Formal Methods in System Design (2021) 58:251–293
https://doi.org/10.1007/s10703-021-00376-2

Information-flow control on ARM and POWER multicore processors

Graeme Smith[1,2] · **Nicholas Coughlin**[2] · **Toby Murray**[3]

Received: 28 February 2020 / Accepted: 19 May 2021 / Published online: 8 June 2021

Abstract
Weak memory models implemented on modern multicore processors are known to affect the correctness of concurrent code. They can also affect whether or not the concurrent code is secure. This is particularly the case in programs where the security levels of variables are *value-dependent*, i.e., depend on the values of other variables. In this paper, we illustrate how instruction reordering allowed by ARM and POWER multicore processors leads to vulnerabilities in such programs, and present a compositional, flow-sensitive information-flow logic which can be used to detect such vulnerabilities. The logic allows step-local reasoning (one instruction at a time) about a thread's security by tracking information about dependencies between instructions which guarantee their order of occurrence. Program security can then be established from individual thread security using rely/guarantee reasoning. The logic has been proved sound with respect to existing operational semantics using Isabelle/HOL, and implemented in an automatic symbolic execution tool.

Keywords Information-flow security · Weak memory models · Non-blocking algorithms

1 Introduction

It is well known that compiler optimisations may violate security guarantees apparent at the level of source code [11], even when these optimisations have been proven to preserve functional correctness. For example, compilers may reorder instructions, or even remove them as seen in dead code elimination. While these modifications may not change the functional

✉ Graeme Smith
smith@itee.uq.edu.au

Nicholas Coughlin
n.coughlin@uq.edu.au

Toby Murray
toby.murray@unimelb.edu.au

1 Defence Science and Technology Group, Brisbane, Australia

2 School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia

3 School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia

outcomes of a program, they may modify the durations within which secret information is held in memory. Consequently, this data may be leaked due to correctness issues in the application or malicious code linked to the compiled binary.

Moreover, some security properties, such as an absence of timing side-channels, cannot be established directly via preservation of traditional program-language semantics. Since traditional semantics ignore timing, and semantics preservation ignores internal branching, the compiler is able to introduce arbitrary control flow and calculations with data-dependent timings potentially leaking secret information.

One approach to combating such a problem would be to establish a sufficiently detailed semantics and ensure preservation of security properties via a verified compiler [20,38]. See Barthe et al. [3] and Sison and Murray [36] for recent work in this direction. However, this approach represents a tremendous amount of effort since current verified compilation projects focus on traditional semantics. Alternatively, security properties can be shown on the late stages of the compilation process under the assumption they are preserved by the final compilation stages, or on the assembly instructions directly [5]. Under the latter approach, only a semantics of the hardware assembly instructions is required.

Optimisations implemented by the processor architecture need to be taken into account when verifying security properties at the assembly level. For example, when considering concurrency, the architecture's memory model can lead to additional security violations when compared to an analysis that assumes a sequentially consistent memory model. This has been shown for the TSO architecture by Vaughan and Milstein [39] and for TSO, PSO and IBM-370 by Mantel et al. [22]. While TSO [35] is widely used (by chip manufacturers Intel, AMD and SPARC), PSO and IBM-370 are not supported on recent processors. More relevant architectures are ARM [14,31] and IBM POWER [33]; the former being widely used in mobile devices [13]. These architectures are significantly weaker than those studied by the papers above, yet have received little attention from the security foundations community.

In recent work [37], we provide an information-flow logic for reasoning about security on the latest (revised) version of ARMv8 [31].[1] This logic is restricted to a core subset of (abstracted) ARM instructions, ignoring the different types of memory barriers available, and mechanisms inherent in low-level assembly code such as address shifting. Furthermore, the ARMv8 architecture is simpler than previous ARM architectures, and also simpler than IBM POWER, due to being *multi-copy atomic*, i.e., an update to a variable by a given thread is seen by all other threads at the same time.

In this paper, we extend the work of [37] to provide a more complete logic and reasoning approach, not just for ARMv8, but also for earlier versions of ARM and for POWER. We also describe how reasoning in the logic can be automated via symbolic execution.

We begin in Sect. 2 by providing an overview of information-flow logics for concurrent programs. In Sect. 3 we introduce instruction reordering as a common weak memory model optimisation, and explain how it can lead to security vulnerabilities in a concurrent setting. We also introduce *instruction dependencies* as a key concept in determining information-flow under typical hardware optimisations. In Sect. 4, we show how this concept can be used in defining information-flow rules for various ARM instructions and optimisations. This includes a number of instructions not covered in our earlier work such as store fences, control fences, compare-and-swap, load-acquire and store-release instructions and address shifting. It also includes optimisations not covered by our earlier work such as write elimination and load speculation.

[1] We will refer to this revised version as simply ARMv8 in the remainder of this paper.

Our logic has been encoded in Isabelle/HOL [30] and proved sound with respect to an encoding of operational semantics of ARM and POWER [8]. We provide a high-level overview of the soundness argument in Sect. 5. Our logic enforces a so-called *constant-time* [2] security guarantee that forbids programs from branching on secrets and from performing secret-dependent memory accesses. This kind of property is commonly used to guard against timing channel leakage via caches. However, we make the novel observation that constant-time security appears to be a necessary ingredient for ensuring the absence of leakage via the resolution of nondeterminism in the weak memory model itself (see Sect. 5.3).

We have also encoded the logic for ARMv8 in a prototype symbolic execution tool building on that for C code of Ernst and Murray [12]. This tool is described in Sect. 7.

