2, wpif(c,Inv))) \wedge \\ \qquad (Inv \wedge x \neq e_1 \Rightarrow Q)) \end{array}$$

which is equivalent to

$$wpif(\mathsf{while}(x = e_1, (x := e_2;\ c)), Q).$$

The rules for instructions with negated CAS instructions are derived similarly.

E. Sequential composition

When we analyse a sequence of instructions, the weakest precondition transformation of each instruction c generates its own proof obligation and modifies the post-condition Q. The conjunction of these two predicates is passed on as the post-condition for preceding instructions as is the case for standard weakest precondition computations. That is,

$$wpif(c_1;\ c_2, Q) \mathrel{\hat{=}} wpif(c_1, PO(c_2) \wedge wp_Q(c_2,Q))$$

which equals $wpif(c_1, wpif(c_2,Q))$.

As we are not interested in assuring a final post-condition (but rather detecting information leakage), our backwards analysis starts with the final post-condition *true*. At each step an additional proof obligation is generated which is then passed on as part of the post-condition to the predicate transformer analysing the next instruction upwards in the program. Whenever an instruction is an assignment, these post-conditions are potentially simplified through the substitutions that apply. The resulting predicate can either be *true* (indicating that the proof obligation is satisfied), *false* in which case the analysis terminates and reports an information leak, or any other predicate that is passed on further. If in the end we are left with a predicate p, this must be satisfied by the initial state S_0, i.e., $S_0 \Rightarrow p$, for the program to be secure.

F. Example: **sync_write** under wpif

The required security policy for the code of Figure 1 is that z is always low and x is controlled by z (being low when z is not set to 1), i.e., $\mathcal{L}(z) = true \wedge \mathcal{L}(x) = (z \neq 1)$.

Applying the transformer *wpif* results in the (backwards) reasoning that is outlined in Figure 2. The process begins at the last line from the post-condition *true* (nothing specific is required at this point). The transformer generates the predicate $wpif(z := 0, true)$ which is shown in curly brackets above the last instruction. It contains the conjunction of the proof obligations at this point as well as the modified post-condition $true[z \leftarrow 0, \Gamma_z \leftarrow \Gamma_E(0)]$ which equals *true*. The condition for global variables reduces to *true* since $\Gamma_E(0)$ is *true*. Substituting $\mathcal{L}(x)$ with the given security policy in the condition for

```
$\{ \mathit{true} \}$
  $\mathbf{while}(\neg\mathsf{CAS}(\mathbf{z},\mathbf{0},\mathbf{1}))\{$
    $\{ \mathit{true} \}$
      $\mathbf{while}(\mathbf{z} \neq \mathbf{0})\ \{\mathbf{skip}\};$
    $\{ \mathit{true} \}$
  $\}$
$\{ z = 1 \}$
$\{ z \neq 1 \Rightarrow \Gamma_{\mathrm{E}}(\mathit{secret}) \}$
$\left\{\begin{array}{l}(x \in \mathit{Global} \Rightarrow (\mathcal{L}(x) \Rightarrow \Gamma_{\mathrm{E}}(\mathit{secret}))) \wedge \\ (x \in \mathcal{C}\mathit{trl} \Rightarrow \\ \quad (\forall\, y \in \mathit{ctrled}(x).\, \mathcal{L}(y)[x \leftarrow \mathit{secret}] \Rightarrow \Gamma_y \vee \mathcal{L}(y))) \wedge \\ \mathit{true}[x \leftarrow \mathit{secret}, \Gamma_x \leftarrow \Gamma_{\mathit{secret}}]\end{array}\right\}$
  $\mathbf{x} := \mathbf{secret};$
$\{ \mathit{true} \}$
$\{ (z \neq 1 \Rightarrow \Gamma_{\mathrm{E}}(0)) \wedge (\Gamma_{\mathrm{E}}(0) \vee z \neq 1) \}$
$\left\{\begin{array}{l}(x \in \mathit{Global} \Rightarrow (\mathcal{L}(x) \Rightarrow \Gamma_{\mathrm{E}}(0))) \wedge \\ (x \in \mathcal{C}\mathit{trl} \Rightarrow \\ \quad (\forall\, y \in \mathit{ctrled}(x).\, \mathcal{L}(y)[x \leftarrow 0] \Rightarrow \Gamma_y \vee \mathcal{L}(y))) \wedge \\ (\Gamma_x \vee z \neq 1)[x \leftarrow 0, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(0)]\end{array}\right\}$
  $\mathbf{x} := \mathbf{0};$
$\{ \Gamma_x \vee z \neq 1 \}$
$\left\{\begin{array}{l}\mathit{true} \wedge \\ (\mathcal{L}(x)[z \leftarrow 0] \Rightarrow \Gamma_x \vee \mathcal{L}(x)) \wedge \\ \mathit{true}\end{array}\right\}$
$\left\{\begin{array}{l}(z \in \mathit{Global} \Rightarrow (\mathcal{L}(z) \Rightarrow \Gamma_{\mathrm{E}}(0))) \wedge \\ (z \in \mathcal{C}\mathit{trl} \Rightarrow (\forall\, y \in \mathit{ctrled}(z).\, \mathcal{L}(y)[z \leftarrow 0] \Rightarrow \\ \hfill \Gamma_y \vee \mathcal{L}(y))) \wedge \\ \mathit{true}[z \leftarrow 0, \Gamma_z \leftarrow \Gamma_{\mathrm{E}}(0)]\end{array}\right\}$
  $\mathbf{z} := \mathbf{0};$
$\{ \mathit{true} \}$
```

Fig. 2. Transformer *wpif* applied to **sync_write**

control variables, allows us to simplify the whole conjunction to $\Gamma_x \vee (z \neq 1)$. This becomes the post-condition of instruction $x := 0$ and feeds into the next step. All further steps of the analysis work in a similar fashion.

Reasoning over the nested loops reveals that it is sufficient to assume *true* for both loop invariants as shown in the following. With $Q = (z = 1)$ we have

$$\begin{array}{l}\mathit{wpif}(\mathit{while}(\neg\mathsf{CAS}(z,0,1)\{\mathit{while}(z \neq 0)\{\mathit{skip}\}\}, z = 1) = \\ \quad \mathit{Inv}_1 \wedge \forall\, \sigma.\, (\mathit{Inv}_1 \Rightarrow \Gamma_{\mathrm{E}}(z = 0) \wedge \\ \qquad (\mathit{Inv}_1 \wedge z \neq 0 \Rightarrow \mathit{wpif}(\mathit{while}(z \neq 0)\{\mathit{skip}\}, \mathit{Inv}_1)) \wedge \\ \qquad (\mathit{Inv}_1 \wedge z = 0 \Rightarrow \mathit{wpif}(z := 1, z = 1))\end{array}$$

With $\mathit{Inv}_1 = \mathit{true}$ we also have

$$\begin{array}{l}\mathit{wpif}(\mathit{while}(z \neq 0)\{\mathit{skip}\}, \mathit{Inv}_1) = \\ \quad \mathit{Inv}_2 \wedge \forall\, \sigma.\, (\mathit{Inv}_2 \Rightarrow \Gamma_{\mathrm{E}}(z \neq 0) \wedge \\ \qquad (\mathit{Inv}_2 \wedge z \neq 0 \Rightarrow \mathit{wpif}(\mathit{skip}, \mathit{true})) \wedge \\ \qquad (\mathit{Inv}_2 \wedge z = 0 \Rightarrow \mathit{true})\end{array}$$

Since $\Gamma_{\mathrm{E}}(z \neq 0) = \Gamma_{\mathrm{E}}(z = 0) = \mathit{true}$, with $\mathit{Inv}_2 = \mathit{true}$ we get $\mathit{wpif}(\mathit{while}(z \neq 0)\{\mathit{skip}\}, \mathit{Inv}_1) = \mathit{true}$. Substituting this into the above results in

$$\mathit{wpif}(\mathit{while}(\neg\mathsf{CAS}(z,0,1)\{\mathit{while}(z \neq 0)\{\mathit{skip}\}\}, z = 1) = \mathit{true}$$

The result of our analysis shows that the weakest precondition for the operation **sync_write** to be secure (disregarding its environment) is *true* which indicates that independent of the initial state, no information is leaked.

The predicate transformer defined so far only assures local security of a thread and hence does not consider the impact of other threads' behaviours. The consideration of the security on the global level is treated via the rely/guarantee framework introduced in the following section.

V. Adding rely/guarantee reasoning

The rely/guarantee framework, as introduced by Jones [18], allows for a compositional approach to analysing concurrent programs which synchronise via shared variables. Assume a program consists of components (threads) which are running in parallel, $P_1 \parallel \ldots \parallel P_n$ with local variables Local_i for $i \in \{1, \ldots, n\}$. Each of these components can be analysed in isolation if we assume *rely* conditions which "summarise" the effects of all other components. This reasoning is sound if, in addition, each of the components *guarantees* a behaviour that conforms with what the other components assume.[1]

Rely and guarantee conditions are captured as relations over pre- and post-states, which have to be *reflexive*. That is, they must be valid when no step is taken, or a step is taken for which the pre- and post-state are the same. Rely conditions additionally need to be *transitive*, i.e., they have to hold over the thread behaviour as a whole, as well as over each of its instructions.

