Compositional noninterference on hardware weak memory models

Nicholas Coughlin[a,b], Graeme Smith[b]

[a]*Defence Science and Technology Group, Australia*
[b]*School of Information Technology and Electrical Engineering, The University of Queensland, Australia*

Abstract

Weak memory models are employed by all modern multicore processors to improve their performance. For most code, the effects of such memory models can be largely ignored by the programmer. However, for low-level operating system or library code which can include data races for efficiency, these effects may lead to information leaks which cannot be detected without taking the specific memory model into account. While there have been some efforts to develop information flow logics which can detect such leaks, the existing approaches are either not compositional, hindering scalability, or support only a limited form of compositionality, reducing applicability to programs with only simple interactions between threads.

This paper is the first to provide a compositional logic for enforcing noninterference properties on more complex concurrent algorithms, while taking into account the underlying hardware memory model. It uses rely/guarantee reasoning to establish how security classifications may be modified by concurrent threads, and considers effects of out-of-order execution allowed on modern multicore processors. The results have been formalised and proved sound in the Isabelle/HOL theorem prover, and automated in a prototype symbolic execution tool.

Keywords: Information flow, Concurrency, Weak memory models

1. Introduction

Modern multicore processors utilise *weak memory models* which, for reasons of efficiency, allow instructions to take effect in an order different to that in the program text [1]. For example, the TSO memory model [2], employed by chip manufacturers Intel and AMD, buffers writes to memory, letting hardware control when those writes actually occur. As accesses to global memory are expensive in multicore systems, this leads to significant performance gains. Pipelining of instructions in weaker memory models, such as those of ARM [3, 4] and IBM POWER processors [5], allows both writes and reads of memory to occur out of order with other writes and reads when no dependencies between them exist, i.e., when the instructions do not reference a common memory address.

Such instruction reordering is constrained by basic principles, the key one being that the sequential semantics of each thread in the original code should be preserved [6, 7]. This ensures the effects of weak memory models can largely be ignored by programmers whose code is either not concurrent, or is concurrent but data-race free. However, these effects do need to be considered by programmers writing efficient low-level code for device drivers and data structures. Such code is generally concurrent and often *non-blocking*, i.e., using no, or minimal, locking, and hence inherently not data-race free [8]. It is well known that data races affect the correctness of such code on weak memory models [9]. As shown by Vaughan and Milstein [10] (for the weak memory model TSO), Mantel et al. [11] (for TSO, PSO and IBM-370) and in our own work [12] (for ARMv8, the latest version of ARM), it also leads to security violations which are not detectable using the standard approaches to information flow security.

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

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

write:
$x := low\_in$

secret_write:
$z := z + 1;$
$x := high\_in;$
...
$x := 0;$
$z := z + 1$

read:
do
  do
    $r_1 := z;$
  while ($r_1$ % $2 \neq 0$)
  $r_2 := x;$
while ($z \neq r_1$)
$low\_out := r_2$