In Sect. 8, we consider using the logic for IBM POWER and define rules for additional memory barrier instructions available on POWER. Both POWER and earlier versions of ARM are non-multi-copy atomic. In Sect. 9, we reflect on security vulnerabilities that non-multi-copy atomicity can introduce, and argue that these do not affect the soundness of our logic. In Sect. 10 we apply our logic and symbolic execution tool to a case study, a cross-domain work stealing deque, before concluding in Sect. 11.

2 Information flow control for concurrent programs

Information-flow logics [32] comprise a set of rules, typically one for each kind of program instruction, which are used to determine whether an instruction can leak information. To do this, the logic needs to assign a *security classification* to each variable, denoted $\mathcal{L}(x)$ for variable x. This denotes the maximum *security level* of data the variable may hold. Generally, the security levels are defined by a lattice, the simplest being a two-point lattice with values $High$ and Low such that $Low \sqsubseteq High$ and $High \not\sqsubseteq Low$. This simple lattice indicates that $High$ data (representing sensitive information) should not flow to variables with Low classification (assumed to be visible to an attacker trying to learn sensitive information), but the other direction of flow is allowed (capturing confidentiality).

Each rule in an information-flow logic refers to the context in which the instruction occurs. This context keeps track of required information such as the security level of the data held by program variables. In *flow-sensitive* logics, the context is updated by the rule to provide the context for the following instruction in the program. For example, a typical rule for assignment is

$$\textsc{Assign}\ \frac{\Gamma \vdash e : t \qquad t \sqsubseteq \mathcal{L}(x)}{\Gamma \quad \{x{:=}e\} \quad \Gamma[x \mapsto t]}$$

where the premisses state that from the security levels in the context Γ, we can deduce the security level t of expression e, and that t is lower than the security classification of variable x. The rule updates Γ so that x maps to t.

An important issue when reasoning about concurrent systems is compositionality. For scalability, we want to reason about individual threads in isolation and combine this reasoning to deduce properties of the entire program. One way to do this is to utilise rely/guarantee reasoning [6,16]. An assumption (or rely condition) expresses what a thread can rely on its environment (the other threads) doing. For example, a thread may rely on the fact that no other thread writes to a variable x. A guarantee expresses what a thread guarantees to its environment. For example, a thread may guarantee that it does not write to a variable x. Reasoning done on an individual thread will be valid in the wider context of its execution if all of its assumptions are matched by a guarantee from all other threads. For example, if

the thread assumes that no other thread writes to a variable x then all other threads must guarantee that they do not.

Mantel et al. [23] adopt this approach in their seminal concurrent information-flow logic by associating variables referenced by a thread with one or more of the following *modes*.

- $ass\text{-}noread$ - the variable is not read by another thread
- $ass\text{-}nowrite$ - the variable is not written to by another thread
- $guar\text{-}noread$ - this thread does not read the variable
- $guar\text{-}nowrite$ - this thread does not write to the variable

Their context Γ is restricted to low variables with mode $ass\text{-}noread$ and high variables with mode $ass\text{-}nowrite$, i.e., $\text{dom}\,\Gamma = \{x | (\mathcal{L}(x) = Low \wedge x \in ass\text{-}noread) \vee (\mathcal{L}(x) = High \wedge x \in ass\text{-}nowrite)\}$.

There are then two assignment rules. The first is for variables not in Γ. If the variable is low, the third premise ensures that it is not assigned a high value. Γ is not updated since another thread may overwrite the value in x before this thread's next instruction.

$$\textsc{Assign1}\ \frac{x \notin \text{dom}\,\Gamma \qquad \Gamma \vdash e : t \qquad t \sqsubseteq \mathcal{L}(x)}{\Gamma \quad \{x := e\} \quad \Gamma}$$

The second rule is for variables in Γ. In this case, it is not necessary to restrict the value assigned to x; if it is a low variable, the thread is relying on it not being read, so there is no chance of information leaking via the variable. Also, since the thread relies on high variables in Γ not being written to, and low variables only being overwritten with lower, and not higher, values, the security level of the value written can be used when the rule for the following instruction is applied (and hence is maintained in the context Γ).

$$\textsc{Assign2}\ \frac{x \in \text{dom}\,\Gamma \qquad \Gamma \vdash e : t}{\Gamma \quad \{x := e\} \quad \Gamma[x \mapsto t]}$$

Murray et al. [29] extend this approach to allow *value-dependent* security classifications, i.e., where the security classification of a variable depends on the values held by one or more other variables [21,27,40]. In that work, $\mathcal{L}(x)$ denotes a predicate on the program's variables.[2] This predicate is true if, and only if, x's classification is low. Variables which appear in such predicates, and hence control the security classification of other variables, are called *control variables*. Control variables are always low and are not included in Γ.
The modes of Mantel et al. are replaced by

- $AssNoRW$ - the variable is not read or written to by another thread
- $AssNoW$ - the variable is not written to by another thread (but may be read by it)
- $GuarNoRW$ - this thread does not read or write to the variable
- $GuarNoW$ - this thread does not write to the variable (but may read it)

The variables in Γ are the non-control variables that are *stable*, i.e., that are associated with either mode $AssNoW$ or $AssNoRW$ and hence are assumed not to be writable by other threads. As well as Γ, an instruction's context includes the sets of stable variables, captured by the ordered pair $S = (AssNoW, AssNoRW)$, and a predicate P reflecting the current program state.[3] $\mathcal{C}$ denotes the set of control variables of a program.

[2] $\mathcal{L}(x)$ is denoted $\mathcal{L}\text{type}\ x$ in [29]. It actually denotes a set of predicates; here we simplify our presentation by referring to the conjunction of that set's elements.

[3] Again P in [29] is a set of predicates and we simplify our presentation by referring to the conjunction of that set's elements.