For example, assume each operation of Figure 1 is called by a different thread. For the **sync_write** operation, it is the case that no other thread changes the global variable x to a *high* value. Hence, the trusted writer's rely condition can state that if the value of x is *low* in the pre-state of any environment action, it remains *low* in that action's post-state, i.e., $\Gamma_x \Rightarrow \Gamma'_x$ (we follow the custom to denote variables in the post-state as primed variables). Also, when $z = 1$ no other thread changes z, and hence an additional rely condition is $z = 1 \Rightarrow z = z'$. Similarly, for **sync_read** we know that when $z = 2$, z cannot be changed by the environment and if x is *low*, it remains *low*. Hence, its rely condition can include $z = 2 \Rightarrow z = z' \wedge \Gamma_x \Rightarrow \Gamma'_x$. The guarantee conditions of these threads have a similar form.

Let $\mathcal{R}s$ and $\mathcal{G}s$ denote the set of rely and guarantee conditions of all components in a system. Let $\mathcal{R}_i$ and $\mathcal{G}_i$ be the rely and guarantee conditions of a specific component i. *Compatibility* of rely and guarantee conditions requires that the guarantee of each component implies the rely conditions of all other components, i.e., $\mathcal{G}_i \Rightarrow \mathcal{R}_j$ for all $i \neq j$. Note that each $\mathcal{G}_i$ has to be strong enough to satisfy all $\mathcal{R}_j$, otherwise thread i (by itself) can potentially invalidate what another thread relies on. For the overall system, compatibility is defined as follows.

$$\mathit{compatible}(\mathcal{R}s, \mathcal{G}s) = \bigwedge_{\mathcal{G}_i \in \mathcal{G}s} (\mathcal{G}_i \Rightarrow \bigwedge_{\mathcal{R}_j \in \mathcal{R}s \wedge j \neq i} \mathcal{R}_j)$$

If we have shown that $\mathcal{R}s$ and $\mathcal{G}s$ are compatible then it suffices for the overall analysis to focus on each component together with its rely and guarantee condition in isolation. In the next section, we introduce a further extension to our predicate transformer that shows information flow security of a component under rely and guarantee conditions.

[1]Note that a program with dynamic thread creation can be handled using an additional auxiliary global variable to indicate whether the dynamic thread is active, which is accessible to the rely/guarantee specification and updated at fork/join operations.

A. Stability under $\mathcal{R}$

The predicate transformer introduced so far works on a single component and is designed to create proof obligations for each instruction and at the same time simplify previously generated proof obligations where possible. In the concurrent setting, we additionally need to assure that the component's rely condition $\mathcal{R}$ maintains all proof obligations throughout the analysis. If at any point the environment, by executing one or more steps, can invalidate the proof obligations then the current thread cannot provide assurance that they are satisfied. We refer to this property of predicates as *stability under* $\mathcal{R}$ and define it as

$$stable_{\mathcal{R}}(p) \;\;\widehat{=}\;\; \forall\, v'_1, ..., v'_n.\ \mathcal{R} \wedge p \Rightarrow primed(p).$$

where $\{v_1, .., v_n\} = Var$. The predicate $primed(p)$ refers to p in the post-state, i.e., replacing every unprimed variable, v_i with its primed counterpart, v'_i. Note that for any p with no free variables, $stable_{\mathcal{R}}(p)$ obviously evaluates to *true*.

To simplify the onus on the user specifying the rely condition, we introduce the following conditions which occur in every $\mathcal{R}_i$. For a thread i, they state that (a) the environment cannot update variables local to thread i, (b) if the environment does not update a (global) variable then its security level does not change during (zero or more) environment steps, and (c) the environment is checked to be information flow secure, i.e., no global variable is assigned a value in an environment step whose security level exceeds its classification.

R1: $\forall\, v \in Local_i.\ v = v' \wedge \Gamma_v = \Gamma'_v$
R2: $\forall\, v \in Var.\ v = v' \Rightarrow \Gamma_v = \Gamma'_v$
R3: $\forall\, v \in Global.\ \mathcal{L}(v)' \Rightarrow \Gamma'_v$

B. Checks on $\mathcal{G}$

Each thread also needs to validate that each instruction meets the component's guarantee $\mathcal{G}$. We introduce a predicate $guar(\mathcal{G}, c)$ which checks the guarantee condition against instruction c. To do so, we want to substitute all primed variables in $\mathcal{G}$ with what we know about their evaluation in the post-state (i.e., after c).

This can be done using the basic predicate transformer wp' which performs the substitution of updated variables in a (non-relational) predicate. However, we first need to reshape the relational predicate $\mathcal{G}$ to be phrased in terms of variables v_0 and v for pre- and post-state variants of any variable (instead of v and v'). Similarly, the security levels are renamed. We introduce

$$\mathcal{G}' = \mathcal{G}[\forall\, v \in vars(\mathcal{G}).\ v \leftarrow v_0, v' \leftarrow v, \Gamma_v \leftarrow \Gamma_{v_0}, \Gamma_{v'} \leftarrow \Gamma_v].$$

Based on the rephrased guarantee condition $\mathcal{G}'$, the guarantee check for an instruction c substitutes the (renamed) post-state variables (via wp') and subsequently renames pre-state variables v_0 back to their original names v. This results in a non-relational predicate which can be evaluated in a state.

$$guar(\mathcal{G}, c) \mathrel{\widehat{=}} wp'(c, \mathcal{G}')[\forall\, v \in vars(\mathcal{G}).\ v_0 \leftarrow v, \Gamma_{v_0} \leftarrow \Gamma_v]$$

For example, let $\mathcal{G} = (z = 2 \Rightarrow z = z') \wedge x \geq x'$ then $\mathcal{G}' = (z_0 = 2 \Rightarrow z_0 = z) \wedge x_0 \geq x$ and hence

$$\begin{aligned}
&guar(\mathcal{G}, x := e)\\
&\quad = wp'(x := e, \mathcal{G}')[z_0 \leftarrow z, x_0 \leftarrow x, \Gamma_{z_0} \leftarrow \Gamma_z, \Gamma_{x_0} \leftarrow \Gamma_x]\\
&\quad = (z_0 = 2 \Rightarrow z_0 = z) \wedge x_0 \geq e\\
&\qquad\qquad\qquad [z_0 \leftarrow z, x_0 \leftarrow x, \Gamma_{z_0} \leftarrow \Gamma_z, \Gamma_{x_0} \leftarrow \Gamma_x]\\
&\quad = (z = 2 \Rightarrow z = z) \wedge x \geq e\\
&\quad = x \geq e
\end{aligned}$$

That is, the update of x satisfies the guarantee condition if x in the prestate is greater than or equal to expression e.

Since we assume that $\mathcal{G}$ is reflexive, it needs to be satisfied by all instructions, even those that do not update the state. The check $guar(\mathcal{G}, c)$ equals *true* whenever c does not update any variables since the renaming of pre-state variables results in equality between pre- and post-state variables.

To ensure compatibility between rely and guarantee conditions, we mirror conditions R1-R3 for guarantees $\mathcal{G}_i$. Note that G1 refers to variables that are local to other threads and furthermore that G3 below overlaps with the proof obligation for assignments, $PO(x := e)$.[2]

G1: $\forall\, v \notin (Local_i \cup Global).\ v = v' \wedge \Gamma_v = \Gamma'_v$
G2: $\forall\, v \in Var.\ v = v' \Rightarrow \Gamma_v = \Gamma'_v$
G3: $\forall\, v \in Global.\ \mathcal{L}(v)' \Rightarrow \Gamma'_v$

C. The transformer $wpif_{\mathcal{RG}}$

To account for rely and guarantee conditions, we introduce the predicate transformer $wpif_{\mathcal{RG}}$ based on the predicate transformer *wpif*. For any instruction c, in addition to the proof obligation $PO(c)$ and the transformation of the post-condition $wp_Q(c, Q)$, the transformer also generates two further conditions to prove that (a) c meets the *guarantee* $\mathcal{G}$, and (b) the pre-condition generated so far is *stable under rely* $\mathcal{R}$. The former implements the checks on $\mathcal{G}$ for each instruction c as introduced in Section V-B, while the latter ensures that even if the code in the thread delivers the pre-condition required to prevent any information leaking, the environment of the thread is not compromising this pre-condition before instruction c is performed. That is, if the pre-condition holds and the environment performs one or more steps at this point in time, the pre-condition will still hold afterwards such that c can execute securely.

$$\begin{aligned}
wpif_{\mathcal{RG}}(c, Q) \mathrel{\widehat{=}} &PO(c) \wedge guar(\mathcal{G}, c) \wedge wp_Q(c, Q) \wedge\\
&\quad stable_{\mathcal{R}}(PO(c) \wedge guar(\mathcal{G}, c) \wedge wp_Q(c, Q))
\end{aligned}$$

The transformer produces the usual proof obligation and transformation of the post-condition $wp_Q(c, Q)$, checks that the guarantee is not violated, and checks whether $PO(c)$, $guar(\mathcal{G}, c)$, and $wp_Q(c, Q)$ are stable under $\mathcal{R}$.