secret_read:
$high\_out := x$
```

Figure 1: Readers/writer example based on seqlock

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

This implementation in Figure 1 is secure on TSO where writes, although buffered, are seen by other threads in the order in which they occur. For the weaker memory models of ARM and POWER, the independent writes to z and x can be reordered from the perspective of threads running read. Hence, the assignment of the classified data to x could occur (in memory) before the first assignment to z, and consequently be read into the variable *low_out*. Similarly, the second assignment to z could occur before the assignment of 0 to x, and again the classified data in x could be read into *low_out*. For this reason, it is necessary to take into account the specific memory model (whether it is TSO, or a weaker memory model such as ARM or POWER) when checking information flow; a point previously made by Mantel et al. [11].

Noninterference [13] formally characterises an information flow security property in which an information source of a particular classification cannot influence sinks at lower classifications. As a result, a program exhibiting noninterference would prevent an attacker from determining secret values (such as the value of *high_in*) given access to public outputs (such as *low_out*). It is well accepted that for scalable analysis of concurrent programs, the analysis technique should be *compositional*, i.e., analysis should occur on individual threads in isolation, rather than over the whole program. A number of approaches for proving noninterference have utilised *rely/guarantee reasoning* [14] in which assumptions (or *rely* conditions) about a thread's environment can be used in reasoning about that thread provided all other threads *guarantee* that the assumption holds. For example, Mantel et al. [15] 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.

The approach of Mantel et al. [15] is also used in our work [12] for reasoning about code on the ARMv8 weak memory model. However, while trivial to apply, such simple modes limit the applicability of the approach. For example, the check that z has not changed in the **read** operation of Figure 1 is made by comparing z with r_1. This check is sufficient provided that z always increases (something we know is true from **secret_write** which is the only operation to change z). To reason in isolation about a thread calling **read** requires two things: a *value-dependent* security policy [16, 17] stating that x does not contain classified information when z is even, and an assumption that z only increases. While value-dependent security policies are supported by our earlier work [12], the modes in that work, restricting read and write access to variables, cannot express assumptions such as 'z only increases'.

Our recent work on compositional noninterference [18] allows the use of fully general rely/guarantee conditions, i.e., any predicate can be used to describe allowable state changes by a thread and its environment. It also supports value-dependent security policies. However, it does not support weak memory models. In this paper, we build on that work to support compositional reasoning about concurrent code which takes into account the underlying hardware's weak memory model. We begin in Section 2 with a discussion of related work. In Section 3, an overview of the programming language and principal concepts and definitions required for our information flow logic which is then defined in Section 4. In Section 5, we detail the model of compositional noninterference which we have adopted to prove soundness of our logic in the theorem prover Isabelle/HOL. A high-level description of the soundness proof is given in Section 6. In Section 7, we describe simplifications to the logic to facilitate its automation. These simplifications have been incorporated in a prototype symbolic execution tool. We conclude in Section 8.

2. Related Work

2.1. Compositionality

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 [14], and those based on separation logic [19].

The earliest approach using rely/guarantee reasoning is that of Mantel et al. [15]. 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. [20] who additionally added value-dependent security classifications.While trivial to apply, these modes introduce complex proof obligations to establish overall program correctness via compatibility of assumptions and guarantees [21, 22]. Later work by Murray et al. [23] simplifies 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 we have shown in recent work [18], 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., [23] limits applicability to lock-based programs. Our recent work [18] provides a value-dependent information flow logic which supports general rely and guarantee conditions, and can be used with *non-blocking* programs which do not employ locks. The only other approach supporting general rely and guarantee conditions that are not associated with locks is that of Schoepe et al. [24]. 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. [25], Ernst and Murray [26] and Frumin et al. [27]. 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. [15]. Ownership is transferred between threads by sending and receiving signals over channels, similar to the use of locks by Murray et al. [23]. 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 classsified 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 timing-sensitive analysis (where branching on classified 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 [26] 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. [23], however, the conditions on shared variables are associated with the acquisition and release of locks. Hence, it only supports lock-based programs. An interesting feature of this work is its automation via symbolic execution coupled with SMT solvers. This approach has been used by other projects in enforcing information flow properties [28] [29]. It was also the basis for the symbolic execution tool developed to support our logic.

Frumin et al. [27] combine a separation logic with a type system, the latter facilitating automation. In contrast to the work of Murray et al. [23] (and Ernst and Murray [26]) their approach supports non-blocking algorithms not requiring the use of locks. The approach is comparable to our previous work [18] on which this paper builds. Like that work, the approach of Frumin et al. does not provide support for reasoning about weak memory behaviour.

Zdancewic et al. [30] have explored the application of observational determinism to define compositional type systems for concurrent program security, suitable for notions of refinement. However, this work requires data race freedom to ensure deterministic behaviour and is, therefore, not applicable to non-blocking algorithms.

2.2. Weak memory models

The earliest work on information flow on weak memory models is that of Vaughan and Milstein [10] who investigate information leaks on TSO which cannot be detected using standard information flow techniques. Mantel et al. [11] show that information flow security on a variety of memory models (TSO, PSO and IBM 370) does not imply information flow security in the absence of a weak memory model, nor vice versa. They also show that information flow on any one of TSO, PSO or IBM370 does not imply information flow security on any of the others. Hence, the specific memory model needs to be taken into account when reasoning about information flow. Neither [10] nor [11] support compositional reasoning on programs.

In earlier work [12], we provide a compositional approach that can provide information flow assurance on *any* multi-copy-atomic[1] memory model, and instantiate it with the weak memory model of ARMv8 [4]. That approach, however, only supports a limited form of compositional reasoning where rely/guarantee properties are restricted to read and write permissions on variables, and these permissions are fixed for the duration of a program's execution. Hence, the approach is not generally applicable. Additionally, the analysis over-approximates the security level of expressions when deriving precise information is not immediately possible. This limits the completeness of the approach compared to the work in this paper where predicates are used for security levels of expressions until they can be resolved to a specific value.

3. Overview

Our logic encodes a forward pass over the code of a single thread of a concurrent program, one instruction at a time. As each instruction is considered, a number of proof obligations are checked with failure of an obligation indicating a potential insecure flow of information. The logic is sound meaning that if all proof obligations can be discharged for the entire thread, there are no security leaks. The proof obligations refer to interference from other threads via an assumption about their behaviour, and a check is made that each thread fulfils the assumptions of other threads in the program. Hence, security of the entire program can be deduced from security of each thread in isolation [14]. The forward nature of the analysis, one instruction at a time, makes it readily implementable using symbolic execution. In this section, we outline the main concepts and definitions used in the logic.

3.1. Programming language

While the instruction reordering of hardware weak memory models apply to compiled (assembly-level) code, for presentation purposes, we follow previous work in the area [10, 11, 12] by making use of a simple while-language for our logic. The intention is that the while-language can provide a high-level representation of code on any hardware memory model that we wish to consider. In addition to the standard programming constructs, this while-language includes *fences*, used in low-level programs to prevent instruction reordering [1].

Variables either belong to the set *Global* and are shared between all threads, or to the set *Local*. Distinct threads may have the same *Local* variable names, however, they refer to distinct memory addresses. Throughout examples, as seen in Figure 1, we refer to *Local* variables by r, r_1, r_2. We refer to inputs and outputs of the program by ending their variable names in *_in* and *_out* respectively. We let b refer to a boolean expression and let e, e_1, e_2 represent expressions. Our language is defined as follows[2].

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

Note that different memory models support different kinds of fences (varying depending on which types of reordering they prevent). For the purposes of this paper, we assume fence represents a *full fence* preventing all reordering.

[1]Multi-copy-atomicity is the property that a write to global memory by one thread is visible to all other threads at the same time. For non-multi-copy-atomic memory models, such as that of IBM POWER [5] and earlier versions of ARM [3], restrictions have to be made on programs in order to use our approach.

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

3.2. Security classifications and policies

We restrict security classifications of variables to operate over a two-point security lattice containing values *High* and *Low*, and structured such that $Low \sqsubseteq High$ and $High \not\sqsubseteq Low$. Similar to prior work on value-dependent logics [20, 23, 12, 26, 18, 24], security classifications of variables are expressed as predicates such that a variable is considered *Low* when its associated predicate evaluates to *true*. We refer to such predicates as *classification predicates*.

Security policies are modelled as mappings from *Global* variables to classification predicates. Our approach supports the enforcement of a global security policy, $\mathcal{L}$, which is intended to hold for all threads in a concurrent program. In the example of Figure 1, this would encode the value-dependent classifications as $\mathcal{L}(low\_out) = \mathcal{L}(z) = true$, $\mathcal{L}(high\_in) = false$ and $\mathcal{L}(x) = (\exists n \cdot z = 2 * n)$. We enforce the constraint that a variable may not refer to itself in its classification predicate, i.e., $x \notin \mathsf{vars}\ \mathcal{L}(x)$, where $\mathsf{vars}\ e$ denotes the free variables referenced in expression (or predicate) e. This simplifies the logic's rules without being overly restrictive.

This security policy serves two purposes: it describes the classifications of inputs and outputs of the program, in addition to providing a means for threads to coordinate their information flow. However, there are cases where a single global policy is insufficient. For example, consider a privileged thread capable of storing *High* information to a variable, where all others can only store *Low*. It is not possible to encode this situation given a single policy, as a more restrictive policy must be enforced on the unprivileged threads.

To support such a scenario, our logic allows for the specification of local security policies for each thread: $\mathcal{L}_G$, which constrains the thread's own writes; and $\mathcal{L}_R$, which restricts the writes due to the environment. To ensure threads still conform to the global security policy $\mathcal{L}$, all local securities policies must match the global security policy, i.e.

$$\forall x \in Global \cdot \mathcal{L}(x) = \mathcal{L}_R(x) \wedge \mathcal{L}_G(x) \tag{1}$$

For example, if a thread is constrained to write *Low* values to x when $z \leq 10$, i.e., $\mathcal{L}_G(x) = (z \leq 10)$, and its environment is constrained to write *Low* values of x when $5 < z$, i.e., $\mathcal{L}_R(x) = (5 < z)$, then globally x is guaranteed to be *Low* only when $5 < z \wedge z \leq 10$.

We introduce comparisons on classifications via predicate entailment. For example, if it can be shown that $\mathcal{L}(y)$ is true in all contexts where $\mathcal{L}(x)$ is true, formally $\mathcal{L}(x) \Rightarrow \mathcal{L}(y)$, the classification of y is the same as that of x or lower. Consequently, it is always safe to perform the assignment $x := y$, as any context where x is *Low* would imply y is *Low*. Furthermore, these comparisons can be made *context-aware* through the consideration of some state predicate P, given $P \wedge \mathcal{L}(x) \Rightarrow \mathcal{L}(y)$.

3.3. Logic judgements

Our logic operates over individual threads with a context consisting of the triple $\{P, \Gamma, D\}$, i.e., judgements are of the form $\vdash \{P, \Gamma, D\}\ c\ \{P', \Gamma', D'\}$ where c is a sequence of one or more instructions in a thread. Once established, this judgement implies our notion of noninterference over the thread, which can subsequently be composed with similar properties over other threads.

P encodes a predicate over the current state in a similar approach to Hoare logic [31]. In operation **secret_write** of Figure 1, for example, it would include $\exists n \cdot z = 2 * n + 1$ after the first increment of z, and $\exists n \cdot z = 2 * n$ after the second increment of z.

Γ encodes the classification of values held in variables based on local writes. It is defined as a partial mapping from variables to classification predicates. In operation **secret_write** of Figure 1, it would map x to *false* after the first assignment to x, and x to *true* after the second. After the assignment to r_2 in **read**, it would map r_2 to $\exists n \cdot v = 2 * n$ where v denotes the value of z when the assignment occurred.