Fig. 1 Example illustrating the need for timing-sensitive security

Thread 1:

```
low := 0;
if (high=0)
then while (high < 1000) high++;
else skip;
low := 1;
```

Thread 2:

```
output := low;
```

For non-control variables, there are again two rules for assignment. The first is for non-stable variables. It requires that t, the security level of the value assigned to x, is less than x's classification under P, i.e., unless $P \implies \neg\mathcal{L}(x)$, t must be low. This is denoted by $t \sqsubseteq_P \mathcal{L}(x)$.

$$\textsc{Assign1}\ \frac{x \notin \mathrm{dom}\ \Gamma \cup \mathcal{C} \qquad \Gamma \vdash e : t \qquad t \sqsubseteq_P \mathcal{L}(x)}{\Gamma, S, P \quad \{x := e\} \quad \Gamma, S, P}$$

The second rule is for stable variables. It is only necessary to restrict the value assigned to x if $x \notin \mathrm{snd}\ S$, i.e., x can be read by other threads.[4] The rule updates Γ with the new security level for x, and updates P to the strongest postcondition reachable from P when executing $x := e$ (denoted here as $P[x := e]$), restricting the resulting predicate to stable variables. The restriction is necessary since unstable variables may be modified by another thread and hence should not appear in P when checking the next instruction in the program.

$$\textsc{Assign2}\ \frac{x \in \mathrm{dom}\ \Gamma \qquad \Gamma \vdash e : t \qquad x \notin \mathrm{snd}\ S \implies t \sqsubseteq_P \mathcal{L}(x)}{\Gamma, S, P \quad \{x := e\} \quad \Gamma[x \mapsto t], S, P[x := e] \upharpoonright S}$$

Although both Mantel et al.'s and Murray et al.'s approaches allow assumptions and guarantees to change during the execution of a program, neither provide a means of ensuring modes are consistent between threads, i.e., that the assumptions of one thread are guaranteed by all others. This is due to not enforcing synchronisation between threads when an assumption is updated. Such synchronisation is required for other threads to update their guarantees to match the new assumption. This issue is addressed in COVERN [28], an extension to the approach of Murray et al. in which locks protect access to shared variables, allowing the owner of a lock to assume no other thread can write to the variables that the lock protects, or read those variable, as appropriate. For this paper, we focus on non-blocking (i.e. lock-free) code, and avoid the issue by assuming that assumptions and guarantees do not change during the execution of a program. A more flexible approach is the subject of ongoing work as discussed in Sect. 11.

2.1 Timing sensitivity

The approach of Murray et al. does not allow branching based on a high value. The justification for this restriction is based on the fact that a compositional information-flow logic must be *timing-sensitive*, i.e., information should not be leaked to an attacker who is able to time the execution of a program. As argued in [28], this is not possible in the presence of paths entered depending on the value of a high value.

For example, consider the program in Fig. 1 in which $high$ is a high variable and low and $output$ are low variables. Both threads are timing-insensitive secure since low is never dependent on the value of $high$. However, when they are composed the value written to $output$ is more likely to be 0 than 1 when $high$ is 0. Hence, although the threads are timing-insensitive secure, their composition is not. This does not require a probabilistic argument:

[4] We use snd S to denote the second element of pair S. Similarly, we will use fst S to denote the first element.

under a round-robin scheduler with time slices less than the time it takes to execute the loop, the result $output = 1$ would indicate that $high \neq 0$.

The first thread is obviously not timing-sensitive secure (as its execution time depends directly on $high$) and hence under timing-sensitive security the issue with compositionality does not arise. Eliminating branching on high values from code can be achieved using program transformations as described, for example, in [2,25].

3 Weak memory models

Hardware weak memory models, as exemplified by TSO [35], ARM [14,31] and IBM POWER [33], aim at optimising assembly code by restricting accesses to global shared memory: a well known cause of inefficiency in multicore systems. This can be achieved, for example, by buffering writes to memory and letting the hardware control when those writes actually occur, or by allowing *speculative execution* of code occurring in a branch of the program before evaluating whether that branch should be taken (which may require access to shared memory). It can also be achieved by propagating writes to other cores rather than the shared memory (referred to as *non-multi-copy atomicity* since different cores may receive a particular write at different times).

The effects of such optimisations can lead to the instructions of one thread appearing to occur out-of-order from the perspective of threads running on other cores. For example, if a thread t buffers the writes to variables x and y while executing the code $x := 1; y := 2$ and then the hardware flushes the value assigned to y first, it appears to threads running on other cores as if t executed the code $y := 2; x := 1$.

Colvin and Smith [8,9] define four constraints related to this perceived reordering of assignments on weak memory models. These constraints, which are common to all contemporary weak memory models, ensure that the sequential semantics of the thread on which the reordering occurs is unchanged. An assignment $x := e$ can be reordered with an assignment $y := f$ if, and only if,

(i) x and y are distinct variables;
(ii) x is not referred to in f;
(iii) y is not referred to in e; and
(iv) e and f do not reference any common global variables.

Constraint (i) is obviously required as $x := 1; x := 2$ has a different final value of x (and hence different behaviour) than $x := 2; x := 1$. Constraint (ii) is required since $x := 1; y := x$ will result in a different value for y than $y := x; x := 1$ when the initial value of x is not 1. Similarly, constraint (iii) is required since $x := y; y := 1$ can result in a different value for x than $y := 1; x := y$. Finally, constraint (iv) is required so that the order of updates and accesses of each global variable, considered individually, is maintained: $x := z; y := z$ will not behave the same as $y := z; x := z$ in an environment which modifies z since the former will never result in y having an earlier value of z than x.