The predicates $guar(\mathcal{G}, c)$ and $stable_{\mathcal{R}}(wp_Q(c, Q))$ are specific to the type of instruction c which results in the instruction specific definitions for $wpif_{\mathcal{RG}}$ as follows.

1) Assignments:

$$\begin{aligned}
&wpif_{\mathcal{RG}}(x := e, Q) \mathrel{\widehat{=}}\\
&\quad PO(x := e) \wedge guar(\mathcal{G}, x := e) \wedge Q[x \leftarrow e, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(e)] \wedge\\
&\quad stable_{\mathcal{R}}(PO(x := e) \wedge\\
&\qquad\qquad\qquad guar(\mathcal{G}, x := e) \wedge Q[x \leftarrow e, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(e)])
\end{aligned}$$

[2]Note that it suffices to state $\Gamma_v \Rightarrow \Gamma'_v$ instead of $\Gamma_v = \Gamma'_v$ in Axioms R1, R2, G1 and G2, as has been done in the Isabelle/HOL encoding.

2) Conditionals: If c is a conditional statement $\mathsf{ite}(b, c_1, c_2)$, assignments only occur at the lower level of the branches c_1 and c_2, and the guarantee check on the top level simplifies to *true*. Further guarantee checks on the instructions within c_1 and c_2 are covered by the nested calls to the transformer. Note that in the presentation we separate the stability check on the proof obligation and the stability check on the remainder.

$$\begin{aligned}
&\mathit{wpif}_{\mathcal{RG}}(\mathsf{ite}(b, c_1, c_2), Q)) \mathrel{\widehat{=}} \\
&\quad \Gamma_{\mathrm{E}}(b) \wedge \mathit{stable}_{\mathcal{R}}(\Gamma_{\mathrm{E}}(b)) \wedge \\
&\quad (b \Rightarrow \mathit{wpif}_{\mathcal{RG}}(c_1, Q)) \wedge (\neg b \Rightarrow \mathit{wpif}_{\mathcal{RG}}(c_2, Q)) \\
&\quad \mathit{stable}_{\mathcal{R}}((b \Rightarrow \mathit{wpif}_{\mathcal{RG}}(c_1, Q)) \wedge (\neg b \Rightarrow \mathit{wpif}_{\mathcal{RG}}(c_2, Q)))
\end{aligned}$$

3) Loops: When c is a loop, similarly to conditional statements, the guarantee does not need to be checked at the top level of this statement. Only the body of the loop can update variables and the check of the guarantee is deferred to the nested call of $\mathit{wpif}_{\mathcal{RG}}$, hence $\mathit{guar}(\mathcal{G}, \mathsf{while}(b, c)) = \mathit{true}$.

$$\begin{aligned}
&\mathit{wpif}_{\mathcal{RG}}(\mathsf{while}(b, c), Q)) \mathrel{\widehat{=}} \mathit{Inv} \wedge \mathit{stable}_{\mathcal{R}}(\mathit{Inv}) \wedge \\
&\quad (\forall \sigma.\ (\mathit{Inv} \Rightarrow \Gamma_{\mathrm{E}}(b)) \wedge \\
&\qquad ((\mathit{Inv} \wedge b) \Rightarrow \mathit{wpif}_{\mathcal{RG}}(c, \mathit{Inv})) \wedge \ ((\mathit{Inv} \wedge \neg b) \Rightarrow Q))
\end{aligned}$$

Note that for any predicate p, $\forall \sigma. p$ amounts to either *true* or *false*, and since $\mathit{stable}_{\mathcal{R}}(\mathit{true}) = \mathit{stable}_{\mathcal{R}}(\mathit{false}) = \mathit{true}$, it suffices to check stability of *Inv* only.

4) Instructions with CAS *expressions:* As can be seen for the transformer *wpif*, the weakest precondition of a CAS assignment resembles that of a conditional. The difference when it comes to interference with the environment, however, lies in the atomicity of the test and the (potential) update of the CAS variable. Consequently, stability does not need to be checked between test and update. That is, stability is only checked on the instruction(s) which follows the (potential) update of the CAS (using $\mathit{wpif}_{\mathcal{RG}}$), and on the weakest precondition of the instruction as a whole. It is not checked on the update to the CAS variable (to which *wpif* is applied).

Also, only in the case of a positive test in CAS do we need to check whether the subsequent update to the CAS variable satisfies the guarantee condition. These principles result in the following definitions for CAS instructions.

$$\begin{aligned}
&\mathit{wpif}_{\mathcal{RG}}(y := \mathsf{CAS}(x, e_1, e_2)) \mathrel{\widehat{=}} \\
&\quad \Gamma_{\mathrm{E}}(x = e_1) \wedge \mathit{stable}_{\mathcal{R}}(\Gamma_{\mathrm{E}}(x = e_1)) \wedge \\
&\quad (x = e_1 \Rightarrow \mathit{guar}(\mathcal{G}, x := e_2) \wedge \\
&\qquad\qquad \mathit{wpif}(x := e_2, \mathit{wpif}_{\mathcal{RG}}(y := \mathit{true}, Q))) \wedge \\
&\quad (x \neq e_1 \Rightarrow \mathit{wpif}_{\mathcal{RG}}(y := \mathit{false}, Q)) \wedge \\
&\quad \mathit{stable}_{\mathcal{R}}((x = e_1 \Rightarrow \mathit{guar}(\mathcal{G}, x := e_2) \wedge \\
&\qquad\qquad \mathit{wpif}(x := e_2, \mathit{wpif}_{\mathcal{RG}}(y := \mathit{true}, Q))) \wedge \\
&\qquad (x \neq e_1 \Rightarrow \mathit{wpif}_{\mathcal{RG}}(y := \mathit{false}, Q)))
\end{aligned}$$

$$\begin{aligned}
&\mathit{wpif}_{\mathcal{RG}}((\mathsf{if}\ (\mathsf{CAS}(x, e_1, e_2))\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, Q)) \mathrel{\widehat{=}} \\
&\quad \Gamma_{\mathrm{E}}(x = e_1) \wedge \mathit{stable}_{\mathcal{R}}(\Gamma_{\mathrm{E}}(x = e_1)) \wedge \\
&\quad (x = e_1 \Rightarrow \mathit{guar}(\mathcal{G}, x := e_2) \wedge \\
&\qquad\qquad \mathit{wpif}(x := e_2, \mathit{wpif}_{\mathcal{RG}}(c_1, Q))) \wedge \\
&\quad (x \neq e_1 \Rightarrow \mathit{wpif}_{\mathcal{RG}}(c_2, Q)) \wedge \\
&\quad \mathit{stable}_{\mathcal{R}}((x = e_1 \Rightarrow \mathit{guar}(\mathcal{G}, x := e_2) \wedge \\
&\qquad\qquad \mathit{wpif}(x := e_2, \mathit{wpif}_{\mathcal{RG}}(c_1, Q))) \wedge \\
&\qquad (x \neq e_1 \Rightarrow \mathit{wpif}_{\mathcal{RG}}(c_2, Q)))
\end{aligned}$$

$$\begin{aligned}
&\mathit{wpif}_{\mathcal{RG}}(\mathsf{while}\ (\mathsf{CAS}(x, e_1, e_2))\ \mathsf{do}\ c, Q) \mathrel{\widehat{=}} \\
&\quad \mathit{Inv} \wedge \mathit{stable}_{\mathcal{R}}(\mathit{Inv}) \wedge \\
&\quad (\forall \sigma.(\mathit{Inv} \Rightarrow \Gamma_{\mathrm{E}}(x = e_1)) \wedge \\
&\qquad (\mathit{Inv} \wedge x = e_1 \Rightarrow \mathit{guar}(\mathcal{G}, x := e_2) \wedge \\
&\qquad\qquad \mathit{wpif}(x := e_2, \mathit{wpif}_{\mathcal{RG}}(c, \mathit{Inv}))) \wedge \\
&\qquad (\mathit{Inv} \wedge x \neq e_1 \Rightarrow Q))
\end{aligned}$$

5) Sequential composition: For sequential composition the predicate transformer is defined as

$$\mathit{wpif}_{\mathcal{RG}}(c_1;\ c_2, Q) \mathrel{\widehat{=}} \mathit{wpif}_{\mathcal{RG}}(c_1, \mathit{wpif}_{\mathcal{RG}}(c_2, Q))$$

Note that the predicate Q in the above stems from previously generated proof obligations which have been checked for stability at the time of their generation.