To simplify the use of Γ, we introduce the notation $\Gamma\langle x\rangle$, which is defined for all x. For *Global* variables, it equals $\Gamma(x)$ when x is in the domain of Γ, and $\mathcal{L}(x)$ otherwise. Since *Local* variables cannot be read by another thread (or attacker), they are able to hold values of any classification which is encoded by the predicate *false*.

$$\Gamma\langle x\rangle \equiv \begin{cases} \Gamma(x) & \text{if } x \in \mathsf{dom}\ \Gamma \\ \mathcal{L}(x) & \text{if } x \in Global \\ false & \text{otherwise} \end{cases} \tag{2}$$

We introduce the notation $\Gamma \vdash e : t$ for determining the classification t of an expression e, which evaluates to a conjunction over the classifications of the variables referenced by the expression.

$$\Gamma \vdash e : t \;\equiv\; t = \bigwedge_{y \in \mathsf{vars}\ e} \Gamma\langle y \rangle \tag{3}$$

D, the final component of our context, comprises four functions used to enable reasoning about instruction reordering. These are detailed in Section 3.5.

3.4. Rely/guarantee reasoning

A program is assumed to consist of a static set of threads, each of which requires a valid logic judgement. Rely/guarantee reasoning is then employed to compose these judgements and establish noninterference over the entire program. To achieve this, each thread is coupled with a rely, R, and guarantee, G. These definitions follow the standard style of rely/guarantee reasoning [14], where R and G are predicates describing relations on states. A thread assumes all other threads in its environment conform to the relation R and guarantees its own instructions conform to G. For rely/guarantee reasoning a proof of *compatibility* between the relations is necessary such that, for any two threads in the program, the rely relation of one contains the guarantee relation of the other, i.e., the guarantee is stronger than the rely. As the R and G relations apply to the entire execution of a thread, this proof of compatibility is relatively straightforward.

Following [14], we ensure two constraints on the relations R and G. The first of these restricts R to exhibit transitivity and reflexivity. Transitivity allows for the consideration of multiple environment steps at once, whilst reflexivity encodes the possibility of no environment steps. G is restricted to exhibit reflexivity, avoiding constraints which would limit the current thread's instructions to those that mutate the global state.

In our logic, a thread's instructions must conform to its guarantee G. Additionally, P and Γ must be *stable* with respect to the rely condition R, i.e., they cannot be invalidated by any interleaved environment steps. For example, for the thread executing the operation **read** of Figure 1, it is necessary to show that the environment will not change z between its initial read and the later check that it is equal to r_1 whenever the check succeeds. This would be the case if R included the conjunct $z' = z$. This implies the value of z is preserved under all conditions, resulting in the predicate $z = r_1$ being considered stable. Alternatively, the constraint on the environment could be weakened to $z \leq z'$, stating that the environment will only increase z. In this case, the predicate following z's read must be weakened to achieve stability. This is done in a way that preserves information, resulting in $r_1 \leq z$.

The form of R and G that we adopt is a conjunction of predicates of the form $c \Rightarrow r(x, x')$ where c is a stable condition on the program state, and $r(x, x')$ is a predicate relating *Global* variable x and the value of x in the post state of a transition, x'. For example, the thread executing the operation **secret_write** of Figure 1, relies on other threads not changing z. Hence, its R would include the conjunct $true \Rightarrow z' = z$, or equivalently $z' = z$. While this form for R and G is slightly more restrictive than general predicates, as argued in [18] they better support automation for large or complex code.

To simplify the automation of the logic's rules (see Section 7.1), we introduce the set R_{var} of tuples of the form (x, c, r), where x is a variable, c is a predicate and r a relation on values of x and x', stating that the environment is constrained to modifying x in accordance with r if c holds. The rely condition R is then the conjunction of the predicates $c \Rightarrow r(x', x)$ of each tuple, capturing the allowable changes to all variables in the program state.

$$R \;\equiv\; \bigwedge_{(x,c,r) \in R_{var}} c \Rightarrow r(x, x') \tag{4}$$

We have an identically typed set of tuples G_{var} for guarantees.

$$G \;\equiv\; \bigwedge_{(x,c,r) \in G_{var}} c \Rightarrow r(x, x') \tag{5}$$

3.5. Instruction reordering

Weak memory models allow for the reordering of program instructions based on certain constraints; the most common being that the instructions do not access the same memory addresses, and hence are independent. We base our logic on the operational semantics of weak memory models by Colvin and Smith [6, 7] in which a reordering relation $\overset{R}{\Leftarrow}$ is defined for the specific memory model. This relation establishes the conditions under which two instructions may reorder, such that $\alpha \overset{R}{\Leftarrow} \beta$ implies that a processor with a memory model described by $\overset{R}{\Leftarrow}$ could execute $\alpha;\ \beta$ by executing β before α. To facilitate this, if and while statements are treated as guard instructions together with non-deterministic choice. For example, the program if b then c_1 else c_2 is treated as $([b];\ c_1) \sqcap ([\neg\ b];\ c_2)$ where $[b]$ is a guard with condition b and $\sqcap$ is non-deterministic choice. A guard can be executed only when its condition is true. Hence, the nondeterministic choice will be resolved to follow the path with the true guard.

As an example of the semantics, consider the snippet $x := 0;\ z := 1$. These instructions operate on distinct memory addresses, x and z, allowing weak memory models such as ARM or POWER to reorder their effects, encoded as $x := 0 \overset{R}{\Leftarrow} z := 1$. Concurrently executing threads may therefore observe this snippet as if the program $z := 1;\ x := 0$ had been executed.

Additionally, the semantics allows for *forwarding*, in which dependent instructions can be reordered by transforming the later instruction [1]. For example, the snippet $x := 0;\ y := x$ (where there is a dependency due to variable x) may be observed as $y := 0;\ x := 0$, where the value of x is forwarded from its write to the subsequent read which can then be reordered with the write.

To account for such reordering in our logic, we need to consider at each point in the program the possibility of *earlier* instructions being reordered and hence not having occurred yet, and *later* instructions being reordered and hence having occurred already. To facilitate this, we introduce four functions which appear in component D of our context. The first of these, W, maps instructions of a thread to sets of global variables. $W(\alpha)$ denotes the set of global variables which are known to be *up-to-date* when instruction α is reached, i.e., all writes to them before α in the program have occurred. Note that only the global variables are of concern, as local variables cannot influence other threads. Consequently, instruction reordering cannot introduce new behaviours between threads without a global variable.

The value of $W(\alpha)$ is initially the set of all global variables, and evolves as the program progresses. For example, given the code $z := x;\ y := x;\ z := 0;\ y := x$, after the first assignment $W(y := x)$ contains z since the first assignment must occur before the second due to the dependency on variable x. On a memory model such as ARM or POWER, however, after the third assignment $W(y := x)$ does not contain z since the fourth assignment can be reordered before the third. To define the update to W at instruction α, we use the current value of W to provide the required context associated with α's position in the code.

Let $W + [\alpha]$ be the update of W when instruction α occurs. After the first assignment above, $W(y := x)$ is extended to include z plus any variables whose prior writes in the program cannot be reordered after $z := x$. This can be captured by $(W + [z := x])(y := x) = W(y := x) \cup W(z := x)$; note that $W(z := x)$ contains z since no writes to z can be reordered with $z := x$. After the third assignment, z is removed from $W(y := x)$. This can be captured by $(W + [z := 0])(y := x) = W(y := x) - wr(z := 0)$ where $wr(\alpha)$ is the set of global variables written to by α.

As the above illustrates, the update of W at each instruction depends on whether it can be reordered with preceding instructions. It is defined as follows where $\beta_{\langle\alpha\rangle}$ denotes the instruction β with the effects, if any, of forwarding the earlier assignment α, i.e., $y := f_{\langle x:=e\rangle}\ =\ y := f[e/x]$ where $f[e/x]$ replaces each free occurrence of x in f with e.

$$(W + [\alpha])(\beta)\ \equiv\ \begin{cases} W(\beta_{\langle\alpha\rangle}) - wr(\alpha) & \text{if } \alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle} \\ W(\beta) \cup W(\alpha) & \text{otherwise} \end{cases} \tag{6}$$

When updating W for an instruction α, a subsequent instruction β could execute out-of-order given $\alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle}$ holds. As a result, we remove the writes of α from β's W mapping. Additionally, in the event of forwarding, it is necessary to use $W(\beta_{\langle\alpha\rangle})$ rather than $W(\beta)$, as forwarding may weaken constraints from other instructions earlier than α, allowing for writes to other variables to execute out-of-order.

To illustrate this weakening case, consider $x := y;\ y := 5;\ z := y$ in which forwarding enables reordering of $y := 5$ and $z := y$. Assume before the first assignment that $W(z := 5) = \{z\}$. Since $x := y \overset{R}{\Leftarrow} z := 5_{\langle x:=y\rangle}$, after the first assignment $W(z := 5) = \{z\} - \{x\} = \{z\}$. Similarly, after the second assignment $W(z := 5) = \{z\} - \{y\} = \{z\}$. Therefore, according to (6), after the second assignment $W(z := y) = W(z := 5) - \{y\} = \{z\}$. Hence, when the third assignment is evaluated, neither x nor y are considered to be up-to-date.

Our second function L maps instructions to sets of global variables such that $L(\alpha)$ denotes the set of global variables whose prior reads in the program have definitely occurred.[3] Initially L maps each instruction to the set of all global variables. Updates to L can be calculated in an analogous fashion to those of W as follows (where $rd(\alpha)$ is the set of global variables read by α).

$$(L + [\alpha])(\beta) \;\equiv\; \begin{cases} L(\beta_{\langle\alpha\rangle}) - rd(\alpha) & \text{if } \alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle} \\ L(\beta) \cup L(\alpha) & \text{otherwise} \end{cases} \tag{7}$$

When reasoning about proof obligations of an instruction α, we restrict the state predicate P to those variables in $W(\alpha) \cap L(\alpha)$, i.e., we restrict P to those variables whose previous accesses have definitely occurred. This restriction to P, denoted $P_{\restriction\alpha}$, is defined as follows, where $globals(P)$ denotes the global variables free in P.

$$P_{\restriction\alpha} \;\equiv\; \exists\, y_1, ..., y_n \cdot P \qquad\qquad \text{where } \{y_1, ..., y_n\} = globals(P) - (W(\alpha) \cap L(\alpha)) \tag{8}$$

The third function, I, keeps track of references in the context (specifically, in P and Γ) which are unaffected by potential reordering. To illustrate why this is needed, consider the snippet $r := x$; $[z < 1]$ with initial conditions $0 \leq z$. We assume that the environment can increment z, i.e., R includes $z \leq z'$, and can modify x arbitrarily, and that $\mathcal{L}(x) = (z = 0)$, i.e., x is *Low* given $z = 0$. Let v represent the value of z during the execution of the assignment $r := x$. Thus, the context after $r := x$ and zero or more environment steps must contain $P = (0 \leq v \wedge v \leq z)$, as initial conditions established that z was greater than 0 and the environment may have incremented it afterwards, and $\Gamma(r) = (v = 0)$, as the value read from x was *Low* if $z = 0$ was true during the load.

Executing the guard $[z < 1]$ from this context, and using v' to represent the value of z during its execution, would update P to $0 \leq v \wedge v \leq v' \wedge v' = 0 \wedge v' \leq z$ and leave $\Gamma(r)$ unchanged. This context can therefore establish $v = 0$, i.e., the value of z when $r := x$ executed must have been 0, and, therefore, the value in r must be *Low* due to the security policy for x. This relies on the test of z executing after the load $r := x$, to bound the value of z. However, under a weak memory model such as that of ARM or POWER, these two instructions can be reordered. As a result, x could be read in a state where z is not 0, resulting in r potentially holding *High* data.

This issue can be attributed to the *indirect* reference to z added to the context by the instruction $r := x$, in the form of $v \leq z$. We refer to this reference as indirect as the instruction does not directly access the variable z. Consequently, its ordering with instructions that directly modify or test z is not immediately known under weak memory models.

To successfully manage such examples it is necessary to take additional precautions when introducing new global variable references to the context. When introducing a *direct* reference, i.e. a variable that is used in the instruction, there must be no indirect references to the same variable already in the context due to reorderable instructions. This is illustrated in the prior example due to the introduction of a direct reference to z when considering $[z < 1]$, where an indirect reference exists due to the reorderable instruction $r := x$. Hence, we define I to be the function which initially maps each instruction to all global variables, and is updated as follows (where $ind(\alpha)$ is the set of indirect references introduced by α, which can be computed from R and the security policies of variables occurring in α).

$$(I + [\alpha])(\beta) \;\equiv\; \begin{cases} I(\beta_{\langle\alpha\rangle}) - ind(\alpha) & \text{if } \alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle} \\ I(\beta) \cup I(\alpha) & \text{otherwise} \end{cases} \tag{9}$$

This function is defined such that $I(\alpha)$ returns the set of variables that have no indirect references introduced by reorderable instructions from the perspective of α. Prior to computing the strongest post-condition for an instruction α, we remove all existing references to global variables in α from P unless they are in $I(\alpha)$. The reduced P, denoted $P_{\backslash\alpha}$, is defined as follows.

$$P_{\backslash\alpha} \;\equiv\; \exists\, y_1, ..., y_n \cdot P \qquad\qquad \text{where } \{y_1, ..., y_n\} = globals(\alpha) - I(\alpha) \tag{10}$$

Similarly, we remove all references to variables in α from $\Gamma(x)$ for each variable x unless they are in $I(\alpha)$. In this case, we use universal quantification to ensure that the classification in Γ will be true only when it does not refer to the

[3]We use L (for load) rather than R (for read) to avoid overloading the use of R, which is used in our logic for rely conditions.

removed variables. For example, if $\Gamma(x) = (z = 0)$ and z needs to be removed then we want $\forall z \cdot z = 0$ which evaluates to *false*. The reduced Γ, denoted $\Gamma_{\backslash\alpha}$, is defined as follows.