In contemporary processors, constraint (ii) is weakened by *forwarding* which allows a program such as $x := e; y := x$ to be reordered to $y := e; x := e$ when e does not refer to global variables, i.e., the effect of the first assignment is taken into account when determining whether the second can be reordered with it.

Specific memory models may add additional constraints, e.g., TSO does not allow a write to a global variable to be reordered with a subsequent write to a global variable. Fences are a means by which the programmer can enforce ordering where necessary in their program. For

```
write:          secret_write:        read:                secret_read:
x := data       z := z+1;            do                   y := x
                x := secret;             do
                ...                          r1:= z;
                x := 0;                  while (r1 % 2 ≠ 0)
                z := z+1                 r2 := x;
                                     while (z ≠ r1)
                                     y := r2
```

Fig. 2 An IO-driver object with operations for accepting input from a keyboard at unclassified (write) and classified (secret_write) levels, and for reading input data at unclassified (read) and classified (secret_read) levels

example, letting $fence$ denote a full fence (e.g., the instruction DMB on ARM), the program $x := 1; fence; y := 2$ ensures the write to x is seen by other threads before the write to y.

A full set of reordering constraints for TSO, ARM and POWER which have been validated against existing test suites on hardware is provided in [8,9]. These include reordering constraints related to other types of instructions such as branch instructions and fences. For ARM and POWER, we have the following constraints for branch instructions.

(v) An assignment $x := e$ following a branch instruction with branching condition b can be reordered with the branch instruction if, and only if, x is a local variable and does not appear free in b, and b and e do not reference common global variables.
(vi) An assignment $x := e$ preceding a branch with branching condition b can be reordered with the branch if, and only if, x does not appear free in b, and b and e do not reference common global variables.
(vii) Two branch instructions can be reordered if, and only if, their branching conditions do not reference common global variables.

In rule (v) the assignment is speculatively executed (before the branch condition is evaluated). It is therefore restricted to assignments to local variables since if it is later determined that the branch should not be executed, it is necessary to discard the results of such assignments. This cannot be done with assignments to global variables.

As with assignment, constraint (vi) is weakened by *forwarding*. A program such as $x := e; if\ (x = f) \ldots$ can be reordered to $if\ (e = f)\ x := e; \ldots$ when e does not refer to global variables.

3.1 Instruction reordering and value-dependent security

To illustrate how instruction reordering may affect security in the presence of value-dependent security classifications [29], we introduce the example of Fig. 2. In this example, the four operations are of an IO-driver object which receives input data from an IO device, such as a keyboard, and stores it in the variable x. This variable is intended to be an abstract representation of an input buffer.

As well as a simple $write$ operation, the object has a $secret\_write$ operation. This is used when the user indicates (via the keyboard or another input device) that the information to be input is classified. The operation sets a variable z, which is initially 0, to an odd number by incrementing it before allowing the input data to be assigned to x. After allowing the data to be read (how this is done is elided in the abstract representation of Fig. 2), the operation enters some unclassified data in x (the value 0) before setting z back to an even number by

incrementing it again. Since x is guaranteed to be low whenever z is even, the value-dependent security classification for x is $\mathcal{L}(x) = (z \,\%\, 2 = 0)$.

Consider the operations which read from the buffer. We have a $secret\_read$ operation which can only be called by applications that are allowed access to classified information, as well as a general $read$ operation which all applications can call. To avoid leaks of classified data, the latter should not read the variable x when z is odd; this is the only time when x can contain classified data. A naive approach would be to use an if statement in $read$ to disallow reading x when z is odd: $if\ (z \,\%\, 2 = 0)\ then\ y := x\ else\ skip$ where y is a variable which the application calling the operation can access. Obviously, this will not work in a concurrent setting since the check of z's value could be made immediately before z is incremented for the first time by $secret\_write$ and subsequently the assignment to y made immediately after x is assigned the classified data; a classic Time-of-check to time-of-use (TOCTOU) vulnerability.

To avoid this undesirable behaviour, we could ensure mutual exclusion between the operations $secret\_write$ and $read$ using a lock; each of these operations would acquire the lock as it first step and release it as its last. This, however, would be highly inefficient. Firstly, there may be many applications running and wishing to access the keyboard data, and requiring each to acquire the lock before reading would create an obvious bottleneck. Secondly, the $secret\_write$ operation should preferably not be made to acquire a lock as it needs to react without delay in order to accept (real-time) keyboard input.

A better solution is to use a *non-blocking algorithm* [24]. Such algorithms allow threads to run concurrently on the same object with no, or minimal, use of locking. For example, consider the implementation of $read$ in Fig. 2 where $r1$ and $r2$ are local variables. This operation waits in a loop until z is even (and hence x does not contain classified information) and then reads x into $r2$. It then checks that z has not changed (and hence has been even the entire time since it was checked) before copying the value of $r2$ to y. Since z can only stay at its current value or increase, if its value is the same as at some earlier time t, we can deduce that z has not changed since time t.

This algorithm allows the $secret\_write$ operation to operate without locking or delay, and allows multiple threads to call the $read$ operation simultaneously. It is based on a Linux read-write mechanism called seqlock [4], and is a typical example of a non-blocking algorithm.

The implementation in Fig. 2 is secure on a *sequentially consistent* memory model, i.e., one that does not allow instruction reordering. It is also secure on a memory model such as TSO where writes are seen by other threads in the order in which they occur. For weaker memory models such as ARM and POWER, this is not the case. These memory models allow writes by a thread to be seen out-of-order by other threads since no additional constraints are added to the four common reordering constraints for assignments presented in Sect. 3.