D. Example under $\mathit{wpif}_{\mathcal{RG}}$

As stated earlier the security classifications of the two global variables of our example are as follows.

$$\mathcal{L}(x) = (z \neq 1) \text{ and } \mathcal{L}(z) = \mathit{true}$$

To show the working of the predicate transformer $\mathit{wpif}_{\mathcal{RG}}$ we now develop the rely and guarantee conditions for each operation. For the operation **sync_write** of Figure 1, it can be observed that it is the only operation that writes a *high* value into the buffer variable x (the environment can either overwrite a *high* value or maintain a *low* level). Additionally, the environment of **sync_write** can only alter the control variable z if it is not equal to 1. In return **sync_write** can guarantee that no variable is changed when the synchronised reader has the lock ($z = 2$). In addition to conditions R1-R3 and G1-G3, we have the following rely and guarantee conditions for **sync_write**.

$$\begin{aligned}
\mathcal{R}_{\mathit{sync\_write}} &= (\Gamma_x \Rightarrow \Gamma'_x) \wedge (z = 1 \Rightarrow z = z') \\
\mathcal{G}_{\mathit{sync\_write}} &= (z = 2 \Rightarrow z = z' \wedge x = x')
\end{aligned}$$

The operation **sync_read** relies on the fact that when it holds the lock ($z = 2$), z remains unchanged and if x changes it will only change to a *low* value (which might happen through un-synchronised writes which only write *low* values). The operation can guarantee never to change x and to not change z when it is held by the synchronised writer ($z = 1$). Hence, in addition to conditions R1-R3 and G1-G3, we have the following rely and guarantee conditions for **sync_read**.

$$\begin{aligned}
\mathcal{R}_{\mathit{sync\_read}} &= (z = 2 \Rightarrow z = z' \wedge \Gamma_x \Rightarrow \Gamma'_x) \\
\mathcal{G}_{\mathit{sync\_read}} &= (x = x') \wedge (z = 1 \Rightarrow z = z')
\end{aligned}$$

Apart from conditions R1-R3, the rely conditions for the un-synchronised operations are simply *true* and hence, since the security classification of x is dependent on $z \neq 1$, the writer will only be allowed to write a *low* value into x (which is checked through the proof obligations generated by our logic). The guarantee conditions add to G1-G3 the conditions that z and x are unchanged, in the case of **read**, and that z is unchanged and x is not assigned a *high* value, in the case of **write**.

$$\begin{aligned}
&\mathcal{R}_{\mathit{read}} = \mathcal{R}_{\mathit{write}} = \mathit{true} \\
&\mathcal{G}_{\mathit{read}} = (z = z' \wedge x = x') \\
&\mathcal{G}_{\mathit{write}} = (z = z' \wedge (\Gamma_x \Rightarrow \Gamma'_x))
\end{aligned}$$

It is easy to prove that these conditions are compatible, i.e., for all operation i,j with $i \neq j$, $\mathcal{G}_i \Rightarrow \bigwedge_j \mathcal{R}_j$.

We show the workings of the transformer $wpif_{\mathcal{RG}}$ on the operation **sync_write**. As in the previous section, the analysis proceeds backwards through the code, starting with the post-condition *true*. The results are used in the (backwards) reasoning shown in Figure 3. In the following we develop the guarantee checks and stability conditions for the assignments in the operation. Note that for the sake of readability, we abbreviate with PO_n (for $n \in 1..3$) the proof obligation generated at each instruction.

For the last instruction $z := 0$ of the operation **sync_write**, the following guarantee condition is generated.

$$\begin{aligned}
&guar(\mathcal{G}, z := 0)\\
&= wp'(z := 0, z_0 = 2 \Rightarrow z = z_0 \wedge x = x_0)\\
&\qquad\qquad [z_0 \leftarrow z, x_0 \leftarrow x, \Gamma_{z_0} \leftarrow \Gamma_z, \Gamma_{x_0} \leftarrow \Gamma_x]\\
&= (z = 2 \Rightarrow 0 = z \wedge x = x)\\
&= (z \neq 2)
\end{aligned}$$

For the same instruction, the stability condition reduces to the stability of the proof obligation and the guarantee condition (as shown above). $PO(z := 0)$ reduces to $secureUpd(z := 0)$ and since $\mathcal{L}(x)[z \leftarrow 0] = true$ this reduces further to $\Gamma_x \vee \mathcal{L}(x)$ which equals $(\Gamma_x \vee z \neq 1)$. Since $\mathcal{R}_{sync\_write}$ provides stability on a *low* value of x, and stability on z when $z = 1$ we can derive the given result.

$$\begin{aligned}
&stable_{\mathcal{R}}((\Gamma_x \vee z \neq 1) \wedge z \neq 2) =\\
&\quad = ((\Gamma_x \vee z \neq 1) \wedge z \neq 2) \wedge \mathcal{R}_{sync\_write} \Rightarrow\\
&\qquad ((\Gamma'_x \vee z' \neq 1) \wedge z' \neq 2)\\
&\quad = (z = 1)
\end{aligned}$$

For the next instruction $x := 0$, the following guarantee condition is generated.

$$\begin{aligned}
&guar(\mathcal{G}, x := 0)\\
&= wp'(x := 0, z_0 = 2 \Rightarrow z = z_0 \wedge x = x_0)\\
&\qquad\qquad [z_0 \leftarrow z, x_0 \leftarrow x, \Gamma_{z_0} \leftarrow \Gamma_z, \Gamma_{x_0} \leftarrow \Gamma_x]\\
&= (z = 2 \Rightarrow z = z \wedge 0 = x)\\
&= (z = 2 \Rightarrow x = 0)
\end{aligned}$$

With $PO_2 = \mathcal{L}(x) \Rightarrow \Gamma_{\mathrm{E}}(0)$ which simplifies to *true* and the above, the stability condition is computed as follows.

$$\begin{aligned}
&stable_{\mathcal{R}}(PO_2 \wedge guar(\mathcal{G}, z := 0) \wedge\\
&\qquad\qquad (\Gamma_x \wedge (z = 1))[x \leftarrow 0, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(0)])\\
&\quad = stable_{\mathcal{R}}((z = 2 \Rightarrow x = 0) \wedge z = 1)\\
&\quad = stable_{\mathcal{R}}(z = 1)\\
&\quad = (z = 1)
\end{aligned}$$

For the update $x := secret$, the proof obligation $PO_3 = \mathcal{L}(x) \Rightarrow \Gamma_{\mathrm{E}}(secret)$ which equals $(z = 1)$ and $guar(\mathcal{G}, x := secret) = (z = 2 \Rightarrow x = secret)$. Hence the stability of the proof obligation takes the following shape.

$$\begin{aligned}
&stable_{\mathcal{R}}(z = 1 \wedge (z = 2 \Rightarrow x = secret) \wedge\\
&\qquad\qquad (z = 1)[x \leftarrow secret, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(secret)])\\
&\quad = stable_{\mathcal{R}}(z = 1)\\
&\quad = (z = 1)
\end{aligned}$$

The reasoning over the nested loops at the beginning of the code is equivalent to that in Section IV-F, given that the post-condition is $(z = 1)$ and both loop invariants can be set to

```
{ $true$ }
  while($\neg$CAS(z, 0, 1))){
    { $true$ }
      while(z $\neq$ 0) { };
    { $true$ }
  }
{ $z = 1$ }
{ $z = 1 \wedge guar(\mathcal{G}, x := secret) \wedge$
  $(z = 1)[x \leftarrow secret, \Gamma_x \leftarrow \Gamma_{secret}] \wedge$
  $stable_{\mathcal{R}}(PO_3 \wedge guar(\mathcal{G}, x := secret) \wedge$
        $(z = 1)[x \leftarrow secret, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(secret)])$ }
  x := secret;
{ $z = 1$ }
{ $true \wedge$
  $guar(\mathcal{G}, x := 0) \wedge (\Gamma_x \wedge z = 1)[x \leftarrow 0, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(0)]$
  $stable_{\mathcal{R}}(PO_2 \wedge guar(\mathcal{G}, z := 0) \wedge$
        $(\Gamma_x \wedge z = 1)[x \leftarrow 0, \Gamma_x \leftarrow \Gamma_{\mathrm{E}}(0)])$ }
  x := 0;
{ $\Gamma_x \wedge z = 1$ }
{ $(\Gamma_x \vee z \neq 1) \wedge$
  $guar(\mathcal{G}, z := 0) \wedge true[z \leftarrow 0, \Gamma_z \leftarrow \Gamma_{\mathrm{E}}(0)] \wedge$
  $stable_{\mathcal{R}}(PO_1 \wedge guar(\mathcal{G}, z := 0) \wedge$
        $true[z \leftarrow 0, \Gamma_z \leftarrow \Gamma_{\mathrm{E}}(0)])$ }
  z := 0;
{ $true$ }
```

Fig. 3. Transformer $wpif_{\mathcal{RG}}$ applied to **sync_write**

true (hence stability over both is trivially true). The guarantee condition over the CAS update evaluates to $z \neq 2$, as shown in the following.

$$\begin{aligned}
&guar(\mathcal{G}, z := 1)\\
&= wp'(z := 1, z_0 = 2 \Rightarrow z = z_0 \wedge x = x_0)\\
&\qquad\qquad [z_0 \leftarrow z, x_0 \leftarrow x, \Gamma_{z_0} \leftarrow \Gamma_z, \Gamma_{x_0} \leftarrow \Gamma_x]\\
&= (z = 2 \Rightarrow 1 = z \wedge x = x)\\
&= (z \neq 2)
\end{aligned}$$

Hence the implication $(Inv_1 \wedge z = 0 \Rightarrow guar(\mathcal{G}, z := 1))$ evaluates to *true*.