$$\Gamma_{\backslash\alpha} \quad \equiv \quad \lambda x \cdot \forall y_1, ...y_n \cdot \Gamma(x) \qquad \text{where } \{y_1, ..., y_n\} = globals(\alpha) - I(\alpha) \tag{11}$$

The final function we require, U, keeps track of global variables not used in the proof obligations of earlier reorderable instructions. We define U to be the function which initially maps each instruction to all global variables, and is updated as follows (where $used(\alpha)$ is the set of global variables which appear free in the proof obligations associated with α).

$$(U + [\alpha])(\beta) \quad \equiv \quad \begin{cases} U(\beta_{\langle\alpha\rangle}) - used(\alpha) & \text{if } \alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle} \\ U(\beta) \cup U(\alpha) & \text{otherwise} \end{cases} \tag{12}$$

This function is required to handle cases where a later instruction may execute earlier, enabling the environment to modify the state in such a way that earlier proof obligations are invalidated. For example, consider the snippet $x := 1;\ l := 0$ with initial conditions $l = 1 \wedge z = 0$. We assume that the environment cannot modify l and can modify z when $l \neq 1$, i.e., R includes $l = l'$ and $l = 1 \Rightarrow z = z'$, and that this thread is constrained to only write to x when $z = 0$, i.e., G includes $z \neq 0 \Rightarrow x = x'$. Note that the initial conditions are stable under this R, as $l = 1$ ensures $z = 0$ holds across arbitrary initial environment steps.

When considering $x := 1$ under the given initial conditions, it is evident that the instruction will conform to the thread's guarantee as $z = 0$ must hold. However, the subsequent instruction $l := 0$ may execute prior to $x := 1$ on ARM and POWER processors. While the instruction does not directly influence the variables considered in the guarantee check, z and x, it allows the environment to arbitrarily modify z, such that $z \neq 0$ when $x := 1$ is executed. Consequently, such memory models enable execution traces that may result in the thread violating its guarantee.

The dependency between the instruction $x := 1$ and z is captured via $z \in used(x := 1)$, as z is present in proof obligations constraining how the value of x can be modified. Therefore, as the two instructions can reorder, this results in $z \notin (U + [x := 1])(l := 0)$, enabling detection of a possible violation when considering $l := 0$.

4. Logic

Our logic operates on the code of a single thread for which we have the predicates R and G to account for interactions with other threads, and the functions $\mathcal{L}$, $\mathcal{L}_R$ and $\mathcal{L}_G$ capturing the required global and local security policies. We assume these predicates and functions are available to all our definitions.

The logic encodes a forward pass over the thread's code, computing the context across instructions and interleaved environment steps. The default initial context has *true* for P, an empty mapping for Γ and each function, W, L, I and U, in D mapping instructions to all global variables. However, any wellformed context (see Section 5.6) would be acceptable.

4.1. Context updates due to assignments

The update to the context corresponding to an assignment instruction requires the introduction of temporary variables to retain information shared between P and Γ. To illustrate this, consider the implications of an assignment $x := e$ when both P and $\Gamma(y)$ refer to x. Any references to x in P must be removed, and the new definition of x added to P (as is done when calculating the strongest postcondition of an instruction [31]). However, the reference to x already in $\Gamma(y)$ refers to the value of x at the time $\Gamma(y)$ was updated, not the new value e. Hence, the reference to x in $\Gamma(y)$ also needs to be removed, but without removing the relationship with the old value of x. To achieve this, we replace existing references to x in both P and Γ with a fresh temporary variable v. Given $\Gamma \vdash e : t$ the update to the context corresponding to $x := e$ is defined as follows.

$$\{P, \Gamma, D\} + [x := e] \quad \equiv \quad \{P[v/x] \wedge x = e[v/x], (\Gamma[x \mapsto \forall y_1, ..., y_n \cdot t])[v/x], D + [x := e]\} \tag{13}$$

where $\Gamma[x \mapsto t]$ updates the value of $\Gamma(x)$ to t, $\{y_1, ..., y_n\} = globals(P) - (W[x := e] \cap L[x := e])$, and $D + [x := e]$ updates each of the constituent functions of D according to definitions (6), (7), (9) and (12). As t allows for the introduction of indirect variable references, the quantification of variables $y_1, ..., y_n$ in t ensures that all indirect references added to $\Gamma(x)$ are in $W(x := e) \cap L(x := e)$.

To illustrate why this is required, consider the snippet $z := 0;\ r := x$ where $\mathcal{L}(x) = (z = 0)$. When computing the context updates for this snippet without such quantification, it is possible to derive a postcondition in which $z = 0$ holds and $\Gamma(r) = \mathcal{L}(x)$, consequently establishing r contains *Low* information. However, as demonstrated in Section 3.5, these instructions may reorder and enable an execution in which r contains *High* information. The quantification of z in $\Gamma(r) = \forall z \cdot \mathcal{L}(x)$ captures this reorderable behaviour, allowing $\Gamma(r)$ to instead simplify to *false*, encoding the possibility of *High* information.

4.2. Context updates due to environment steps

As with assignment steps, temporary variables are introduced when updating the context due to environment steps which may occur after an instruction α. We consider the effect of environment steps on each component, P, Γ and D, of the context separately. Given P is the predicate in the context following instruction α, a stable predicate $P + R_\alpha$ which takes into account zero or more environment steps is defined by conjoining P and a restricted form of R, renaming each reference to a variable $x_i \in Global$ before the environment steps to a temporary variable v_i, and renaming each reference to a variable x_i' after the environment steps to x_i.

$$P + R_\alpha \equiv (P \wedge \bigwedge_{(x,c,r)\in R_{var}} (\forall y_1,\dots,y_m \cdot c) \Rightarrow r(x,x'))[v_1,...,v_n,x_1,..,x_n/x_1,...,x_n,x_1',...,x_n'] \quad (14)$$

where $\{y_1,\dots,y_m\} = globals(P) - (W(\alpha) \cap L(\alpha))$. The restriction to the condition c of each element (x,c,r) of R_{var} is identical to that used to restrict t in the update to Γ for an assignment (see Section 4.1). It is required again to ensure that indirect references introduced by α are not taken into account when evaluating the change of variables allowed by R unless they are in $W(\alpha) \cap L(\alpha)$. For example, consider again the snippet $z := 0;\ r := x$ where $R = (z = 0 \Rightarrow x' = 0)$. Without the quantification, we would be able to derive that $r = 0$ holds after the snippet is executed. However, due to reordering this may not be the case. The quantification means R is effectively $(\forall z \cdot z = 0) \Rightarrow x' = 0$ which is equivalent to *true* (since $\forall z \cdot z = 0$ is *false*) allowing any change to x. Note that the quantification is not over the whole predicate $c \Rightarrow r(x,x')$.

$P + R_\alpha$ is stable as the renaming ensures it contains no references to *Global* variables, except those renamed from the post-state variables of $r(x,x')$ of each (x,c,r) in R_{var}. Since each $r(x,x')$ is required to be transitive, additional environment steps will not change the relationship between these variables. To illustrate this definition, consider the example of $R = (z \leq z')$ and $P = (r = z \wedge x = 0)$ after instruction α. An application of this update would produce $P + R_\alpha = (r = v_1 \wedge v_2 = 0) \wedge (v_1 \leq z)$ which, ignoring temporary variables, is equivalent to $r \leq z$. That is, the constraint on r and z is weakened across the update, from $r = z$ to $r \leq z$, while information regarding x has been lost, due to a lack of restrictions in R.

When updating Γ due to environment steps, predicates describing classifications are renamed using the same temporary variables as in definition (14). This preserves the relationships between variables in P and Γ. The resulting predicates in Γ are obviously stable as they do not refer to *Global* variables. Additionally, since Γ represents the classifications of the values of variables based on local writes, the domain of Γ should only include variables where those classifications are stable.

Assume we have $\Gamma(x) = t$ stating that the variable x holds a value with classification t. If an environment step modifies x, its new value is constrained by the security policy $\mathcal{L}_R(x)$ regardless of t. Hence, the new classification of x's value is $t \wedge \mathcal{L}_R(x)$ which is *Low* if, and only if, its value t was *Low* prior to the environment step and if the environment's write was also *Low*, i.e., $\mathcal{L}_R(x)$ is true. Therefore, if it is possible to demonstrate $\mathcal{L}_R(x)$ is true in the current state P, the classification of x's value can be considered unmodified. Furthermore, the classification of x's value is trivially unmodified if the environment is prevented from writing to x due to constraints in R. We encode these properties as the set of variables low_or_eq, which forms the domain of a stable Γ.

$$\mathsf{low\_or\_eq}\ P \equiv \{x \mid (P \Rightarrow \mathcal{L}_R(x)) \vee (P \wedge R \Rightarrow x' = x)\} \quad (15)$$

The update of Γ due to R is then defined using the stable predicate $P + R_\alpha$ to determine the set of low_or_eq variables and set the domain accordingly.

$$\Gamma + R_\alpha \equiv \lambda x \in \mathsf{low\_or\_eq}\ (P + R_\alpha) \cdot \Gamma(x)[v_1,...,v_n/x_1,...,x_n] \quad (16)$$

where $v_1, ...v_n$ are the same temporary variables used in calculating $P + R_\alpha$.

The functions in D are not affected by environment steps (as they refer only to dependencies between local steps of the thread). Hence, the update to the context by environment steps is defined as follows.

$$\{P, \Gamma, D\} + R_\alpha \quad \equiv \quad \{P + R_\alpha, \Gamma + R_\alpha, D\} \tag{17}$$

4.3. Rules

The rules of our logic are shown in Figure 2. There is one rule for each type of instruction and a consequence rule for modifying the context to match what is required for other rules.

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

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

$$\textsc{Conseq}\ \frac{\begin{array}{cc} & P_2, \Gamma_2, D_2 \geq P_1, \Gamma_1, D_1 \\ \vdash \{P_1, \Gamma_1, D_1\}\ c\ \{P'_1, \Gamma'_1, D'_1\} & P'_1, \Gamma'_1, D'_1 \geq P'_2, \Gamma'_2, D'_2 \end{array}}{\vdash \{P_2, \Gamma_2, D_2\}\ c\ \{P'_2, \Gamma'_2, D'_2\}}$$

$$\textsc{If}\ \frac{\begin{array}{cc} \Gamma \vdash b : t & \vdash (\{P_{\backslash[b]} \land b, \Gamma, D + [b]\} + R_{[b]})\ c_1\ \{P', \Gamma', D'\} \\ P_{\restriction[b]} \Rightarrow t & \vdash (\{(P_{\backslash[b]} \land \neg\, b, \Gamma, D + [b]\} + R_{[b]})\ c_2\ \{P', \Gamma', D'\} \end{array}}{\vdash \{P, \Gamma, D\}\ \mathsf{if}\ (b)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{P', \Gamma', D'\}}$$

$$\textsc{While}\ \frac{\begin{array}{cc} \Gamma \vdash b : t & \\ P_{\restriction[b]} \Rightarrow t & \vdash (\{P_{\backslash[b]} \land b, \Gamma, D + [b]\} + R_{[b]})\ c\ \{P, \Gamma, D\} \end{array}}{\vdash \{P, \Gamma, D\}\ \mathsf{while}\ (b)\ \mathsf{do}\ c\ (\{P_{\backslash[b]} \land \neg\, b, \Gamma, D + [b]\} + R_{[b]})}$$

$$\textsc{AssignL}\ \frac{x \notin \mathit{Global} \qquad \Gamma \vdash e : t}{\vdash \{P, \Gamma, D\}\ x := e\ (\{P_{\backslash x:=e}, \Gamma_{\backslash x:=e}, D\} + [x := e] + R_{x:=e})}$$

$$\textsc{AssignG}\ \frac{\begin{array}{cc} & P_{\restriction x:=e} \land \mathcal{L}_G(x) \Rightarrow t \\ x \in \mathit{Global} & \mathsf{guar}\ P_{\restriction x:=e}\ (x := e) \\ \Gamma \vdash e : t & \mathsf{valid}\ P_{\restriction x:=e}\ \Gamma\ (x := e) \end{array}}{\vdash \{P, \Gamma, D\}\ x := e\ (\{P_{\backslash x:=e}, \Gamma_{\backslash x:=e}, D\} + [x := e] + R_{x:=e})}$$

$$\textsc{Fence}\ \frac{}{\{P, \Gamma, D\} f\ \{P, \Gamma, D + [f]\}}$$

Figure 2: Rules of the logic.

The Skip and Seq rules follow the standard structure seen in Hoare logic [31]. For the Conseq rule, we introduce an ordering on contexts.

$$P, \Gamma, D \geq P', \Gamma', D' \quad \equiv \quad \forall\, x.\ (P \Rightarrow P') \land (P \land \Gamma'\langle x\rangle \Rightarrow \Gamma\langle x\rangle) \land D \geq D' \tag{18}$$

where $D \geq D'$ when for all instructions α the functions in D' return a subset of the return value of the corresponding functions in D. Therefore, the stronger context P, Γ, D is aware of all instruction dependencies known in the weaker P', Γ', D'. For the predicate P, we use implication as traditionally done in the consequence rules of Hoare logic. For Γ, the stronger state must have classifications equal to or lower than those in the weaker. Hence, all valid information flow reliant on *Low* classifications in the weaker state must be valid in the stronger.

The If rule restricts the classification of the guard to being *Low*, therefore avoiding potential side-channels due to timing differences of different branch outcomes [23, 12]. The guard result is introduced into the respective branch, along with an application of $+R_\alpha$ to enforce stability. A similar approach is used for the WHILE rule, establishing a *Low* guard and encoding the loop invariant. Both of these rules are not immediately applicable to automation in this form but can be specialised via combination with the CONSEQ rule as in prior work [20].

The simplest assignment rule, ASSIGNL, considers writes to *Local* variables. Due to their static *High* classification, it is not necessary to perform classification comparisons for these assignments.

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

Second, it is necessary to demonstrate the assignment conforms to the thread's guarantee G. This is achieved by computing the original and primed states for the instruction and comparing them with G.

$$\mathsf{guar}\, P\, (x := e) \;\; \equiv \;\; P \wedge x' = e \wedge (\forall\, y \cdot x \not\sim y \Rightarrow y = y') \Rightarrow G \tag{19}$$

where $x \not\sim y$ means x and y are distinct variables.

Finally, it is necessary to consider the effects of this assignment on instructions earlier in the program order due to potential reorderings. To achieve this, we compute a series of over-approximations to capture *potential* cases of interference and compare these with the components of D to detect possible violations of the desired functional and information flow properties.

First we consider cases where the assignment $x := e$ may modify security classifications, i.e. x is a control variable. We decompose these effects into *falling* and *rising* cases. The falling case captures situations in which a variable's classification is potentially considered *High* in the pre-state and *Low* in the post-state. This case may introduce an information leak if *High* information is not cleared from a falling variable prior to its classification change. Therefore, a control variable assignment must ensure all variables with potentially falling classifications hold *Low* information. To capture this, we first define the concept of a *potentially* falling classification.

$$\begin{aligned}\mathsf{falling}\, P\, \Gamma\, (x := e) \;\; \equiv \;\; & \{y \mid x \in \mathit{vars}(\mathcal{L}(y)) \wedge \neg\, (P \Rightarrow \mathcal{L}(y)) \wedge \\ & \neg\, (P + [x := e] + R_{x:=e} \Rightarrow \neg\, \mathcal{L}(y))\}\end{aligned} \tag{20}$$