For example, consider the operation $secret\_write$. If from the perspective of threads running $read$, the assignment of the classified data to x occurred before the first assignment to z then that classified data could be read into the variable y. To avoid this situation, a fence is required between these two assignments. Similarly, if the second assignment to z occurred before the assignment of 0 to x then again the classified data in x could be read into y. The solution again is to maintain the order by placing a fence between these assignments. A secure version of $secret\_write$ is given in Fig. 3.

Similar issues arise with the $read$ operation. Firstly, since $r2$ is a local variable, the assignment to $r2$ could be reordered with the first branch instruction (rule (v)) and further reordered with the assignment to $r1$. This results in reading a value of x into $r2$ before checking that z is even. If this value is classified and subsequently z is made even by $secret\_write$, the check will pass and the classified information in $r2$ will be able to be passed into y. A fence before the assignment to $r2$ will prevent this reordering.

Fig. 3 Versions of the operations (secret_write) and (read) which are secure when run on ARM and POWER processors

```
secret_write:
z := z+1;
fence;
x := secret;
...
x := 0;
fence;
z := z+1
```

```
read:
do
    do
        r1:= z;
    while (r1 % 2 ≠ 0)
    fence;
    r2 := x;
    fence;
while (z ≠ r1)
y := r2
```

Secondly, if the assignment to $r2$ is reordered with the second branch condition (rule (vi)) then it is possible that a $secret\_write$ operation begins after the check of that branch condition and $r2$ is loaded with classified data. Again, a fence can prevent the reordering. A secure version of $read$ is included in Fig. 3.

3.2 Instruction dependencies

The value-dependent information-flow logic of Murray et al. [29] described in Sect. 2 is not capable of detecting the security leaks present in the code of Fig. 2 when run on ARM or POWER processors. Γ and P ignore the effects of reordering possible under the processors' memory models. This is not a problem for Γ as it is only consulted for the reads of an instruction. If an instruction containing an expression e is reordered before a prior write to a variable x then, according to the constraints in Sect. 3, either (i) x is not in e, or (ii) x is in e and the reordering involves forwarding. In case (i), the assignment does not affect the value of Γ for any of the variables in e and hence does not affect the sensitivity of the data contained in e. In case (ii), since forwarding involves taking into account the prior assignment's effect, using the updated value for x in Γ is appropriate.

P, on the other hand, cannot be used directly to determine the security level of a variable or expression. For example, the code $z := z + 1; x := secret$ of Fig. 2 does not ensure z is incremented before the assignment to x. To extend information-flow analysis to account for such weak memory model effects, we need to restrict P to include only the effects of those instructions which have definitely occurred prior to the instruction being considered.

To do this, we introduce a function W mapping instructions to sets of variables. $W(\alpha)$ denotes the set of 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. We then restrict P at instruction α to those variables in $W(\alpha)$. For example, consider the code $z := z+1; x := secret$ starting from a state where $z = 0$. After the first assignment $W(x := secret)$ would include x (since rule (i) prevents any earlier writes to x being reordered with $x := secret$), but would not include z. Hence, the predicate P at this point would not refer to the purported fact that $z = 1$.

Importantly, the value of $W(\alpha)$ changes as we step through a program. Hence, $W(\alpha)$ is not necessarily the same for different occurrences of the same instruction α. 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 constraint (iv) of Sect. 3. However, after the third assignment $W(y := x)$ does not contain z since the fourth assignment can be reordered before the third.

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 $z := x$ (rule (i) of Sect. 3). 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 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. Colvin and Smith [8,9] define a reordering relation $\alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle}$ on instructions α and β, where $\beta_{\langle\alpha\rangle}$ encodes the instruction β with the effects of forwarding the earlier assignment α.

Recall that forwarding enables reordering of instructions such as those in $x := e; y := x$ by replacing the occurrence of x in the second assignment with the value e (forwarded from the first assignment). That is, we replace $x := e; y := x$ by $x := e; y := e$ which is then reorderable under the rules (i)–(iv) to $y := e; x := e$. Forwarding the value of an assignment $x := e$ can therefore be formalised as replacing each occurrence of x in an expression or branch condition with e, and leaving other instructions, such as fences, unchanged. Given $[b]$ denotes a branch condition, we define forwarding as follows.

$$\begin{array}{rll} y := f_{\langle x:=e\rangle} = y := f[e/x] & & \text{if } e \text{ does not refer to global variables} \\ [b]_{\langle x:=e\rangle} = [b][e/x] & & \text{if } e \text{ does not refer to global variables} \\ \beta_{\langle\alpha\rangle} = \beta & & \text{for other instructions} \end{array}$$

We restate the relevant parts of relation $\overset{R}{\Leftarrow}$ of [8,9] corresponding to constraints (i) to (vii) of Sect. 3 below.

$$\begin{array}{rll} x := e \overset{R}{\Leftarrow} y := f & & \text{if constraints (i), (ii), (iii) and (iv) hold} \\ [b] \overset{R}{\Leftarrow} x := e & & \text{if constraint (v) holds} \\ x := e \overset{R}{\Leftarrow} [b] & & \text{if constraint (vi) holds} \\ [b_1] \overset{R}{\Leftarrow} [b_2] & & \text{if constraint (vii) holds} \end{array}$$

The update of W when an instruction α occurs is then defined as follows.

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

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. Returning to the example $z := z+1; x := secret$, irrespective of the value of $W(x := secret)$ before execution, after the first assignment $\{z\}$ will not be in $W(x := secret)$. This is because $x := secret_{\langle z:=z+1\rangle}$ is $x := secret$ and since $z := z + 1 \overset{R}{\Leftarrow} x := secret$ the first case above applies. This case sets $W(x := secret)$ to its current value with $wr(z := z + 1) = \{z\}$ removed.