The result shows that there are no proof obligations on the initial state. The stability of this proof obligation has been assured alongside the backwards reasoning, as prescribed by the transformer $wpif_{\mathcal{RG}}$.

VI. Isabelle encoding and proof support

Our information flow analysis is sound with respect to a definition of *value-dependent non-interference* for shared memory concurrent programs based on [31]. This property establishes a strong low bisimulation that guarantees the preservation of a *low*-equivalence relation between two versions of the memory throughout execution. Given a security policy $\mathcal{L}$, we define our *low*-equivalence relation $\mathcal{S}$ such that the two related memories must agree on the value of a variable if its security classification is *low* in either.

$$\begin{aligned}
&(m_1, m_2) \in \mathcal{S} \equiv\\
&\qquad \forall x \cdot (m_1 \in \mathcal{L}\ x \vee m_2 \in \mathcal{L}\ x) \Rightarrow m_1\ x = m_2\ x
\end{aligned}$$

Given such a property, it is guaranteed that the values of variables classified as *low* are not influenced by the values of those classified as *high*. Hence, an attacker who can observe the former cannot deduce anything about the latter.

To demonstrate soundness of the analysis in Isabelle/HOL [33], we prove that the desired bisimulation must hold for a program executing with initial conditions determined by $wpif_{\mathcal{RG}}$. This has been achieved over the programming language introduced in this paper with an extension to support simple array operations.

The encoding has also been used to automate our weakest precondition analysis and applied to a selection of examples of concurrent, non-blocking algorithms adapted to have security properties. As well as the spinlock-based reader/writer mechanism of Figure 1, we have used our encoding to automatically verify non-interference for a reader/writer mechanism based on the Linux seqlock algorithm [9] in which the writer thread is never blocked by a reader thread, a version of the Treiber stack [39] in which classified stack elements cannot be popped by unauthorised threads, and a version of the Chase-Lev work-stealing deque [10] in which classified tasks on the deque can only be accessed by the worker thread; all other threads are limited to stealing unclassified tasks. All examples were verified within 10 seconds on a standard laptop.

The Isabelle/HOL encodings of the logic, soundness proof and examples can be found under https://bitbucket.org/wmmif/wp-rg-if/src/csf2021/. Code and descriptions for the examples are found in the Appendix.

A. Relational Rely/Guarantee

To minimise the verification burden, the soundness proof relies on an existing relational rely/guarantee logic [11] to establish its desired bisimulation properties. This relational logic enables the expression of predicates over a pair of memories and enforces equivalent branching and termination behaviours between their executions.

Consequently, we encode our definition of *low*-equivalence as an invariant within the rely/guarantee logic, preserved by both the rely and guarantee specifications. We use the shorthand *pres P* to represent the preservation of a state across a state transition.

Lemma 1: Given a relational rely/guarantee judgement $(\mathcal{R} \cap \mathit{pres}\ \mathcal{S}), (\mathcal{G} \cap \mathit{pres}\ \mathcal{S}) \vdash (P \cap \mathcal{S})\ \{c\}\ Q$ that preserves *low*-equivalence, and given two *low*-equivalent memories, m_1 and m_2, that satisfy P, and given the pair (c, m_1) may partially evaluate to (c_1, m_1') via a trace of instructions t, then there must exist a partially evaluated state (c_2, m_2') that (c, m_2) may reach via the trace t, such m_1' and m_2' are *low*-equivalent.

$$\begin{aligned} &(\mathcal{R} \cap \mathit{pres}\ \mathcal{S}), (\mathcal{G} \cap \mathit{pres}\ \mathcal{S}) \vdash (P \cap \mathcal{S})\ \{c\}\ Q \Rightarrow \\ &\quad (m_1, m_2) \in P \cap \mathcal{S} \wedge (c, m_1) \longrightarrow_t (c_1, m_1') \Rightarrow \\ &\qquad \exists\, c_2,\ m_2' \cdot (c, m_2) \longrightarrow_t (c_2, m_2') \wedge (m_1', m_2') \in \mathcal{S} \end{aligned}$$

where $(c, m) \longrightarrow_t (c', m')$ represents the execution of program c and memory m to a new state via a trace of instructions t.

As the language is deterministic, Lemma 1 implies that all memories m_2' that satisfy $(c, m_2) \longrightarrow_t (c_2, m_2')$ must be *low*-equivalent with m_1'.

Note that both executions in Lemma 1 must evaluate the same trace of instructions t. As the program c consists of parallel components with non-deterministic scheduling, this constrains the scheduling behaviour to select the same instructions from the available threads. This is slightly less restrictive than limiting the scheduler to selecting the same threads, as different threads may be chosen as long as their instructions are equivalent.

B. Assertion Language for Proof Obligations

As the information flow analysis does not reason at the level of two *low*-equivalent memories, the Isabelle/HOL encoding introduces a deeply embedded assertion language over a single memory m (mapping variables to an abstract value type) and a type context Γ (mapping variables to their security level). A deep embedding approach was chosen due to the ease with which predicates may be transformed and optimised, as seen in prior work [42]. For instance, it is trivial under such an approach to determine the variables referenced in an assertion, as required by *secureUpd*. Moreover, an operation to convert such assertions to *low*-equivalent memories is included, enabling reasoning within the relational rely/guarantee logic.

Lemma 2: Evaluating a converted assertion, $[A]$, on the memory pair m_1 and m_2 is equivalent to evaluating the assertion A directly on both individual memories with a Γ computed based on their difference (denoted $m_1 \triangle m_2$ below).

$$\begin{aligned} &(m_1, m_2) \in [A] = \\ &\quad \mathit{eval}\ (m_1, m_1 \triangle m_2)\ A \wedge \mathit{eval}\ (m_2, m_1 \triangle m_2)\ A \end{aligned}$$

where $(m_1 \triangle m_2)\ x = (m_1\ x = m_2\ x)$, overapproximating low-equivalent variables, and *eval* defines the evaluation of a deeply embedded assertion.

C. Wellformedness

It is necessary to establish a series of wellformedness properties on the logic's specifications and invariants to demonstrate soundness. For example, the relation $\mathcal{R}$ must be transitive and reflexive, ensuring it models any number of environment steps. Additionally, the relation $\mathcal{G}$ must be reflexive to cover the cases where the thread does not modify the state. Moreover, the security policy $\mathcal{L}$ is constrained to prevent a variable from influencing its own classification.

Wellformedness properties also apply to the logic's assertions A (i.e., the generated proof obligations and their modifications) as they are updated throughout the analysis. For example, the assertions must always be stable under the rely condition $\mathcal{R}$. These properties are all bundled into a definition *wellformed* $\mathcal{R}\ \mathcal{G}\ \mathcal{L}\ A$.

D. Sequential Judgements

To facilitate proof support and handle much of the implicit complexity of the relational rely/guarantee logic, the encoding defines a set of rules for judgements over a single thread. These judgements preserve wellformedness and abstract over the preservation of the security policy $\mathcal{L}$. The resulting rules closely reflect those of a traditional Hoare logic.

$$
\begin{aligned}
&\mathcal{R},\mathcal{G} \vdash_{seq} A \ \{c\}\ A' \equiv \\
&\quad \mathit{wellformed}\ \mathcal{R}\ \mathcal{G}\ \mathcal{L}\ A' \Rightarrow \\
&\qquad (\mathcal{R} \cap \mathit{pres}\ \mathcal{S}), (\mathcal{G} \cap \mathit{pres}\ \mathcal{S}) \vdash ([A] \cap \mathcal{S})\ \{c\}\ [A'] \wedge \\
&\qquad\quad \mathit{wellformed}\ \mathcal{R}\ \mathcal{G}\ \mathcal{L}\ A
\end{aligned}
$$

Lemma 3: Given a rely/guarantee specification, $\mathcal{R}$ and $\mathcal{G}$, a program c and a postcondition A', $\mathit{wpif}_{\mathcal{RG}}(c, A')$ will compute a precondition such that $\mathcal{R},\mathcal{G} \vdash_{seq} \mathit{wpif}_{\mathcal{RG}}(c, A')\ \{c\}\ A'$ holds.