where $P + [x := e] \;\equiv\; P[v/x] \wedge x = e[v/x]$ with v being a fresh temporary variable. This definition includes all variables for which a syntactic dependency on x is known, due to its reference in $\mathcal{L}(y)$, but it is not possible to demonstrate either a *Low* classification in the pre-state or a *High* classification in the post-state, the two contradictory cases for a falling classification.

To avoid security policy violations, it must be shown that all falling variables hold a *Low* value when $x := e$ executes. This can be captured by testing their security values in Γ and requiring that all of their earlier writes (in particular, the write that sets the value to *Low*) have definitely occurred.

$$\mathsf{falling}\, P\, \Gamma\, (x := e) \subseteq \{y \mid (P \Rightarrow \Gamma(y)) \wedge y \in W(x := e)\} \tag{21}$$

The rising case describes the inverse of the falling in which a variable is potentially *Low* in the pre-state and *High* in the post-state. Such a situation may introduce a leak if the classification change can reorder with earlier reads of the rising variable, potentially resulting in these reads returning *High* information where *Low* is anticipated. To illustrate, consider the example $\mathit{out} := y;\; x := e$, where *out* is visible to an attacker, therefore constraining the first assignment to writing a *Low* value. Assuming a pre-state capable of demonstrating a *Low* classification for y, the example appears to implement the desired property. However, these instructions may reorder and execute as $x := e;\; \mathit{out} := y$ enabling a write of *High* information to *out* if y's classification rises after $x := e$ is executed.

To handle this situation, we introduce a definition to capture variables with rising classifications, which parallels that of the falling set.

$$\begin{aligned}\mathsf{rising}\, P\, \Gamma\, (x := e) \;\; \equiv \;\; & \{x \mid x \in \mathit{vars}(\mathcal{L}(y)) \wedge \neg\, (P \Rightarrow \neg\, \mathcal{L}(y)) \wedge \\ & \neg\, (P + [x := e] + R_{x:=e} \Rightarrow \mathcal{L}(y))\}\end{aligned} \tag{22}$$

The required condition for a variable with a rising classification is that its earlier reads have definitely occurred.

$$\mathsf{rising}\, P\, \Gamma\, (x := e) \subseteq L(x := e) \tag{23}$$

It is also necessary to consider how instruction reordering may influence the guarantee proof obligations of prior instructions. We introduce the definition $\mathsf{stronger}$ to capture those cases where the current instruction may result in an earlier instruction having stronger proof obligations due to an effect of the assignment on G. This occurs when a c condition of G *potentially* becomes true, i.e., when it cannot be proven to be *true* before the instruction and not proven to be *false* after it.

$$\begin{aligned}\mathsf{stronger}\ P\ \Gamma\ (x := e) \ \equiv\ & \{y \mid \exists (c, r) \cdot (y, c, r) \in G_{var} \wedge x \in vars(c) \wedge \\ & \neg\,(P \Rightarrow c) \wedge \neg\,(P + [x := e] + R_{x:=e} \Rightarrow \neg\, c)\}\end{aligned} \tag{24}$$

Given $y \in \mathsf{stronger}\ P\ \Gamma\ (x := e)$, it is possible that a condition c may hold after $x := e$, resulting in a new constraint r on assignments of the form $y := f$. If such an assignment exists prior to $x := e$, i.e., $y := f;\ x := e$, and the two instructions can reorder, the value written to y must be constrained by r to enforce the desired guaranteed on all possible executions. This can be established by ensuring that all writes to y have been executed prior to $x := e$, i.e.,

$$\mathsf{stronger}\ P\ \Gamma\ (x := e) \subseteq W(x := e) \tag{25}$$

We also need to consider whether the assignment can weaken restrictions on the environment, potentially invalidating earlier proof obligations. We capture variables which the environment may modify to a greater extent when reordered with a later assignment $x := e$ as follows.

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

Note that this definition is simpler in form to those above due to c being constrained to be stable under R (see Section 3.4). We require that all variables in weaker have not been used in proof obligations of earlier instructions that the assignment can reorder with. This captures the example illustrated in Section 3.5 motivating U.

$$\mathsf{weaker}\ P\ (x := e) \subseteq U[x := e] \tag{27}$$

Finally, we consider the case where an earlier instruction assumed that a variable x was in the domain of Γ, but a later instruction modifies $\mathsf{low\_or\_eq}$ such that this is no longer the case. To simplify the definition of the required proof obligation, we define

$$\mathsf{low\_or\_eq\_exp}\ x \ \equiv\ \mathcal{L}_R(x) \vee (R \Rightarrow x' = x) \tag{28}$$

Given $\mathsf{low\_or\_eq\_exp}\ x$ is true for P, the variable x is in $\mathsf{low\_or\_eq}\ P$. We capture the variables that are potentially removed from the domain of Γ as follows.

$$\begin{aligned}\mathsf{shrink}\ P\ \Gamma\ (x := e) \ \equiv\ & \{y \mid x \in vars(\mathsf{low\_or\_eq\_exp}\ y) \wedge \neg\,(P \Rightarrow \neg\,\mathsf{low\_or\_eq\_exp}\ y) \wedge \\ & \neg\,(P + [x := e] + R_{x:=e} \Rightarrow \mathsf{low\_or\_eq\_exp}\ y)\}\end{aligned} \tag{29}$$

The required condition for a variable whose classification may become unstable is that its earlier reads have definitely occurred.

$$\mathsf{shrink}\ P\ \Gamma\ (x := e) \subseteq L(x := e) \tag{30}$$

We define $\mathsf{valid}\ P\ \Gamma$ to disallow all of the cases where instructions can invalidate proof obligations of earlier reorderable instructions. That is, $\mathsf{valid}\ P\ \Gamma\ (x := e)$ is the conjunction of the conditions (25), (21), (23), (27) and (30).

The Fence rule simply updates D to capture the enforced ordering associated with the given fence. For our full fence, all functions in D would be restored to mapping all instructions to the set of all variables.

4.4. Application of logic

As a demonstration of the logic, we apply it to the example from Figure 1 running on ARMv8 [4]. We show that it is not secure, and show how to make it secure by adding fences. Each operation is treated as being run by an individual thread with a default initial context, except for **secret_write** whose thread we assume has been defined to start from a state satisfying the predicate $\exists\, n \cdot z = 2 * n$.

We first establish the security policy and rely and guarantee conditions. Data flows from one of the two inputs *high_in* or *low_in* to one of the two outputs *high_out* or *low_out*. The security policy should prevent the flow of information from *high_in* to *low_out*. To express this, we establish $\mathcal{L}(high\_in) = \mathcal{L}(high\_out) = false$ and $\mathcal{L}(low\_in) =$

$\mathcal{L}(low\_out) = true$. Communication between the two threads occurs via x and z, such that x holds *High* information when z is odd and *Low* otherwise. This can be expressed as $\mathcal{L}(x) = (\exists\, n \cdot z = 2 * n)$. We set $\mathcal{L}(z) = true$, as it never holds *High* data.

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

$$\begin{aligned} G_{secret\_write} &\equiv z \le z' \\ G_{otherwise} &\equiv z = z' \\ R_{read} &\equiv z \le z' \\ R_{secret\_write} &\equiv z = z' \\ R_{otherwise} &\equiv true \end{aligned} \qquad \begin{aligned} \mathcal{L}_G(x)_{secret\_write} &\equiv \exists\, n \cdot z = 2 * n \\ \mathcal{L}_G(x)_{otherwise} &\equiv true \\ \mathcal{L}_R(x)_{secret\_write} &\equiv true \\ \mathcal{L}_R(x)_{otherwise} &\equiv \exists\, n \cdot z = 2 * n \end{aligned}$$

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

We will first consider the **write** operation. As x is a *Global*, it is necessary to apply AssignG. For $x := low\_in$, the proof obligations are trivial. The classification of the expression *low_in* is *true* via a consultation of $\mathcal{L}$. As a result, $P \wedge \mathcal{L}_G(x) \Rightarrow true$ can be discharged. Additionally, the guarantee for the **write** operation only constrains z, so a write to x can be ignored. As x cannot influence classifications nor any c conditions in R_{var} and G_{var} (they are all *true*), it cannot cause a proof obligation to fail due to reordering with an earlier instruction. As there are no further instructions in this operation, we have verified that it has no security leaks.

For operations with more than one instruction, we need to consider the possibility of instruction reordering. The reordering relation for ARMv8 is defined by Colvin and Smith in [6, 7]. The allowed reorderings for assignments and branches are given below.

$$\begin{array}{rcll} x := e & \overset{R}{\Leftarrow} & y := f & \text{iff } x \not\sim y,\ x \notin vars(f),\ y \notin vars(e) \text{ and } vars(e) \cap vars(f) \cap Global = \varnothing \\ [b] & \overset{R}{\Leftarrow} & x := e & \text{iff } x \in Local,\ x \notin vars(b) \text{ and } vars(b) \cap vars(e) \cap Global = \varnothing \\ x := e & \overset{R}{\Leftarrow} & [b] & \text{iff } x \notin vars(b) \text{ and } vars(b) \cap vars(e) \cap Global = \varnothing \\ [b_1] & \overset{R}{\Leftarrow} & [b_2] & \text{iff } vars(b_1) \cap vars(b_2) \cap Global = \varnothing \end{array}$$

We will now consider the **secret_write** operation. As this thread is more complex, we will establish the context between each line. To simplify the presentation, we omit the input variable *high_in* from the functions of D. We also only show those functions in D which are relevant to this operation and use _ to stand for any value.

$\{P = (\exists\, n \cdot z = 2 * n), \Gamma = \{\}, W = L = \{\_ \mapsto \{x, z\}\}\}$
$z := z + 1$
$\{P = (\exists\, n \cdot z = 2 * n + 1), \Gamma = \{z \mapsto true\}, W = L = \{z := \_ \mapsto \{x, z\}, x := \_ \mapsto \{x\}\}$
$x := high\_in$
$\{P = (\exists\, n \cdot z = 2 * n + 1), \Gamma = \{z \mapsto true, x \mapsto false\}, W = \{z := \_ \mapsto \{z\}, x := \_ \mapsto \{x\}\},$
$L = \{z := \_ \mapsto \{x, z\}, x := \_ \mapsto \{x\}\}\}$
...
$x := 0$
$\{P = (\exists\, n \cdot z = 2 * n + 1), \Gamma = \{z \mapsto true, x \mapsto true\}, W = \{z := \_ \mapsto \{z\}, x := \_ \mapsto \{x\}\},$
$L = \{z := \_ \mapsto \{x, z\}, x := \_ \mapsto \{x\}\}\}$
$z := z + 1$

Considering the first assignment to z, we must work through the proof obligations of the AssignG rule. For the information flow test, since $\mathcal{L}(z) = true$, the expression $z + 1$ is trivially *Low*. Also, we can show the value of z is increasing, satisfying the guarantee. Since each c condition and $\mathcal{L}_R$ for **secret_write** are *true*, we can ignore the conditions of valid apart from rising and $\mathsf{falling}$. As we know $\exists n \cdot z = 2 * n$ holds before the instruction and $\exists n \cdot z = 2 * n + 1$ holds after, we can show $\mathcal{L}(x)$ is true before the instruction and $\neg \mathcal{L}(x)$ after. That is, the classification of x rises. However, since $L(z := z + 1) = \{x, z\}$, the associated proof obligation is trivially discharged. We then compute a post state, with the new value of z. Due to the rely condition, we can establish that no other thread will modify z, allowing for the preservation of all information across environment steps.

For the first assignment to x, we only have to consider the information flow test, for reasons outlined above for **write**. Given $W(x := \_) \cap L(x := \_) = \{x\}$ after the first assignment, $P_{\restriction x := high\_in} = true$. Hence, the proof obligation is $true \wedge (\exists n \cdot z = 2 * n) \Rightarrow false$ which cannot be discharged. This is due to the information leak discussed in Section 1 where the first two assignments are reordered. The failure to discharge this proof obligation indicates the leak exists.

To prevent this leak, we could place a fence between the two assignments to prevent the reordering. Such a fence would add z back into $W(x := \_)$ and $L(x := \_)$ resulting in $P_{\restriction x := high\_in} = P$, and hence the proof obligation $(\exists n \cdot z = 2 * n + 1) \wedge (\exists n \cdot z = 2 * n) \Rightarrow false$, which is true due to the contradiction on the left-hand side of the implication.

The assignment to x introduces the entry $x \mapsto false$ in Γ. As all other threads are restricted to writing *Low* values to x, since $\mathcal{L}_R(x) = true$, its Γ mapping is preserved. That is, since it is possibly *High* (when x is not changed by the environment) or *Low* (when x is changed by the environment), we need to record it as potentially holding *High* information.

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

Finally, we have the second assignment to z. Similar to the first assignment, it is trivial to demonstrate the information flow and guarantee checks. However, this assignment will result in the classification of x falling, as it is *High* in the pre-state and *Low* in the post. Therefore, it is necessary to show $(P \Rightarrow \Gamma(x)) \wedge x \in W(z := z + 1)$ which corresponds to $((\exists n \cdot z = 2 * n + 1) \Rightarrow true) \wedge x \in \{z\}$. Although the first conjunct is trivially true, the second (and hence the proof obligation) is obviously false. This failure of the proof obligation indicates the other information leak described in Section 1 when the final two assignments are reordered.

Again, the leak can be prevented by placing a fence between the instructions. This would restore $W(z := z + 1)$ to $\{x, z\}$ making the second conjunct $x \in \{x, z\}$, which is obviously true.

For **secret_read** we apply the AssignG rule to $high\_out := x$. This case is trivial, as the classification of $high\_out$ is *false*, discharging the proof obligation. Similar to the **write** operation, we can ignore the other proof obligations.

The **read** operation introduces the most complexity. It relies on the increasing value of z to detect an interleaving with **secret_write**, in which case it re-attempts the read of x. We introduce the following rule for do loops, which is a composition of Seq and While.

$$\textsc{Do}\ \frac{\begin{array}{ll} \Gamma' \vdash b : t & \vdash \{P, \Gamma, D\}\ c\ \{P', \Gamma', D'\} \\ P'_{\restriction[b]} \Rightarrow t & \vdash (\{P'_{\backslash[b]} \wedge b, \Gamma', D' + [b]\} + R_{[b]})\ c\ \{P', \Gamma', D'\} \end{array}}{\vdash \{P, \Gamma, D\}\ \mathsf{do}\ c\ \mathsf{while}\ (b)\ (\{P'_{\backslash[b]} \wedge \neg b, \Gamma', D' + [b]\} + R_{[b]})}$$

Note that, given $\{P'_{\backslash[b]} \wedge b, \Gamma', D' + [b]\} + R_{[b]} \geq \{P, \Gamma, D\}$ and the first proof over c, $\vdash \{P, \Gamma, D\}\ c\ \{P', \Gamma', D'\}$, it is possible to establish the second proof over c via the Conseq rule. Therefore, given a sufficiently weak initial state for P, Γ and D, only a single loop iteration has to be considered. The initial context for **read** already establishes the weakest possible values for P and Γ, *true* and an empty map respectively, however D is initialised to its strongest value, mapping all instructions to all global variables. To enable analysis of only a single loop iteration, significantly simplifying the proof, we weaken the initial D (using the Conseq rule) to return an empty set for all inputs.

The context between each line is shown below. In this case, we have added the required fences in advance. Again only those functions in D which are relevant to this operation are shown and _ is used to stand for any value. Additionally, unchanged parts of the context at each step are elided. We use the temporary variables v and v' for values of z at each instruction $r_2 := x$ and $[z \neq r_1]$ respectively.