It is necessary to use $W(\beta_{\langle\alpha\rangle})$ rather than $W(\beta)$ in this case of the definition, 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 $W(z := 5) = \{z\} - \{y\} = \{z\}$. Also, after the second assignment $W(z := y) = W(z := 5) - \{x\} = \{z\}$. Hence, when the third assignment is evaluated, neither x nor y are considered to be up-to-date.

This function can be used in any rules where we need to refer to the prior writes in a program that have definitely occurred. In some rules, we also need to refer to the prior reads in a program that have definitely occurred. Let R be a function mapping instructions to sets of variables such that $R(\alpha)$ denotes the set of variables whose prior reads in the program have occurred. Updates to R can be calculated in an analogous fashion to those of W as follows (where $rd(\alpha)$ is the set of variables read by α).

$$R[\alpha](\beta) = \begin{cases} R(\beta_{\langle\alpha\rangle}) - rd(\alpha) & \text{if } \alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle} \\ R(\beta) \cup R(\alpha) & \text{otherwise} \end{cases}$$

Considering the example $z := z + 1; x := secret$ once more, we have that after the first assignment $R(x := secret)$ does not include variables in $rd(z := z + 1) = \{z\}$. This reflects that the read of z in $z := z + 1$ need not occur before the assignment $x := secret$.

4 Information flow on ARMv8

In this section, we present our information-flow logic for ARMv8 (we consider earlier versions of ARM in Sect. 9). We let Var be the set of all program variables. Variables are partitioned into global (i.e., shared) variables $Global$, and local variables $Local$, i.e., $Var = Global \cup Local$ and $Local \cap Global = \emptyset$. We let $var(e)$ denote the set of variables which occur free in an expression e.

Our logic is defined over a high-level programming language which can model ARM and POWER instructions (as in [9]). Let b and e represent boolean and value expressions respectively, with the constraint that they are deterministic and their execution time is data-independent. Additionally, we let x represent any program variable, r represent a $Local$ variable, g represent a $Global$ variable, and l a value expression over $Local$ variables. Our language is then defined in terms of commands c as follows.

$$\begin{aligned} a &::= x := e | r := \mathsf{CAS}(g, l, l) | \mathsf{acq}(r, g) | \mathsf{rel}(g, l) \\ f &::= \mathsf{fence} | \mathsf{fence.st} | \mathsf{cfence} | \mathsf{eieio} | \mathsf{loadgate} | \mathsf{storegate} \\ c &::= \mathsf{skip} | c; c | \mathsf{if}\ (b)\ \mathsf{then}\ c\ \mathsf{else}\ c | \mathsf{while}\ (b)\ \mathsf{do}\ c | a | f \end{aligned}$$

where $r := \mathsf{CAS}(g, l, l)$ is a compare-and-swap instruction (detailed in Sect. 4.13), $\mathsf{acq}(r, g)$ and $\mathsf{rel}(g, l)$ denote load-acquire and store-release instructions respectively (detailed in Sect. 4.14), and f denotes a range of fences (detailed in Sects. 4.11 and 8).

The reordering semantics [8,9], upon which this logic is based, is defined in terms of Kleene algebra. Therefore, we define our language constructs accordingly. A statement $\mathsf{if}\ (b)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ represents the sequence of instructions $([b]; c_1) \sqcap ([\neg b]; c_2)$ where $\sqcap$ is non-deterministic choice, and $[b]$ and $[\neg b]$ acts as guards, resolving the statement to one branch or the other in any state. Similarly, $\mathsf{while}\ (b)\ \mathsf{do}\ c$ denotes $([b]; c)^*;[\neg b]$. The $\mathsf{do}\ c\ \mathsf{while}\ (b)$ construct used in the code of Figs. 2 and 3 is simply a shorthand for c ; $\mathsf{while}\ (b)\ \mathsf{do}\ c$.

We restrict classifications to operate over a two-point security lattice, containing $High$ and Low, structured such that $Low \sqsubseteq High$ and $High \not\sqsubseteq Low$. Moreover, we allow for the specification of a value-dependent security policy $\mathcal{L}$, mapping variables to their value-dependent security classifications as detailed in Sect. 4.2.

The logic supports variable modes $AssNoW$, $AssNoRW$, $GuarNoRW$ and $GuarNoW$, similar to prior work [28,29]. We represent these modes as a mapping M from modes to sets

of variables, e.g., $x \in M(AssNoRW)$ indicates that x is assumed to not be read or written by another thread. These modes only apply to $Global$ variables as $Local$ variables can only be read or written by their thread. We introduce $\mathsf{stable}\ M \mathrel{\widehat{=}} M(AssNoRW) \cup M(AssNoW) \cup Local$ to denote the set of variables that no other component will modify.

4.1 State

Our information-flow logic is applied as a forward pass over the command c of an individual thread, under mode state M, using judgements of the form $\Gamma, P, D\ \{c\}_M\ \Gamma', P', D'$. These individual judgements are then composed to establish information-flow properties over a concurrent system.

The logic's state is an extension of the program state with the information required to evaluate information flow under a weak memory model. It consists of the triple Γ, P, D, where Γ encodes the type context, mapping variables to the security level of their values (which are either $High$ or Low); P encodes a predicate over the program state; and D captures instruction dependencies as a tuple (W, R), where W and R are the functions introduced in Sect. 3.2. For brevity, we introduce the shorthands $D_W \mathrel{\widehat{=}} \mathrm{fst}\ D$ and $D_R \mathrel{\widehat{=}} \mathrm{snd}\ D$. Additionally, we introduce an update function $D[\alpha] \mathrel{\widehat{=}} (D_W[\alpha], D_R[\alpha])$.