Lemma 3 can be established via induction over c and application of appropriate rules from the relational rely/guarantee logic where necessary. Therefore, given $\mathit{wellformed}\ \mathcal{R}\ \mathcal{G}\ \mathcal{L}\ A'$ and initial conditions that imply $\mathit{wpif}_{\mathcal{RG}}(c, A')$, it is possible to establish the desired security property via Lemma 1. Moreover, the rely/guarantee framework enables the parallel composition of these properties, given compatible $\mathcal{R}$ and $\mathcal{G}$ specifications.

E. Proof Support

Various existing program logics have been mechanised in Isabelle/HOL with some degree of automation via verification condition generation using weakest-precondition transformations [16], [32]. We employ these techniques to enable similar levels of automation when attempting to establish information flow properties of concurrent objects within Isabelle/HOL. While its scalability is limited, this approach allows for experimentation with the analysis without significant reimplementation. Additionally, due to the application of the verified definitions, such an approach can provide much stronger guarantees when compared with alternative implementations.

To achieve the automation when applying the verified definitions, the predicate transformation of the $\mathit{wpif}_{\mathcal{RG}}$ function is rephrased as logic rules, with multiple variations for special cases of some program constructs. These rules are then applied at the tactic level, using pattern matching and backtracking to determine the best rule for a particular situation using Isabelle/HOL's proof tactic language Eisbach [25].

This approach is preferable to the computation of $\mathit{wpif}_{\mathcal{RG}}$ function directly, as it allows for the simplification and elimination of proof obligations early in the analysis. For example, it is possible to eliminate the $\mathit{stable}_{\mathcal{R}}$ proof obligation on an instruction that does not write or read global variables. This considerably reduces the growth of the weakest-precondition, simplifying later reasoning.

After automatic application of these rules to a program, the remaining subgoals consist of predicate entailments phrased in the deeply embedded assertion language. Reasoning in this form is difficult due to the lack of theorems over the assertion language. To alleviate this, operations over the assertion language are made executable, enabling normalization via Isabelle/HOL's code generation capabilities. The resulting normalized predicate implications are fully expanded to only use Isabelle/HOL's built-in operators over an abstract memory and type state. These are then fed into simplification tactics to be decomposed and clarified, with most subgoals being immediately discharged.

Any remaining subgoals typically require additional lemmas and more complex tactics to be successfully discharged. The mechanised examples focus on operations over natural numbers, for which there is a large amount of existing support in Isabelle/HOL. Consequently, all non-trivial subgoals can be resolved via application of Isabelle/HOL's sledgehammer tactic to identify the required lemmas and suitable tactics.

F. Arrays

The encoding includes basic support for arrays, in order to verify additional concurrent algorithms and explore the limitations of the proof support.

Arrays are modelled as a collection of individual variables, with specialised operations to enable writing and reading. The store operation (i.e., the assignment to an array at a particular index) takes a list of variables, representing the array; an index operation, identifying which variable to access; and an expression to evaluate and store. To compute the weakest-precondition of such a store all constituent variables of the array are substituted with an expression representing the possibility of modification. For instance, a store of the form $A[i] := e$, where A is the array, i is the modified index and e is the written expression, would substitute A_n, the nth variable of the array A, with $\mathsf{if}\ i = n\ \mathsf{then}\ e\ \mathsf{else}\ A_n$.

In a similar fashion, the proof obligations for such a store consist of the proof obligations for each variable that could be assigned, guarded by a test on the outcome of the index expression.

The load operation (i.e., the read of an array at a particular index) takes a destination variable, to write the value to; a list of variables, representing the array; and an index operation. To compute the weakest-precondition for a load, an expression is generated to represent the possible outcomes of the load given each potential index, which the destination variable is then substituted with.

Similar to the store, the proof obligations for a load consist of the proof obligations for each variable that could be read, guarded by a test on the outcome of the index expression.

Additionally, the index expression for array operations is constrained to be *low*-equivalent, similar to branching expressions. This is required as an invalid array access will halt the program, under our semantics. As this may be observable to an attacker, a difference in halting behaviour can leak information if the index is influenced by a secret.

G. Predicate Size

All mechanised examples can be verified within 10 seconds on a standard laptop. However, this execution time is highly dependent on the size of the predicate, which is in turn significantly influenced by the $\mathcal{R}$ specification via the $\mathit{stable}_{\mathcal{R}}$ proof obligations for each instruction. Hence, the examples are constructed with the smallest possible $\mathcal{R}$.

The use of a compact $\mathcal{R}$ is employed in the Treiber stack example. Rather than providing a detailed rely/guarantee specification that one would require to demonstrate *correctness* of the algorithm, a simplified specification is used with sufficient information to reason about *information flow*. This results in a significantly quicker verification.

VII. Treatment of general security lattices

To generalise the treatment to non-binary security lattices we propose the following formalisation. Assume a complete lattice of security values $\mathcal{S}ec$, with possibly more than two values, with partial order $\sqsubseteq$, a join ($\sqcup$) and a meet operator ($\sqcap$) such that $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$ and $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$.

Within this lattice of security values, the user needs to specify a threshold defining which security levels are considered observable by a potential attacker, $sec_{attack} \in \mathcal{S}ec$, such that all $sec \sqsubseteq sec_{attack}$ are considered publicly accessible.

In order to specify value-dependent classifications that range over more than two security values, we introduce *conditional security expressions*, CondSec. These expressions evaluate to a value in $\mathcal{S}ec$ depending on the current state. For example, assume $\mathcal{S}ec = \{secret, confidential, public\}$, then the security classification for a multi-purpose buffer can be defined as the following conditional security expression (assuming a syntax of ternary *if-then-else* expressions).

$$\begin{array}{l} \mathcal{L}(buffer) = \\ \quad \textsf{if } userId = customer \textsf{ then } public \\ \qquad \textsf{else if } userId = finance \textsf{ then } confidential \\ \qquad\qquad \textsf{else } secret \end{array}$$

A wellformedness condition on *CondSec* requires that each expression is complete in the sense that we can decide in every state the security value of the expression (i.e., the conditions in a conditional security expression must form a partition of the state space).

CondSec subsumes the security lattice $\mathcal{S}ec$, e.g., the $\mathcal{S}ec$ value *public* could be represented by the expression (if *true* then *public*), and therefore its partial order $\sqsubseteq$ and meet and join operators, $\sqcap$ and $\sqcup$. The semantics is defined via an evaluation function that takes the current state $\sigma \in \Sigma$ into account.

$$eval_{sec} : CondSec \times \Sigma \to \mathcal{S}ec$$

The wellformedness condition on conditional security expressions assures that $eval_{sec}$ is a total function.

Substitution over a conditional security expression, $csec[x \leftarrow e]$, substitutes free variables in the conditions with the corresponding expression, e.g.,

$$\begin{array}{l} \mathcal{L}(buffer)[userId \leftarrow finance] = \\ \quad \textsf{if } finance = customer \textsf{ then } public \\ \qquad \textsf{else if } finance = finance \textsf{ then } confidential \\ \qquad\qquad \textsf{else } secret \end{array}$$

which simplifies to *confidential*.

Building on this we define the security classification of variables as a mapping from variables to conditional security expressions $\mathcal{L} : Var \to CondSec$. The security level maps each variable to a security value, i.e., $\gamma : Var \to \mathcal{S}ec$. The security level of expressions maps an expression to a conditional security expression, i.e., $\Gamma_{\mathrm{E}} : Exp \to CondSec$, combining security classification and security level of the involved variables.

$$\Gamma_{\mathrm{E}}(e) \mathrel{\widehat{=}} \textstyle\bigsqcap_{v \in vars(e)} (\Gamma_v \sqcup \mathcal{L}(v)).$$

The proof obligation to ensure information flow security can then be phrased as follows.

$$\begin{array}{l} PO(x := e) \mathrel{\widehat{=}} (x \in Global \Rightarrow \mathcal{L}(x) \sqsupseteq \Gamma_{\mathrm{E}}(e)) \wedge \\ \quad (x \in \mathcal{C}trl \Rightarrow \forall\, y \in ctrled(x).\, \mathcal{L}(y)[x \leftarrow e] \sqsupseteq (\Gamma_y \sqcup \mathcal{L}(y))) \end{array}$$

For conditional and loop instructions, we need to ensure that the security level of the guard lies below the accessible security level of a potential attacker, sec_{attack}. Hence we define

$$PO(\textsf{ite}(b, c_1, c_2)) \mathrel{\widehat{=}} \Gamma_{\mathrm{E}}(b) \sqsubseteq sec_{attack}$$
$$PO(\textsf{while}(b, c)) \mathrel{\widehat{=}} Inv \wedge (\forall\, \sigma.\, Inv \Rightarrow (\Gamma_{\mathrm{E}}(b) \sqsubseteq sec_{attack}))$$

The Isabelle encoding of the corresponding generalised information flow logic and soundness proof can be found in https://bitbucket.org/wmmif/wp-rg-if/src/lattice.