```
{$P = true, \Gamma = \{\}, W = L = I = \{\_ \mapsto \{\}\}$}
do
  do
    $r_1 := z$
    {$P = \{r_1 \leq z\}, \Gamma = \{r_1 \mapsto true\}, ...$}
  while ($r_1\%2 \neq 0$)
  {$P = \{r_1 \leq z \wedge r_1\%2 = 0\}, \Gamma = \{r_1 \mapsto true\}, ...$}
  fence
  {$..., W = L = I = \{\_ \mapsto \{x, z\}\}$}
  $r_2 := x$
  {$P = \{r_1 \leq v \wedge v \leq z \wedge r_1\%2 = 0\}, \Gamma = \{r_1 \mapsto true, r_2 \mapsto \exists n \cdot v = 2 * n\}, I = \{z \neq r_1 \mapsto \{x\}, ...\}, ...$}
  fence
  {$..., W = L = I = \{\_ \mapsto \{x, z\}\}$}
while ($z \neq r_1$)
{$P = \{r_1 \leq v \wedge v \leq v' \wedge v' = r_1 \wedge v' \leq z \wedge r_1\%2 = 0\}, ...$}
{$P = \{r_1 = v \wedge v \leq z \wedge r_1\%2 = 0\}, ...$}
$low\_out := r_2$
```

For the first assignment $r_1 := z$, we apply the AssignL rule, which has no proof obligations. As $\mathcal{L}(z) = true$, the mapping $r_1 \mapsto true$ is introduced to Γ. Also, as the environment steps may increment z, our post state $r_1 = z$ is weakened to $r_1 \leq z$. The components of D are not modified, as the updates for $r_1 := z$ can only merge or remove from the sets. As D maps to only empty sets, they remain empty.

We can then discharge the proof obligation for the inner loop. Based on the introduced mapping for r_1, we have $\Gamma \vdash r_1\%2 \neq 0 : true$. As the inner loop condition does not introduce information regarding a global reference, it is not necessary to remove any existing global variables in P and Γ. Similar to the prior operation, the components of D are not modified, as the updates for the loop condition only merge or remove from the sets.

We then apply the Fence rule restoring W, L and I to their initial values, i.e, mapping each instruction to all global variables. Without the fence, z would not be in $W(r_2 := x) \cap L(r_2 := x)$ and hence (following definition (13)) $\Gamma(x)$ would be $\forall z \cdot \exists n.\ z = 2 * n$ which is equivalent to *false*. That is, r_2 would be regarded as having a *High* value and if the operation reached the final line without updating r_2 again, the information flow check on the final line would fail. This failure indicates an information leak which occurs due to the instruction $r_2 := x$ being reordered before the preceding assignment to r_1 and hence potentially being executed when z is odd. The fence prevents such reordering.

Next, we apply AssignL to $r_2 := x$, which has no proof obligations. As a result $\Gamma(r_2)$ is updated to $(\exists n.\ z = 2 * n)$, the security policy for x, to represent the information flow. Due to possible modifications of z due to the environment, a temporary variable v is introduced to capture to the value of z during this operation. Existing references to z are replaced with v, including the new reference in $\Gamma(r_2)$, and its relation with z is introduced to P. This modification to the context introduces an indirect reference to z, resulting in z being removed from $I(\alpha)$ for all instructions that can reorder prior to $r_2 := x$. Significantly, this includes the outer loop's condition: $z \neq r_1$. Without a fence after $r_2 := x$, this would result in existential quantification of any references to z in the context before the conjunction of the outer loop's exit condition (see the Do rule and (10)). Consequently, it would not be possible to relate the value of z when executing $r_2 := x$ and $z \neq r_1$. In such a scenario, it would not be possible to establish $\Gamma(r_2) = true$ at the exit of the **read** operation, which is essential to successfully validate the operation. This captures the information leak that is possible when $r_2 := x$ executes after $z \neq r_1$ and z is odd. Instead, the additional fence allows for the application of the Fence rule, which restores I to its initial value.

We then consider the proof obligations of the outer loop's condition. Proving $\Gamma \vdash z \neq r_1 : true$ is trivial as $\Gamma(r_1) = true$ has been retained across prior instructions. When computing the postcondition on exit from the outer loop, we introduce a new temporary variable v' to model z, and relate its value to v, the prior value of z, via $v \leq v'$. The context can then be used to deduce that $r_1 = v$, linking the exit condition for the inner loop, $r_1\%2 = 0$, to the security policy for the read of x in $\Gamma(r_2)$. Specifically, we are able to show that $\Gamma(r_2)$ is *true*. This is used to discharge the information flow proof obligation when we apply AssignG to the final instruction. As *low_out* is not referenced in any security policies or rely/guarantee specifications, the remaining proof obligations are trivial.

4.5. Incompleteness

While sound (see Section 6), the logic is incomplete as it is too complex to consider all possible execution traces enabled by weak memory behaviour. For instance, the logic makes use of existential variable quantification in equations (8) and (10) to remove references to previous writes with which the current instruction may reorder. This technique over-approximates the values of the variable to avoid precisely tracking all possibilities. This can be seen in the snippet $x := 1;\ \mathsf{fence};\ x := 2;\ y := 1$. Realistically, the value of x when executing $y := 1$ should be 1 or 2, depending on its reordering with $x := 2$. However, the analysis will remove all information regarding x from the state when considering the instruction. Consequently, it would not be possible to establish properties such as $x > 0$. A similar issue arises due to the uses of universal quantification in the context update definitions (13) and (14).

Additionally, the logic only supports rely/guarantee specifications of the form seen in R_{var} and G_{var} to facilitate automation (see Section 7.1). Based on the typical examples of non-blocking algorithms in [37], we believe this is sufficient for handling the synchronisation and control variable modifications typically observed with information flow in such algorithms. Nevertheless, there may be programs whose rely and guarantee conditions cannot be expressed in this form.

5. Compositional Noninterference

In this section, we provide the theoretical underpinnings of our information flow logic which have been encoded in Isabelle/HOL, along with the logic, in order to prove the logic sound. Our compositional technique for checking non-interference extends prior work [15, 20] to enable a wider variety of security policies in a self-contained logic. The definitions and theorems presented here can be considered a specialisation of the rely/guarantee parallel rule [14], describing how to appropriately compose the analysis of individual threads.

We first define our language semantics, capturing weak memory behaviour via a reordering relation. Following this, we introduce our notion of low equivalence and enforce its preservation via rely/guarantee specifications. We then establish the properties necessary to express a compositional bisimulation and define the security property it enforces. Moreover, we detail some of the wellformedness properties preserved throughout the analysis, mostly required to avoid environmental interference. Finally, we comment on other work related to this compositional approach.

5.1. Thread Semantics

The thread semantics is similar to a standard semantics for a while language, however, with the introduction of a reordering rule based on the techniques introduced by Colvin et al. [6]. We first define our small step semantics over a simple language supporting sequential composition ; , non-deterministic choice $\sqcap$ and iteration *. The semantics is parameterised by the executed instruction or a silent instruction τ in the event of internal steps. We assume a suitable reordering relation and forwarding function are provided for the memory model. Note that instructions are only executed through in-order evaluation, via the sequential rule, or out-of-order, via the reordering rule.

$$\begin{array}{ll} \alpha;\ c \rightarrow_{\alpha} c & \alpha;\ c \rightarrow_{\beta_{\langle\alpha\rangle}} \alpha;\ c' \quad \text{if } \alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle} \wedge c \rightarrow_{\beta} c' \\ c_1 \sqcap c_2 \rightarrow_{\tau} c_1 & c_1 \sqcap c_2 \rightarrow_{\tau} c_2 \\ c^* \rightarrow_{\tau} c;\ c^* & c^* \rightarrow_{\tau} \mathsf{skip} \end{array} \tag{31}$$

Let $\langle c, mem\rangle$ denote the local configuration of a thread, where c is a thread's command and mem encodes memory as a mapping from variables to values. We further assume a partial function exec defining the deterministic semantics for instructions. From this, we extend the small step semantics to operate over thread configurations.

$$\langle c, mem\rangle \rightarrow_{\alpha} \langle c', mem'\rangle \quad \text{if } c \rightarrow_{\alpha} c' \wedge \mathsf{exec}\ mem\ \alpha = mem' \tag{32}$$

We map the language defined in Section 3.1 to the introduced operators, such that $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ is treated as $([b];\ c_1) \sqcap ([\neg\ b];\ c_2)$, where $[b]$ is a guard with condition b. Moreover, $\mathsf{while}\ b\ \mathsf{do}\ c$ is represented as $([b];\ c)^*;\ [\neg\ b]$. All other language constructs are trivially mapped.

5.2. Program Semantics

As we assume a multi-copy-atomic memory model, we are able to represent global behaviour based on the standard notion of interleaving [6]. Let $(comp, mem_G)$ denote the global configuration of a program, where mem_G encodes the memory for only *Global* variables and *comp* consists of a list of threads. We assume a fixed set of threads throughout execution. These threads are represented as a tuple, (c, mem_L), such that c is the thread's command and mem_L encodes the memory for *Local* variables. We overload $\rightarrow_\alpha$ to denote a transition over global configurations, where one thread evaluates the instruction α.

$$\frac{comp[i] = (c, mem_L) \qquad \langle c, mem_G + mem_L\rangle \rightarrow_\alpha \langle c', mem'_G + mem'_L\rangle}{(comp, mem_G) \rightarrow_\alpha (comp[i \mapsto (c', mem'_L)], mem'_G)} \tag{33}$$

where $m_1 + m_2$ represents the merging of two mappings, under the assumption that their domains do not overlap, $l[i]$ denotes the element at index i of list l, and $l[i \mapsto e]$ denotes updating the ith position of l to value e. The structure of our global transitions provides each thread with a local memory and clearly enforces their separation.

We extend this notion to capture the possible execution traces of the program, encoded as a list of instructions.

$$\frac{}{(comp, mem_G) \longrightarrow_{[]} (comp, mem_G)} \tag{32.1}$$

$$\frac{(comp, mem_G) \rightarrow_\alpha (comp', mem'_G) \qquad (comp', mem'_G) \longrightarrow_t (comp'', mem''_G)}{(comp, mem_G) \longrightarrow_{\alpha\#t} (comp'', mem''_G)} \tag{32.2}$$

where # as list concatenation.

5.3. Low Equivalence

Noninterference is commonly established via *low equivalence* between two program configurations under a bisimulation. This property enforces equivalence between the two configurations for all variables classified as *Low*. Consequently, it is not possible to distinguish the two program configurations via inspection of only *Low* variables. A program satisfies noninterference when any two low-equivalent configurations remain low-equivalent when undergoing the same transitions. That is, the values of their *Low* variables are not influenced by those of their *High* variables.

We introduce the following definition of low equivalence between two memories, given a security policy l for a set of variables V. As we assume a two level security lattice, where *true* corresponds to *Low* and *false* to *High*, we encode the security policy as a mapping from variables to predicates, such that a variable is considered *Low* for a given state if the state satisfies the predicate.

$$mem_1 =^{l,V} mem_2 \;\; \equiv \;\; \forall\, x \in V \cdot \mathsf{eval}\; mem_1\; (l\; x) \vee \mathsf{eval}\; mem_2\; (l\; x) \Rightarrow mem_1\; x = mem_2\; x \tag{35}$$

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

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

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

5.4. Rely/Guarantee Relations

To achieve a simple compositional approach based on rely/guarantee reasoning, we introduce combined rely and guarantee relations that encapsulate both the security policy definitions, $\mathcal{L}_R$ and $\mathcal{L}_G$, and the state relations R and G. Note it would be possible to use $\mathcal{L}_R$, $\mathcal{L}_G$, R and G rather than these combined relations, which we refer to as $\mathcal{R}$ and $\mathcal{G}$, throughout the following definitions. However, merging them simplifies definitions and corresponding proofs as it assists in the reuse of standard rely/guarantee properties and theorems.

As $\mathcal{R}$ and $\mathcal{G}$ are intended to include both bisimulation and behavioural properties, it is necessary to encode them as relations over pairs of bisimilar memories. To illustrate, given $((mem_1, mem_2), (mem_1', mem_2')) \in \mathcal{G}$, the thread would transition mem_1 to mem_1' and mem_2 to mem_2' while potentially establishing bisimulation properties between mem_1' and mem_2' given suitable properties between mem_1 and mem_2.

$$\begin{aligned}\mathcal{R} \;\equiv\; \{((mem_1, mem_2), (mem_1', mem_2')) \mid (mem_1, mem_1') \in R \wedge (mem_2, mem_2') \in R \wedge \\ mem_1' =^{\mathcal{L}} mem_2' \wedge mem_1' =^{\mathcal{L}_R, V_d} mem_2'\}\end{aligned} \tag{36}$$

$$\begin{aligned}\mathcal{G} \;\equiv\; \{((mem_1, mem_2), (mem_1', mem_2')) \mid (mem_1, mem_1') \in G \wedge (mem_2, mem_2') \in G \wedge \\ mem_1' =^{\mathcal{L}} mem_2' \wedge mem_1' =^{\mathcal{L}_G, V_d} mem_2'\}\end{aligned} \tag{37}$$

where V_d is the set of all variables which have been modified in either the transition from mem_1 to mem_1' or from mem_2 to mem_2'.

These definitions first enforce the thread's R and G relations across all transitions, ensuring rely/guarantee reasoning is fully supported by the logic. This is crucial to establishing a self-contained analysis capable of capturing a variety of synchronisation behaviours, in contrast to other approaches where annotations or locks are employed.

Two low equivalence properties are required to constrain information flow. The first, $mem_1' =^{\mathcal{L}} mem_2$, ensures both the thread and the environment preserve the global security policy $\mathcal{L}$ across all variables. Consequently, a local analysis can always assume this security property holds at a minimum. The remaining property enforces the more specific thread-level security policy, $\mathcal{L}_R$ or $\mathcal{L}_G$, restricted to only those variables that have been modified, V_d.

5.5. Compositional Bisimulation

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

$$\begin{aligned}\mathsf{bisim}\; \mathcal{B}\; \mathcal{G} \;\equiv\; & \mathsf{sym}\; \mathcal{B} \wedge \forall\, c_1, mem_1, c_2, mem_2, c_1', mem_1' \cdot \\ & \quad (c_1, mem_1)\; \mathcal{B}\; (c_2, mem_2) \Rightarrow \\ & \quad\quad \langle c_1, mem_1\rangle \rightarrow \langle c_1', mem_1'\rangle \Rightarrow \\ & \quad\quad\quad (\exists\, c_2', mem_2' \cdot \langle c_2, mem_2\rangle \rightarrow \langle c_2', mem_2'\rangle \wedge \\ & \quad\quad\quad\quad ((mem_1, mem_2), (mem_1', mem_2')) \in \mathcal{G} \wedge \\ & \quad\quad\quad\quad (c_1', mem_1')\; \mathcal{B}\; (c_2', mem_2'))\end{aligned} \tag{38}$$

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

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