We introduce a concept of well-formedness for the logic's state, to ensure modifications from concurrent threads cannot invalidate properties and, therefore, judgements. The first component of this ensures that the free variables in P are all in $\mathsf{stable}\ M$, avoiding invalidation due to an assignment in a concurrent thread. Similarly, it is necessary to restrict the domain of Γ to $\mathsf{stable}\ M$, as threads may modify other variables and, consequently, the security level of their values.

$$wf\ M\ \Gamma\ P \mathrel{\widehat{=}} var(P) \subseteq \mathsf{stable}\ M \wedge \mathrm{dom}\ \Gamma = \mathsf{stable}\ M$$

Initial conditions for the logic's P and Γ components may be any pair satisfying the wf property given an initial mode setting M. We default to the weakest possible values, such that P is $True$ and Γ maps variables in $\mathsf{stable}\ M$ to $High$. The initial D is structured such that all instructions map to all variables; formally $\forall \alpha \cdot D_W(\alpha) = D_R(\alpha) = Var$.

4.2 Classifications

As in the work of Murray et al. [28,29], the logic supports a security policy $\mathcal{L}$, mapping variables to their value-dependent classifications. These value-dependent classifications are encoded as predicates over control variables, such that $\mathcal{L}(x)$ is true precisely when x has a security level Low. To illustrate, consider the example of Sect. 2, in which $\mathcal{L}(x) = (z \,\%\, 2 = 0)$. This policy states that the classification of x depends on the value of control variable z, such that x must hold Low information when z is even. The security policy $\mathcal{L}$ is provided by the user and remains unmodified throughout execution.

We let $\mathcal{C}$ be the set of control variables of a program, such as z in the running example. These variables are restricted to having Low classifications. Such a restriction avoids possible side-channels due to changes in classifications. For example, if an attacker is permitted to access variables with value-dependent classifications when they are Low but not when they are $High$, the difference in access permissions may introduce a side-channel.

We prevent $Local$ variables from being used as control variables, formally $Local \cap \mathcal{C} = \emptyset$, as it is not possible for threads to coordinate based on variables they cannot read. Moreover, we

assume that $Local$ variables are classified as $High$, allowing them to hold any information. This is enforced via a constraint on the security policy: $\forall r : Local \cdot \mathcal{L}(r) = False$.

The logic determines the classification of a variable x based on $\mathcal{L}$, given the current state predicate P. To enable this, we introduce $low_P(x) \mathrel{\widehat{=}} P \vdash \mathcal{L}(x)$ and $high_P(x) \mathrel{\widehat{=}} P \vdash \neg\mathcal{L}(x)$ to determine if x is provably Low or $High$ respectively. Note that, given insufficient information in P, it may not be possible to demonstrate either of these conditions in which case the classification of x is considered unknown. It is necessary to handle this unknown case as though either classification is valid. To achieve this, it is necessary to distinguish whether a read or write is taking place.

First, consider writing to a variable x. If it is possible to show x is classified as $High$, then it can hold any value. However, if it is Low, then the value written to it must be Low. Therefore, it is necessary to assume a Low classification when writing to a variable with unknown classification as it is the constraining case. We capture this defaulting behaviour with $\mathcal{W}_P$.

$$\mathcal{W}_P(x) \mathrel{\widehat{=}} \begin{cases} High & \text{if } high_P(x) \\ Low & \text{otherwise} \end{cases}$$

Second, consider reading from a variable x. If it is possible to show x is classified as Low, then it can be included in an expression without changing the expression's security level. However, if it is $High$, the expression is consequently considered $High$. Therefore, it is necessary to assume a $High$ classification when reading from a variable with unknown classification. We introduce $\mathcal{R}_P$ to capture this behaviour.

$$\mathcal{R}_P(x) \mathrel{\widehat{=}} \begin{cases} Low & \text{if } low_P(x) \\ High & \text{otherwise} \end{cases}$$

When considering variable reads in expressions, it is also necessary to consider the local type context Γ. Γ may provide more accurate security levels for values held in variables in $\mathsf{stable}\ M$, such as capturing situations where a stable variable is classified $High$ but currently holds Low data. To simplify the use of Γ, we introduce a total mapping, defaulting to $\mathcal{R}_P$ where necessary.

$$\Gamma_P(x) \mathrel{\widehat{=}} \begin{cases} \Gamma(x) & \text{if } x \in \operatorname{dom}\Gamma \\ \mathcal{R}_P(x) & \text{otherwise} \end{cases}$$

We also define the following shorthand for determining the security level t of an expression e, as the highest level of any free variable in e.

$$\Gamma, P \vdash e : t \mathrel{\widehat{=}} t = \sqcup_{x \in var(e)} \Gamma_P(x)$$

4.3 Skip

Based on the preceding sections, we now introduce the rules of our logic. The rule for the skip instruction leaves the program state unchanged.

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

4.4 Sequential composition

The rule for sequential composition introduces an intermediate state between the composed commands.

$$\textsc{Seq}\ \frac{\Gamma, P, D\ \ \{c_1\}_M\ \ \Gamma', P', D' \qquad \Gamma', P', D'\ \ \{c_2\}_M\ \ \Gamma'', P'', D''}{\Gamma, P, D\ \ \{c_1; c_2\}_M\ \ \Gamma'', P'', D''}$$

4.5 Consequence

The CONSEQ rule is based on the consequence rules of Hoare logic [15]. It is typically required when applying rules for if and while instructions to ensure both branches result in the same state and to establish loop invariants, respectively. The rule allows for the state before a command to be strengthened, and the state after the command to be weakened. To express this, we introduce an ordering on the logic's state such that a stronger state captures all valid information flow exhibited under a weaker state.