VIII. Related work

A number of approaches to verification of information flow in shared-variable concurrent programs have been developed over the last decade. These approaches fall into two camps: those based on rely/guarantee reasoning, and those based on separation logic [34].

The earliest approach using rely/guarantee reasoning is that of Mantel et al. [24]. This work associates *modes* with each variable referenced by a thread. The modes are either assumptions or guarantees on read and write access to the variable, e.g., an assumption that no other thread writes to the variable, or a guarantee that this thread will not read the variable. This simple form of rely and guarantee conditions is adopted by Murray et al. [31] who additionally added value-dependent security classifications, and Smith et al. [37] in a value-dependent information flow logic for programs running on hardware weak memory models. While trivial to apply, these modes introduce complex proof obligations to establish overall program correctness via compatibility of assumptions and guarantees [4], [23]. Smith et al. [37] avoid this problem by requiring modes to be static over the execution of a program, limiting applicability. In their later work [30], Murray et al. simplify the analysis by associating mode changes with the acquisition and release of locks. This work also associates invariants with the acquisition of locks opening the way for more general rely and guarantee conditions.

As shown by Coughlin and Smith [11], simple rely and guarantee conditions supported by modes are not sufficient for all concurrent programs. Furthermore, the use of locks, as proposed by Murray et al., [30] limits applicability to lock-based programs. Coughlin and Smith [11] provide a value-dependent information flow logic which supports general rely and guarantee condition, and can be used with *non-blocking* programs which do not employ locks. The approach however, based on a forwards strongest postcondition analysis, suffers from complexity, hindering its automation. The only other approach supporting general rely and guarantee conditions that are not associated with locks is that of Schoepe et al. [36]. This approach separates the rely/guarantee analysis from the security analysis to simplify the latter. While this sounds promising, the rely/guarantee analysis is not described in the paper but is assumed to be available using an external tool. As such, its automation is yet to be explored.

The separation logic approaches include those of Karbyshev et al. [20] and Ernst and Murray [14]. In the approach of Karbyshev et al., the thread-local conditions on shared variables are restricted to ownership of the variables, i.e., whether the thread has exclusive access to a variable, or whether its access is shared with other threads. This is similar to the modes of Mantel et al. [24]. Ownership is transferred between threads by sending and receiving signals over channels, similar to the use of locks by Murray et al. [30]. The approach also does not support value-dependent security levels. Hence, it does not support the range of programs supported by our approach.

The focus of Karbyshev et al.'s work, however, is on timing channels due to branching on *high* values. In particular, it considers where such branching forces a particular outcome in a data race, due to either significant delay in a branch or the scheduler becoming tainted by adapting to the workload in a branch. This is complementary to our analysis (where branching on *high* values is not allowed) and it would be interesting to use its ideas to extend the applicability of our approach.

The separation logic approach of Ernst and Murray [14] is more in line with our work. It does not allow branching on *high* variables, and supports value-dependent security classifications and general thread-local conditions on shared variables. Like the earlier work of Murray et al. [30], however, the conditions on shared variables are associated with the acquisition and release of locks. Hence, it only supports lock-based programs. Its focus is on providing information flow support for C, in particular pointers are supported through the points-to notation of separation logic. Leveraging ideas from this work would be interesting to move our approach to a more realistic programming language.

IX. Conclusion

Reasoning about concurrent programs constitutes a complex task as can be seen in the approaches presented in the literature. The need for minimal verification conditions generated by the analysis is only amplified in the concurrent setting, since the effects of the environment's behaviour (the rely condition in a rely/guarantee approach) need to be considered at every single step.

Our solution proposes a backwards-directed analysis, based on weakest precondition computations, in which verification conditions only carry information relevant to the security analysis. Our logic is the first to combine such a backwards analysis with general rely/guarantee reasoning. The encoding of the logic in Isabelle/HOL delivers a high degree of automation for the reasoning process as is showcased by a number of examples. A similar result has not been achieved with other approaches for information flow security for concurrent programs conducted in a forward fashion.

Future work is directed towards the analysis of assembly code (lifted to an intermediate representation) and an extension to the logic to incorporate the effects of hardware weak memory models. This will allow us to reason on executable code directly, thereby incorporating the effects of compiler optimisations into the analysis results. The effects of out-of-order executions (as performed by modern hardware architectures) demands special attention in particular for non-blocking algorithm which avoid the use of locks.

References

[1] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16*, pages 53–70. USENIX Association, 2016.

[2] J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *IJES*, 2(3/4):239–247, 2006.

[3] T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In P. Ning, V. Atluri, V. D. Gligor, and H. Mantel, editors, *Proceedings of the 2007 ACM workshop on Formal methods in security engineering, FMSE 2007*, pages 2–11. ACM, 2007.

[4] A. Askarov, S. Chong, and H. Mantel. Hybrid monitors for concurrent noninterference. In C. Fournet, M. W. Hicks, and L. Viganò, editors, *IEEE 28th Computer Security Foundations Symposium, CSF 2015*, pages 137–151. IEEE Computer Society, 2015.

[5] M. Balliu. *Logics for Information Flow Security:From Specification to Verification*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2014.

[6] M. Balliu and I. Mastroeni. A weakest precondition approach to active attacks analysis. In S. Chong and D. A. Naumann, editors, *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009*, pages 59–71. ACM, 2009.

[7] A. Banerjee, R. Giacobazzi, and I. Mastroeni. What you lose is what you leak: Information leakage in declassification policies. In M. Fiore, editor, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2007*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 47–66. Elsevier, 2007.

[8] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Math. Struct. Comput. Sci.*, 21(6):1207–1252, 2011.

[9] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel - from I/O ports to process management: covers version 2.6 (3. ed.)*. O'Reilly, 2005.

[10] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In P. B. Gibbons and P. G. Spirakis, editors, *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28. ACM, 2005.

[11] N. Coughlin and G. Smith. Rely/guarantee reasoning for noninterference in non-blocking algorithm. In *IEEE Computer Security Foundations Symposium (CSF 2020)*, pages 380–394. IEEE Computer Society, 2020.

[12] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 223–234, New York, NY, USA, 2013. Association for Computing Machinery.

[13] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, Heidelberg, 1990.

[14] G. Ernst and T. Murray. SecCSL: Security concurrent separation logic. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 208–230. Springer, 2019.

[15] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In C. Hankin and D. Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205. ACM, 2001.

[16] D. Greenaway. *Automated proof-producing abstraction of C code*. PhD thesis, University of New South Wales, Sydney, Australia, 2014.

[17] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[18] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[19] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37(1-3):113–138, 2000.

[20] A. Karbyshev, K. Svendsen, A. Askarov, and L. Birkedal. Compositional non-interference for concurrent programs via separation and framing. In L. Bauer and R. Küsters, editors, *Principles of Security and Trust - 7th International Conference, POST 2018, Proceedings*, volume 10804 of *Lecture Notes in Computer Science*, pages 53–78. Springer, 2018.

[21] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
[22] L. Lourenço and L. Caires. Dependent information flow types. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 317–328. ACM, 2015.
[23] H. Mantel, M. Müller-Olm, M. Perner, and A. Wenner. Using dynamic pushdown networks to automate a modular information-flow analysis. In M. Falaschi, editor, *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015. Revised Selected Papers*, volume 9527 of *Lecture Notes in Computer Science*, pages 201–217. Springer, 2015.
[24] 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*, pages 218–232. IEEE Computer Society, 2011.
[25] D. Matichuk, T. C. Murray, and M. Wenzel. Eisbach: A proof method language for Isabelle. *J. Autom. Reason.*, 56(3):261–282, 2016.
[26] A. Mettler, D. A. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010*. The Internet Society, 2010.
[27] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In D. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.
[28] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429, 2013.
[29] T. C. Murray. Short paper: On high-assurance information-flow-secure programming languages. In M. Clarkson and L. Jia, editors, *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2015*, pages 43–48. ACM, 2015.
[30] 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*, pages 16–30. IEEE, 2018.
[31] 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*, pages 417–431. IEEE Computer Society, 2016.
[32] L. P. Nieto. The rely-guarantee method in Isabelle/HOL. In P. Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Proceedings*, volume 2618 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2003.
[33] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
[34] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), Proceedings*, pages 55–74. IEEE Computer Society, 2002.
[35] C. Scheben and P. H. Schmitt. Efficient self-composition for weakest precondition calculi. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods - 19th International Symposium*, volume 8442 of *Lecture Notes in Computer Science*, pages 579–594. Springer, 2014.
[36] D. Schoepe, T. Murray, and A. Sabelfeld. VERONICA: Expressive and precise concurrent information flow security. In *IEEE Computer Security Foundations Symposium (CSF 2020)*, pages 79–94. IEEE Computer Society, 2020.
[37] G. Smith, N. Coughlin, and T. Murray. Value-dependent information-flow security on weak memory models. In M. H. ter Beek, A. McIver, and J. N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019*, volume 11800 of *Lecture Notes in Computer Science*, pages 539–555. Springer, 2019.
[38] D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA):89:1–89:26, 2017.
[39] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Res. Ctr., 1986.
[40] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium*, volume 46, page 2, 2010.

```
initially  $z = 0 \wedge x = 0$

sync_write(dataType secret) :          write(dataType data) :
  z := z+1;                              x := data
  x := secret;
  ... // wait until x is read          read : dataType
  x := 0;                                do
  z := z+1                                 do
                                             r1:= z;
sync_read : dataType                       while (r1 % 2 $\neq$ 0)
  return x                                 r2 := x;
                                         while (z $\neq$ r1)
                                         return r2
```

Fig. 4. seqlock reader/writer mechanism

[41] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.
[42] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2004.