$$\begin{aligned}\mathsf{stable}\; \mathcal{B}\; \mathcal{R} \;\equiv\; & \forall\, c_1, mem_1, mem_1', c_2, mem_2\; mem_2' \cdot \\ & \quad (c_1, mem_1)\; \mathcal{B}\; (c_2, mem_2) \Rightarrow \\ & \quad\quad ((mem_1, mem_2), (mem_1', mem_2')) \in \mathcal{R} \Rightarrow \\ & \quad\quad\quad (c_1, mem_1')\; \mathcal{B}\; (c_2, mem_2')\end{aligned} \tag{39}$$

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

$$\begin{aligned}\mathsf{secure}\; c\; P\; \mathcal{R}\; \mathcal{G} \;\equiv\; & \forall (mem_1, mem_2) \in P \cdot \exists \mathcal{B} \cdot \\ & \quad \mathsf{bisim}\; \mathcal{B}\; \mathcal{G} \wedge \mathsf{stable}\; \mathcal{B}\; \mathcal{R} \wedge (c, mem_1)\; \mathcal{B}\; (c, mem_2)\end{aligned} \tag{40}$$

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

$$\mathsf{compat}\ \mathcal{R}_s\ \mathcal{G}_s \quad \equiv \quad \forall\, i < |\mathcal{R}_s| \cdot\ \forall\, j < |\mathcal{G}_s| \cdot\ i \neq j \Rightarrow \mathcal{G}_s[j] \subseteq \mathcal{R}_s[i] \tag{41}$$

where the notation $\mathcal{R}_s$ and $\mathcal{G}_s$ refer to lists of $\mathcal{R}$ and $\mathcal{G}$ respectively, such that $\mathcal{R}_s[i]$ refers to the rely relation for the ith thread. This definition of compatibility is standard for rely/guarantee reasoning.

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

$$\begin{aligned}\mathsf{secure_G}\ comp \quad \equiv \quad & \forall\, mem_1, mem_2 \cdot mem_1 =^{\mathcal{L}} mem_2 \Rightarrow \\ & \quad \forall\, t, comp', mem'_1 \cdot (comp, mem_1) \longrightarrow_t (comp', mem'_1) \Rightarrow \\ & \quad\quad (\exists\, mem'_2 \cdot (comp, mem_2) \longrightarrow_t (comp', mem'_2) \wedge \\ & \quad\quad\quad mem'_1 =^{\mathcal{L}} mem'_2)\end{aligned} \tag{42}$$

Enforcing equivalent execution traces between the two configurations ensures that both evaluations observe the same reordering and scheduling decisions. We believe this is a suitable property, as we assume such decisions are not influenced by *High* information. Relating executions with different reordering and scheduling decisions is significantly more complex and likely requires a probabilistic technique [32].

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

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

Theorem 1 can be established using a similar approach to establishing the rely/guarantee parallel rule. We show the global property holds by defining a global bisimulation in terms of the local bisimulations provided in the premisses and inducting on the trace t in $\mathsf{secure_G}$. The base case of an empty trace is trivial, as the global security policy is known to hold initially. For the inductive case, a thread is selected and executed. It is necessary to establish the global security policy on the resulting memory state, as required by $\mathsf{secure_G}$, and re-establish the secure properties for all threads.

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

Consequently, a local analysis capable of establishing secure $comp[i]\ P\ \mathcal{R}_s[i]\ \mathcal{G}_s[i]$ for all threads in the system and corresponding proofs of compatibility can demonstrate a global security property. Note that this security property holds for an arbitrary multi-copy-atomic weak memory model that can be represented via a reordering relation and forwarding function.

5.6. Wellformedness

We introduce a series of wellformedness properties for the logic context enforcing stability from the perspective of the rely specification R and $\mathcal{L}_R$. These allow us to establish the stable property defined in Section 5.5, and hence that environment steps cannot invalidate the context or the local logic's judgements. First, we require stability on our state predicate P with respect to R, in the traditional sense from rely/guarantee reasoning.

$$\mathsf{stable\_P}\ P \quad \equiv \quad \forall\, mem\ mem' \cdot \mathsf{eval}\ mem\ P \wedge (mem, mem') \in R \Rightarrow \mathsf{eval}\ mem'\ P \tag{43}$$

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

$$\begin{aligned}\mathsf{stable\_}\Gamma\; P\; \Gamma \equiv \forall\, mem\; mem' \cdot \mathsf{eval}\; mem\; P \wedge (mem, mem') \in R \Rightarrow \\ \forall\, x \in \mathsf{dom}\;\Gamma \cdot \mathsf{eval}\; mem\; \Gamma(x) = \mathsf{eval}\; mem'\; \Gamma(x)\end{aligned} \tag{44}$$

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

$$\mathsf{context\_wf}\; P\; \Gamma \;\;\equiv\;\; \mathsf{stable\_}\Gamma\; P\; \Gamma \wedge \mathsf{stable\_P}\; P \wedge \mathsf{dom}\;\Gamma \subseteq \mathsf{low\_or\_eq}\; P \tag{45}$$

5.7. Related approaches

The RGSim framework [33] defines a similar compositional theory to the one detailed in this paper, merging a simulation definition with rely/guarantee relations over pairs of states. Consequently, the authors are able to demonstrate the preservation of properties across program transformations. This technique has been extended by Murray et al. [20] to demonstrate the preservation of noninterference properties across refinement.

Gordon et al. [34] propose a similar approach for statically associating rely/guarantee conditions to variables, in RGRef. This work couples rely/guarantee conditions with references, encoding rely/guarantee relations over their objects, which are then enforced via a type system. Such an approach provides a potential alternative to the rely/guarantee conditions we introduce.

6. Soundness

The compositional theory of the previous section and our logic's rules have been encoded in Isabelle/HOL, building off prior work from [15] and [20]. The encoding and proof of soundness total ~3K lines. The full encoding is available at `https://bitbucket.org/wmmif/wmm-rg-if/src/master`.

We have shown the logic establishes a local bisimulation, which satisfies the constraints seen in secure. The bisimulation enforces a local security policy $\mathcal{L}_\Gamma \equiv \lambda\, x \cdot \mathcal{L}(x) \vee \Gamma\langle x\rangle$, such that *x* is considered *Low* if it is required by the global security policy or can be shown locally via Γ.

$$\begin{aligned}(c, mem_1)\, \mathcal{B}\, (c, mem_2) \;\;\equiv\;\; & \exists\, P\, \Gamma\, D\, P'\, \Gamma'\, D' \cdot \vdash \{P, \Gamma, D\}\; c\; \{P', \Gamma', D'\} \wedge \mathsf{context\_wf}\; P\; \Gamma\; \wedge \\ & \mathsf{eval}\; mem_1\; P \wedge \mathsf{eval}\; mem_2\; P \wedge mem_1 =^{\mathcal{L}_\Gamma} mem_2\end{aligned} \tag{46}$$

Note that the bisimulation only holds for configurations with the same *c*, as constraints on control flow and thread scheduling ensure that these remain the same throughout execution.

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

- The relation must be symmetric.
- The relation must be stable given the thread's rely $\mathcal{R}$, as defined in (39).
- The relation must be a valid bisimulation that conforms to the thread's guarantee $\mathcal{G}$, as defined in (38).

We will focus on each of these cases individually.

6.1. Symmetry

The simplest property is symmetry. Given $(c, mem_1)\, \mathcal{B}\, (c, mem_2)$ and the definition of the bisimulation above, it is trivial to establish the necessary properties for $(c, mem_2)\, \mathcal{B}\, (c, mem_1)$. Only the proof of $mem_1 =^{\mathcal{L}_\Gamma} mem_2 \Rightarrow mem_2 =^{\mathcal{L}_\Gamma} mem_1$ introduces some complexity, however, this is obviously the case due to the symmetry of low equivalence (35).

6.2. Stability

Next, we consider stability. Expanding (39), we demonstrate $(c, mem'_1)\ \mathcal{B}\ (c, mem'_2)$ given $(c, mem_1)\ \mathcal{B}\ (c, mem_2)$ for some mem'_1 and mem'_2 such that $((mem_1, mem_2), (mem'_1, mem'_2)) \in \mathcal{R}$. The first two properties of the relation are preserved across this memory change as they are not dependent on mem_1 or mem_2. Due to context_wf, the predicate P is known to be stable in R. Therefore, it is straightforward to establish eval mem'_1 P and eval mem'_2 P, based on the definition of stability.

To demonstrate the low equivalence property $mem'_1 =^{\mathcal{L}_\Gamma} mem'_2$, where $\mathcal{L}_\Gamma \equiv \lambda x \cdot \mathcal{L}(x) \vee \Gamma\langle x\rangle$, we first unfold the definitions of low equivalence. This simplifies to a proof of $mem'_1\ x = mem'_2\ x$ given $\mathcal{L}_\Gamma(x)$ for any x. Eliminating the disjunction in $\mathcal{L}_\Gamma$, we first consider the case where $\mathcal{L}(x)$ is true. Therefore, the variable x is known to be *Low* via the global security policy $\mathcal{L}$, which $\mathcal{R}$ enforces on the primed memories. Consequently, we can establish equivalence between the two primed memories for x.

In the alternative case, $\Gamma\langle x\rangle$ is true. If x is not in the domain of Γ, then $\Gamma\langle x\rangle = \mathcal{L}(x)$ and the prior case proof holds. Otherwise, x is in the domain of Γ and, therefore, in the set low_or_eq P based on the wellformedness property. Moreover, as we known $\Gamma(x)$ and $mem_1 =^{\mathcal{L}_\Gamma} mem_2$, we can show $mem_1\ x = mem_2\ x$. Therefore, either the environment does not modify x, in which case $mem_1\ x = mem_2\ x \Rightarrow mem'_1\ x = mem'_2\ x$, or the environment guarantees to only write *Low* information to x, which can be rephrased as $mem'_1\ x = mem'_2\ x$.

As a result, all properties necessary to establish $(c, mem'_1)\ \mathcal{B}\ (c, mem'_2)$ can be shown and the relation $\mathcal{B}$ can be considered stable under $\mathcal{R}$.

6.3. Bisimulation

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

To demonstrate this property, we consider the two possible types of steps supported by the semantics in Section 5.1: silent steps and instruction steps. Silent steps are trivially handled, as the memory and context are not changed by the transition. Moreover, due to equivalence over the program's commands in both configurations, it is obvious that both configurations can perform the same silent step.

In the event of an instruction step, we demonstrate that given the execution of an instruction and a judgement over the source command there exists a judgement over the executed instruction followed by a judgement over the remaining command.

$$\frac{c \rightarrow_\alpha c' \qquad \vdash \{P, \Gamma, D\}\ c\ \{P', \Gamma', D'\}}{\vdash \{P, \Gamma, D\}\ \alpha\ \{P_i, \Gamma_i, D_i\}\ \wedge\ \vdash \{P_i, \Gamma_i, D + [\alpha]_f\}\ c'\ \{P', \Gamma', D'\}} \tag{47}$$

where $D + [\alpha]_f$ represents an update applied to each of D's functions where α is assumed to be ordered with all other functions, in effect ignoring the reordering relation. In contrast to $D + [\alpha]$, this encodes the idea that α is being evaluated and, therefore, instructions in c' are known to evaluate after α.

Notably, this property holds regardless of the reordering required to evaluate α given c. To demonstrate this, we induct over the small step semantics capable of producing an instruction step. The base case is the sequential rule while the inductive case is the reordering rule, due to its dependence on an existing step. The property trivially holds in the base case, as a judgment over $\alpha;\ c$ can be split into judgements over α and c. Moreover, $D + [\alpha]_f$ can be shown to be stronger than D_i, establishing the desired outcome via the Conseq rule.

For the induction step, we must demonstrate the ability to reorder judgements over instructions given they may be reorderable under the weak memory model.

$$\frac{\beta \overset{R}{\Leftarrow} \alpha_{\langle\beta\rangle} \qquad \vdash \{P, \Gamma, D\}\,\beta\,\{P', \Gamma', D'\} \qquad \vdash \{P', \Gamma', D'\}\ \alpha\ \{P'', \Gamma'', D''\}}{\vdash \{P, \Gamma, D\}\ \alpha_{\langle\beta\rangle}\ \{P_i, \Gamma_i, D_i\}\ \wedge\ \vdash \{P_i, \Gamma_i, D + [\alpha]_f\}\,\beta\,\{P'', \Gamma'', D' + [\alpha]_f\}} \tag{48}$$

Demonstrating $\vdash \{P, \Gamma, D\}\ \alpha_{\langle\beta\rangle}\ \{P_i, \Gamma_i, D_i\}$ makes use of D to show that the proof obligations of α still hold under a context where β has not executed, as W and L have the effect of hiding any information derived from β. Demonstrating $\vdash \{P_i, \Gamma_i, D + [\alpha]_f\}\,\beta\,\{P'', \Gamma'', D' + [\alpha]_f\}$ makes use of the valid proof obligation on α, to show that the early execution of α will not invalidate any program-order earlier proof obligations, such as those for β. This motivates the use of

$D + [\alpha]_f$, as the restrictions enforced by D_i may hide information available to α from β, whereas $D + [\alpha]_f$ ensures β is aware of all instructions that evaluate earlier than α.

Given 47 holds, we obtain a judgement over the executed instruction, from which we:

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

The first properties are straightforward, as the strongest post-condition operation follows the standard approach and the application of $+R$ has been detailed in Section 4.2. The proof obligations for the AssignG and AssignL rules provide sufficient information to demonstrate the remaining properties. For example, the classification comparisons ensure a *High* expression is never written to a *Low* variable. Moreover, a *Low* expression must have the same result given either configuration. Therefore, if the written variable is *Low*, the written value must be equal between the resultant memories. This corresponds to item 4 in the list, as well as trivial cases for item 3. To fully demonstrate item 3, it is necessary to consider cases of a classification change, which the constraint on $\mathsf{falling}$ addresses (21). Additionally, the guar proof obligation (19) directly demonstrates item 5. Given the above properties are shown, the bisimulation can be re-established on the new memories and the transition can be shown to conform to $\mathcal{G}$.

6.4. Composition

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

$$\{P_0, \Gamma_0, D_0\}\ c\ \{P, \Gamma, D\} \Rightarrow (c, mem)\ \mathcal{B}\ (c, mem)$$

where $P_0 \equiv \mathit{true}$ and Γ_0 is an empty map.

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

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

$$\frac{\begin{array}{c}\mathsf{compat}\ \mathcal{R}_s\ \mathcal{G}_s \\ \forall\, i < |comp| \cdot\ \ \vdash \{P_0, \Gamma_0, D_0\}\ comp[i]\ \{P_i, \Gamma_i, D_i\}\ \text{for}\ \mathcal{R}_s[i], \mathcal{G}_s[i]\end{array}}{\mathsf{secure_G}\ comp}$$

7. Automation

We have implemented a prototype symbolic execution tool to automate the application of our logic for programs running on the ARMv8 memory model. The tool was based on that described in [26] utilising Scala, to take advantage of Scalas powerful pattern matching and compatibility with Java through the JVM, and the SMT solver Z3 [35], through the Z3 Java API, in order to reason about predicates in the program state and determine if the security properties described in rules of the logic hold.

Using the Conseq rule requires user intervention and is hence not amenable to automation. Hence, we implemented specialisations of the If and While rules which incorporate an application of the Conseq rule to provide a suitable post-state automatically. For the If rule, this post-state is the strongest state that is weaker than each of the post-states resulting from the instructions branches, as illustrated by the AutoIf rule, where $D_1 \cap D_2$ returns the intersection of the corresponding components of D_1 and D_2.

$$\textsc{AutoIf}\ \frac{\Gamma \vdash b : t \qquad \vdash (\{P_{\setminus[b]} \wedge b, \Gamma, D + [b]\} + R_{[b]})\ c_1\ \{P_1, \Gamma_1, D_1\} \\ P_{\restriction[b]} \Rightarrow t \qquad \vdash (\{(P_{\setminus[b]} \wedge \neg\, b, \Gamma, D + [b]\} + R_{[b]})\ c_2\ \{P_2, \Gamma_2, D_2\}}{\vdash \{P, \Gamma, D\}\ \mathsf{if}\ (b)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{P_1 \vee P_2, \lambda x \cdot \Gamma_1(x) \vee \Gamma_2(x), D_1 \cap D_2\}}$$

The specialised While rule requires the user to provide an invariant, in advance of running the tool, by providing suitable values for Γ and P. The tool is able to compute D based on a data flow analysis. Note that the symbolic execution tool need only consider one iteration of the loop to validate the invariant, avoiding significant analysis cost.

While type system approaches to establishing noninterference benefit from a high degree of automation, context-aware, value-dependent variants do not provide for trivial implementations due to the necessity in establishing and maintaining a predicate P throughout the analysis. Below we outline simplifications that have been employed in our tool to account for this complexity. The prototype tool is available at `https://github.com/l-kent/wemelt/tree/rgweakmemory`.

7.1. Restrictions on Variables

A significant barrier to automation is the environment step function $+R_\alpha$, due to increases in P's size and complexity in the form of temporary variables and addition of R. It is for this reason that we structure the relational predicate R by introducing the set R_{var}. This structure enables reasoning about the effects of the environment at an individual variable level. Therefore, we are able to modify $+R_\alpha$ to only introduce temporary variables and expand P on a variable-by-variable basis. Moreover, $+R_\alpha$ enforces stability given any context, disregarding the fact that the pre-context was stable. Therefore, by inspecting the changes to the context throughout the logic's rules, it is possible to reduce the overhead of $+R_\alpha$.

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

$$\begin{aligned} \mathsf{new\_var}\ b &\equiv \mathsf{vars}\ b \\ \mathsf{new\_var}\ (x := e) &\equiv \{x\} \cup \mathsf{vars}\ e \cup \bigcup_{y \in \mathsf{vars}\ e \setminus \mathsf{dom}\ \Gamma} \mathsf{vars}\ (\mathcal{L}(y)) \end{aligned} \tag{49}$$

In addition to introducing new variable references, assignments may reduce the constraints on the environment by negating the conditional c portions of R_{var} properties. We can determine this set of variables by comparing the conditions before and after the assignment, such that, given c holds prior to the assignment, it should hold after.

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

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

$$\mathsf{equal}\ P \equiv \{y \mid \exists\, c \cdot (y, c, \mathcal{I}) \in R_{var} \wedge P \Rightarrow c\} \tag{50}$$

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

Given these definitions, we restrict the domain of renaming in $P + R_\alpha$ to those *Global* variables in $\mathsf{new\_var}$ and weaker, but not equal. Moreover, we restrict the expansion of P to only those conditional relations that are required. Let $\{x_1, ..., x_n\} = (\mathsf{new\_var}(\alpha) \cup \mathsf{weaker}\ P\ \alpha) - \mathsf{equal}\ P$ if α is an assignment, and $\{x_1, ..., x_n\} = \mathsf{new\_var}(\alpha) - \mathsf{equal}\ P$ otherwise.

$$P + R_\alpha \equiv (P \wedge \bigwedge_{(x,c,r) \in R'_{var}} (\forall\, y_1, \ldots, y_m \cdot c) \Rightarrow r(x, x'))[v_1, ..., v_n, x_1, .., x_n / x_1, ..., x_n, x'_1, ..., x'_n] \tag{51}$$

where $\{y_1, ..., y_m\} = vars(P) - (W(\alpha) \cap L(\alpha))$ and $R'_{var} \equiv \{(x, c, r) \mid (x, c, r) \in R_{var} \wedge x \in \{x_1, .., x_n\}\}$.

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

7.2. Rely Invariants

We introduce support for a rely invariant R_{inv} which is maintained throughout execution. Rather than introducing this to P at the application of $+R_\alpha$, which could result in a dramatic increase to the predicate size, we modify all applications of $P \Rightarrow Q$ and $P \wedge t' \Rightarrow t$ to consider $P \wedge R_{inv}$. As this optimisation fails to preserve restrictions due to the invariant on variable definitions prior to assignments, it may result in their rejection where they would have been accepted by the general logic. In these cases, R_{inv} could be injected into P via the Conseq rule. Investigating how to do this automatically has not yet been explored.

8. Conclusion

In this paper, we have presented the first compositional information flow logic for concurrent programs that is (a) capable of handling complex interactions between threads that are not restricted to locking, and (b) supports detection of information flow problems on a wide range of hardware weak memory models. The logic has been encoded and proved sound in the Isabelle/HOL theorem prover, and optimisations introduced in the paper have been used to automate the application of the logic in a symbolic execution tool.

The results of this work build on two streams of related work. Firstly, the use of rely/guarantee reasoning in information flow logics to provide a general approach to composing threads with complex interactions, and secondly the use of reordering relations to define a hardware-agnostic approach to reason about program behaviour on weak memory models.

What is evident from this work is the complexity of checking information flow in concurrent programs. In order to capture the effects of weak memory in our logic while allowing for complex inter-thread interactions requires four distinct functions mapping instructions to sets of global variables. Each of these functions needs to be updated as each instruction of a thread is analysed. While the approach has been automated using symbolic execution, the inherent complexity limits the size of the programs that our tool can handle. In recent work [36], we have presented a general approach for separating the analysis of weak memory effects from rely/guarantee reasoning on concurrent programs. A future direction aimed at a more efficient and hence scalable analysis tool, is to combine this approach with a simpler information flow logic supporting rely/guarantee reasoning but not weak memory models (e.g., [18, 37]).

Acknowledgements Thanks to Kirsten Winter for many insightful discussion on this work, and to Liam Kent for implementing the symbolic execution tool. This work was supported by Australian Research Council Discovery Grant DP160102457.

[1] D. J. Sorin, M. D. Hill, D. A. Wood, A Primer on Memory Consistency and Cache Coherence, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2011. `doi:10.2200/S00346ED1V01Y201104CAC016`.
URL `https://doi.org/10.2200/S00346ED1V01Y201104CAC016`

[2] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, M. O. Myreen, x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors, Commun. ACM 53 (7) (2010) 89–97. `doi:10.1145/1785414.1785443`.
URL `http://doi.acm.org/10.1145/1785414.1785443`

[3] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, P. Sewell, Modelling the ARMv8 architecture, operationally: Concurrency and ISA, in: R. Bodík, R. Majumdar (Eds.), Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, ACM, 2016, pp. 608–621. `doi:10.1145/2837614.2837615`.
URL `http://doi.acm.org/10.1145/2837614.2837615`

[4] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, P. Sewell, Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8, PACMPL 2 (POPL) (2018) 19:1–19:29. `doi:10.1145/3158107`.
URL `http://doi.acm.org/10.1145/3158107`

[5] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, D. Williams, Understanding POWER multiprocessors, in: M. W. Hall, D. A. Padua (Eds.), Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, ACM, 2011, pp. 175–186. `doi:10.1145/1993498.1993520`.
URL `http://doi.acm.org/10.1145/1993498.1993520`

[6] R. J. Colvin, G. Smith, A wide-spectrum language for verification of programs on weak memory models, in: K. Havelund, J. Peleska, B. Roscoe, E. P. de Vink (Eds.), Formal Methods - 22nd International Symposium, FM 2018, Vol. 10951 of Lecture Notes in Computer Science, Springer, 2018, pp. 240–257. `doi:10.1007/978-3-319-95582-7\_14`.
URL `https://doi.org/10.1007/978-3-319-95582-7_14`

[7] R. J. Colvin, G. Smith, A high-level operational semantics for hardware weak memory models, CoRR abs/1812.00996.

[8] M. Moir, N. Shavit, Concurrent data structures, in: D. P. Mehta, S. Sahni (Eds.), Handbook of Data Structures and Applications., Chapman and Hall/CRC, 2004. `doi:10.1201/9781420035179.ch47`.
URL `https://doi.org/10.1201/9781420035179.ch47`

[9] M. F. Atig, A. Bouajjani, S. Burckhardt, M. Musuvathi, On the verification problem for weak memory models, in: M. V. Hermenegildo, J. Palsberg (Eds.), Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, ACM, 2010, pp. 7–18. `doi:10.1145/1706299.1706303`.
URL `http://doi.acm.org/10.1145/1706299.1706303`

[10] J. A. Vaughan, T. D. Millstein, Secure information flow for concurrent programs under Total Store Order, in: S. Chong (Ed.), 25th IEEE Computer Security Foundations Symposium, CSF 2012, IEEE Computer Society, 2012, pp. 19–29. `doi:10.1109/CSF.2012.20`.
URL `https://doi.org/10.1109/CSF.2012.20`

[11] H. Mantel, M. Perner, J. Sauer, Noninterference under weak memory models, in: IEEE 27th Computer Security Foundations Symposium, CSF 2014, IEEE Computer Society, 2014, pp. 80–94. `doi:10.1109/CSF.2014.14`.
URL `https://doi.org/10.1109/CSF.2014.14`

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

[13] J. A. Goguen, J. Meseguer, Security policies and security models, in: 1982 IEEE Symposium on Security and Privacy, 1982, IEEE Computer Society, 1982, pp. 11–20. `doi:10.1109/SP.1982.10014`.
URL `https://doi.org/10.1109/SP.1982.10014`

[14] C. B. Jones, Specification and design of (parallel) programs, in: R. E. A. Mason (Ed.), Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, North-Holland/IFIP, 1983, pp. 321–332.

[15] H. Mantel, D. Sands, H. Sudbrock, Assumptions and guarantees for compositional noninterference, in: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, IEEE Computer Society, 2011, pp. 218–232. `doi:10.1109/CSF.2011.22`.
URL `https://doi.org/10.1109/CSF.2011.22`

[16] L. Lourenço, L. Caires, Dependent information flow types, in: S. K. Rajamani, D. Walker (Eds.), Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, ACM, 2015, pp. 317–328. `doi:10.1145/2676726.2676994`.
URL `https://doi.org/10.1145/2676726.2676994`

[17] T. C. Murray, Short paper: On high-assurance information-flow-secure programming languages, in: M. Clarkson, L. Jia (Eds.), Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2015, ACM, 2015, pp. 43–48. `doi:10.1145/2786558.2786561`.
URL `https://doi.org/10.1145/2786558.2786561`

[18] N. Coughlin, G. Smith, Rely/guarantee reasoning for noninterference in non-blocking algorithms, in: 33rd IEEE Computer Security Foundations Symposium, CSF 2020, IEEE, 2020, pp. 380–394. `doi:10.1109/CSF49147.2020.00034`.
URL `https://doi.org/10.1109/CSF49147.2020.00034`

[19] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), Proceedings, IEEE Computer Society, 2002, pp. 55–74. `doi:10.1109/LICS.2002.1029817`.
URL `https://doi.org/10.1109/LICS.2002.1029817`

[20] T. C. Murray, R. Sison, E. Pierzchalski, C. Rizkallah, Compositional verification and refinement of concurrent value-dependent noninterference, in: IEEE 29th Computer Security Foundations Symposium, CSF 2016, IEEE Computer Society, 2016, pp. 417–431. `doi:10.1109/CSF.2016.36`.
URL `https://doi.org/10.1109/CSF.2016.36`

[21] A. Askarov, S. Chong, H. Mantel, Hybrid monitors for concurrent noninterference, in: C. Fournet, M. W. Hicks, L. Viganò (Eds.), IEEE 28th Computer Security Foundations Symposium, CSF 2015, IEEE Computer Society, 2015, pp. 137–151. `doi:10.1109/CSF.2015.17`.
URL `https://doi.org/10.1109/CSF.2015.17`

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

[23] T. C. Murray, R. Sison, K. Engelhardt, Covern: A logic for compositional verification of information flow control, in: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, IEEE, 2018, pp. 16–30. `doi:10.1109/EuroSP.2018.00010`.
URL `https://doi.org/10.1109/EuroSP.2018.00010`

[24] D. Schoepe, T. Murray, A. Sabelfeld, VERONICA: expressive and precise concurrent information flow security, in: 33rd IEEE Computer Security Foundations Symposium, CSF 2020, IEEE, 2020, pp. 79–94. `doi:10.1109/CSF49147.2020.00014`.
URL `https://doi.org/10.1109/CSF49147.2020.00014`

[25] A. Karbyshev, K. Svendsen, A. Askarov, L. Birkedal, Compositional non-interference for concurrent programs via separation and framing, in: L. Bauer, R. Küsters (Eds.), Principles of Security and Trust - 7th International Conference, POST 2018, Proceedings, Vol. 10804 of Lecture Notes in Computer Science, Springer, 2018, pp. 53–78. `doi:10.1007/978-3-319-89722-6\_3`.
URL `https://doi.org/10.1007/978-3-319-89722-6_3`

[26] G. Ernst, T. Murray, SecCSL: Security concurrent separation logic, in: I. Dillig, S. Tasiran (Eds.), Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings, Part II, Vol. 11562 of Lecture Notes in Computer Science, Springer, 2019, pp. 208–230. `doi:10.1007/978-3-030-25543-5\_13`.
URL `https://doi.org/10.1007/978-3-030-25543-5_13`

[27] D. Frumin, R. Krebbers, L. Birkedal, Compositional non-interference for fine-grained concurrent programs, in: 42nd IEEE Symposium on Security and Privacy, S&P 2021, IEEE, 2021.

[28] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, X. Wang, Scaling symbolic evaluation for automated verification of systems code with Serval, in: T. Brecht, C. Williamson (Eds.), Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, ACM,

2019, pp. 225–242. doi:10.1145/3341301.3359641.
URL https://doi.org/10.1145/3341301.3359641

[29] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, X. Wang, Nickel: A framework for design and verification of information flow control systems, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, 2018, pp. 287–305.
URL https://www.usenix.org/conference/osdi18/presentation/sigurbjarnarson

[30] S. Zdancewic, A. C. Myers, Observational determinism for concurrent program security, in: 16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA, IEEE Computer Society, 2003, p. 29. doi:10.1109/CSFW.2003.1212703.
URL https://doi.org/10.1109/CSFW.2003.1212703

[31] K. R. Apt, E. Olderog, Fifty years of hoare's logic, Formal Aspects Comput. 31 (6) (2019) 751–807. doi:10.1007/s00165-019-00501-3.
URL https://doi.org/10.1007/s00165-019-00501-3

[32] A. Sabelfeld, D. Sands, Probabilistic noninterference for multi-threaded programs, in: Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, IEEE Computer Society, 2000, pp. 200–214. doi:10.1109/CSFW.2000.856937.
URL https://doi.org/10.1109/CSFW.2000.856937

[33] H. Liang, X. Feng, M. Fu, A rely-guarantee-based simulation for verifying concurrent program transformations, in: J. Field, M. Hicks (Eds.), Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, ACM, 2012, pp. 455–468. doi:10.1145/2103656.2103711.
URL https://doi.org/10.1145/2103656.2103711

[34] C. S. Gordon, M. D. Ernst, D. Grossman, Rely-guarantee references for refinement types over aliased mutable data, in: H. Boehm, C. Flanagan (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, ACM, 2013, pp. 73–84. doi:10.1145/2491956.2462160.
URL https://doi.org/10.1145/2491956.2462160

[35] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008. Proceedings, Vol. 4963 of Lecture Notes in Computer Science, Springer, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.
URL https://doi.org/10.1007/978-3-540-78800-3_24

[36] N. Coughlin, K. Winter, G. Smith, Rely/guarantee reasoning for multicopy atomic weak memory models, in: M. Huisman, C. Pasareanu, N. Zhan (Eds.), Formal Methods - 24th International Symposium, FM 2021, Lecture Notes in Computer Science, Springer, 2021.

[37] K. Winter, N. Coughlin, G. Smith, Backwards-directed information flow analysis for concurrent programs, in: 34th IEEE Computer Security Foundations Symposium, CSF 2021, IEEE, 2021, pp. 1–16. doi:10.1109/CSF51468.2021.00017.
URL https://doi.org/10.1109/CSF51468.2021.00017