APPENDIX

We briefly describe other examples that have been verified secure using our Isabelle/HOL theories. The proof for these examples is conducted automatically using a single proof tactic.

Seqlock

Seqlock (short for *sequence lock*) [9] is a Linux reader/writer mechanism which allows reading of shared variables without locking the global memory, thus supporting fast write access. The writer thread increments a counter z. It then proceeds to write to the variables, and finally increments z again. The two increments of z ensure that it is odd when the thread is writing to the variables, and even otherwise. Hence, when a reader thread wishes to read the shared variables, it waits in a loop until z is even before reading them. Also, before returning it checks that the value of z has not changed (i.e., another write has not begun). If it has changed, the process starts over.

We adapt this algorithm in Figure 4 to provide a reader/writer mechanism similar to that of Figure 1. In this algorithm, however, the writer thread is never blocked by a reader thread.

To verify this algorithm automatically within Isabelle/HOL, we use the following security classifications, and rely and guarantee conditions (in addition to R1-R3 and G1-G3).

$$\begin{aligned}
&\mathcal{L}(z) = \mathit{true} \text{ and } \mathcal{L}(x) = (z\%2 = 0)\\
&\mathcal{R}_{\mathit{sync\_write}} = \mathcal{G}_{\mathit{public\_read}} = (\Gamma_x \wedge z' = z)\\
&\mathcal{R}_{\mathit{read}} = \mathcal{G}_{\mathit{sync\_write}} = (z' \geq z)\\
&\mathcal{R}_{\mathit{write}} = \mathcal{R}_{\mathit{sync\_read}} = \mathit{true}\\
&\mathcal{G}_{\mathit{write}} = \mathcal{G}_{\mathit{sync\_read}} = (\Gamma_x \wedge z' = z)
\end{aligned}$$

Treiber stack

The Treiber stack [39] was the first proposed non-blocking implementation of a concurrent stack. It models the stack using a linked-list of nodes, each node with a value in variable *val*, and next pointer in variable *next*. A thread doing a push operation assigns the value being pushed onto the stack to the *val* variable of a new node stored in a local variable *n*. It then repeatedly tries to make *n* the head of the stack by setting a local variable *ss* (short for *snap-shot*) to the global variable *head*, setting *n*'s *next* variable to *ss*, and then assigning *head* to *n* provided it is still equal to *ss* (i.e., provided another thread has not in the meantime changed the value of *head*).

A thread doing a pop operation repeatedly sets a local variable *ss* to *head*, returning *empty* if *ss* is null, and otherwise setting another local variable *ssn* to *ss*'s *next* variable and a local variable *v* to *ss*'s *val* variable. Finally, it assigns *ssn* to *head* and returns *v* provided *head* is still equal to *ss*.

In our adaptation of the Treiber stack in Figure 5, we add an additional variable *level* to a node to record whether the value in the node is *high* or *low*. The pop operation shown is that for an untrusted thread. It only allows a value to be popped from the stack when its level is *low*.

To verify this algorithm automatically within Isabelle/HOL, we use the following rely and guarantee conditions (in addition to R1-R3 and G1-G3).

$\mathcal{L}(head) = true$ and for all nodes n in the stack,
$\mathcal{L}(n.level) = true$ and $\mathcal{L}(n.val) = (n.level = Low)$
$\mathcal{R}_{put} = \mathcal{G}_{pop} = true$
For all nodes n in the stack,
$\mathcal{R}_{pop} = \mathcal{G}_{put} = (n.level = Low \Rightarrow n'.level = Low)$

```
initially head = null

put(Value v, Level l) :
  Node n, ss;
  n := new Node;
  n.level := l;
  n.val := v;
  do
    ss := head;
    n.next := ss;
  while (¬ CAS(head, ss, n));

pop : Value
  Node n, ss, ssn;
  Value v;
  Level Level;
  int exit;
  exit := 0;
  v := empty;
  do
    ss := head;
    if (ss = null)
      level := ss.level;
      if (level = Low)
        ssn := ss.next;
        v := ss.val;
        if (CAS(head, ss, ssn))
          exit := 1;
  while (exit = 0);
  return v;
```

Fig. 5. Treiber stack

Chase-Lev deque

The Chase-Lev work-stealing deque (double-ended queue) [10] is implemented as a circular array of size L with a *head* and *tail* pointer. The pointers are non-wrapping, i.e., if a pointer has the value i, it points to the array element at position $i \bmod L$.

The *put* operation straightforwardly adds an element to the end of the deque, incrementing the *tail* pointer. The interesting behaviour is in the way that the *take* and *steal* operations interact when called concurrently. To take the task at position $t = tail - 1$, the worker process decrements *tail* to equal t, thereby publishing its intent to take that task. This publication means subsequent thief processes will not try to steal the task at position t. It then reads *head* into a local variable h and if $h < t$ knows that there is more than one task in the deque and it is safe to take the task at position t, i.e., no thief process can concurrently steal it.

If $t < h$ the worker knows the deque is empty and sets *tail* back to its original value. The final possibility is that $h = t$. In this case, there is one task on the deque and conflict with a thief may arise. To deal with this conflict, both the *take* and *steal* operations employ a CAS instruction. If $h = t$, rather than decrementing *tail* to take the task, the worker uses the CAS to increment *head*. Therefore, if the worker finds $h = t$, it also restores *tail* to its original value. The *steal* operation works similarly. The operation reads the deque's *head* and *tail* into local variables h and t, and if the deque is not empty tries to increment head from h to $h+1$ using a CAS. If it succeeds, the value of *head* has not been changed since read into the local variable h and hence the thief has stolen the task.

Our adaptation of the Chase-Lev deque is shown in Figures 6 and 7. As well as a circular array of tasks, the deque has a circular array of security levels. This array is also of size L and records in position i the security level of the task in position i of the task array. The *put* operation (Figure 6) has two inputs, a task v and security level u, and updates both arrays. The *steal* operation (Figure 7) reads the security level of the task it is trying to acquire and returns *fail* when that task is high. To ensure that the *steal* operation cannot read tasks which are being concurrently overwritten, we use an approach inspired by seqlock. If z changes at any time while *steal* is reading a level and associated task, the read is restarted.

To verify this algorithm automatically within Isabelle/HOL, we use the following rely and guarantee conditions (in addition to R1-R3 and G1-G3).

$\mathcal{L}(tail) = \mathcal{L}(head) = \mathcal{L}(z) = true$ and for all $i < L$,
$\mathcal{L}(tasks[i]) = (levels[i] = Low)$ and $\mathcal{L}(levels[i]) = true$
$\mathcal{R}_{put} = \mathcal{G}_{steal} = (z' = z \wedge \forall\, i < L \cdot tasks'[i] = tasks[i] \wedge$
$levels[i]' = levels[i])$
$\mathcal{G}_{put} = \mathcal{G}_{take} = \mathcal{R}_{steal} = (z' \geq z \wedge (\forall\, i < L \cdot z\%2 = 0 \wedge$
$z' = z \Rightarrow levels'[i] = levels[i]))$
$\mathcal{R}_{take} = true$

initially $z = 0 \wedge head = 0 \wedge tail = 0$

```
put(Task task, Level level) :
  int t;
  t := tail;
  z := z + 1;
  levels[t mod L] := level;
  tasks[t mod L] := task;
  z := z + 1;
  tail := t + 1

take : Task
  int h, t;
  Task task;
  t := tail - 1;
  tail := t;
  h := head;
  if (h ≤ t)
     task := tasks[ t mod L];
     if (h = t)
       if (¬ CAS(head, h, h + 1))
           task := 0
       tail := tail + 1;
  else
     task := empty;
     tail := tail + 1
  return task
```

Fig. 6. Chase-Lev deque: put operation and take operation

```
steal : Task
  int h, t, r;
  Task task;
  Level level;
  h := head;
  t := tail;
  if (h < t)
     do
        do
           r := z;
        while (¬ (r %2=0));
        level := levels[h mod L];
        if (level = Low)
           task := tasks[h mod L];
        else
           task := fail;
     while (z ≠ r);
     if (¬ CAS(head, h, h+1))
           task := fail;
  else
     task := empty;
  return task;
```

Fig. 7. Chase-Lev deque: steal operation