$$\begin{aligned}
&\Gamma, P, D \geq_M \Gamma', P', D' \mathrel{\widehat{=}} (\forall \alpha \cdot D_W(\alpha) \supseteq D'_W(\alpha) \wedge D_R(\alpha) \supseteq D'_R(\alpha)) \wedge \\
&\quad P \implies P' \wedge \\
&\quad (\forall x \in \mathrm{dom}\ \Gamma \cdot \Gamma(x) \sqsubseteq \Gamma'(x)) \wedge : \\
&\quad wf\ M\ \Gamma\ P \implies wf\ M\ \Gamma'\ P'
\end{aligned}$$

For the D component, this ordering ensures all instructions in the stronger state have greater or equal write and read sets compared to the weaker. Therefore, the stronger state is aware of all instruction dependencies known in the weaker. 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. Finally, we constrain the preservation of well-formedness across the states, consequently preserving well-formedness across the CONSEQ rule.

$$\textsc{Conseq}\ \frac{\begin{array}{c}\Gamma_2, P_2, D_2 \geq_M \Gamma_1, P_1, D_1 \\ \Gamma_1', P_1', D_1' \geq_M \Gamma_2', P_2', D_2'\end{array} \qquad \Gamma_1, P_1, D_1\ \ \{c\}_M\ \ \Gamma_1', P_1', D_1'}{\Gamma_2, P_2, D_2\ \ \{c\}_M\ \ \Gamma_2', P_2', D_2'}$$

For example, if a required premise of a rule was $\Gamma, P, D\ \{skip\}_M\ \Gamma, P \wedge Q, D$, we could apply the CONSEQ rule to weaken the post-state's program state to P, i.e., to transform the premise to $\Gamma, P, D\ \{skip\}_M\ \Gamma, P, D$ which is readily discharged using the SKIP rule.

While the use of the CONSEQ rule requires user interaction, its application can be automated based on the context, e.g., through the introduction of a specialised rule for if instructions as in Murray et al. [28,29]. We introduce such specialised rules for if and while instructions in Sect. 7.

4.6 If

The rule for if statements restricts the branching condition to be Low to provide a timing-sensitive analysis as discussed in Sect. 2.1. The rule requires that both branches result in the

same context, which can be established using the CONSEQ rule.

$$\textsc{If}\ \frac{\Gamma, P_b \vdash b : Low \qquad \begin{array}{c}\Gamma, P \wedge [b]_M, D[b]\ \ \{c_1\}_M\ \ \Gamma', P', D' \\ \Gamma, P \wedge [\neg b]_M, D[b]\ \ \{c_2\}_M\ \ \Gamma', P', D'\end{array}}{\Gamma, P, D\ \ \{\mathsf{if}\ (b)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\}_M\ \ \Gamma', P', D'}$$

When demonstrating a Low classification for the boolean expression b, it is necessary to ensure a Low classification under all possible reorderings. This can be achieved by restricting P to those variables for which all writes are known to have occurred, $D_W(b)$, eliminating others through existential quantification of their free references in P. To facilitate this, we introduce an operation to restrict a predicate P to a set of variables V.

$$P|_V \ \widehat{=}\ \exists y_1, ...y_n \cdot P \ \text{ where } \{y_1, .., y_n\} = Var \setminus V$$

Therefore, given b can be shown to have a Low classification under $P|_{D_W(b)}$, then this outcome should hold for all possible reorderings. We employ the shorthand $P_\alpha \ \widehat{=}\ P|_{D_W(\alpha)}$ to represent this restriction in the rule.

Moreover, it is necessary to preserve well-formedness properties when manipulating the logic state. As we assume the pre-state (Γ, P) is well-formed, it is only necessary to enforce well-formedness on the modifications to the state, specifically, the conjunctions of b and $\neg b$. We introduce the notation $[b]_M \ \widehat{=}\ b|_{\mathsf{stable}\ M}$ to ensure only stable variables are referenced as required for well-formedness.

4.7 While

The rule for while statements, like that for if statements, restricts the loop condition to be Low to provide a timing-sensitive analysis. Moreover, the rule requires the pre-state to be a loop invariant that is maintained throughout all loop iterations. This restriction on the context can be established using the CONSEQ rule to set the context before the loop to be the loop's invariant. Outcomes of the boolean expression b are introduced into appropriate states restricted to maintain well-formedness.

$$\textsc{While}\ \frac{\Gamma, P_b \vdash b : Low \qquad \Gamma, P \wedge [b]_M, D[b]\ \ \{c\}_M\ \ \Gamma, P, D}{\Gamma, P, D\ \ \{\mathsf{while}\ (b)\ \mathsf{do}\ c\}_M\ \ \Gamma, P \wedge [\neg b]_M, D[b]}$$

4.8 Non-blocking loops

The logic is restricted to constant mode annotations M throughout a thread. Evidently, this is insufficient in the event that a thread gains stability or exclusive access to a variable under certain conditions. To illustrate, consider the example in Fig. 2, in which the $read$ operation temporarily gains stability on z to demonstrate an information-flow property: the algorithm speculates on z's stability, rolling back changes in the event that a $secret\_write$ operation interleaves. This temporary reliance on stability is common in non-blocking algorithms.

To enable support for such algorithms in the logic, we introduce a rule specialised for non-blocking loops, such as the outer loop in the $read$ operation. Such loops are annotated by the programmer with a set of variables v which we expect to be stable ($\{z\}$ in the running example). The annotation allows thread-local reasoning to assume that the nominated variables are stable

using the following rule (where c can be a while or do loop).

$$\textsc{NonBlocking}\ \frac{\begin{array}{c}\mathrm{dom}\ \Gamma_n = \mathrm{dom}\ \Gamma'_n = v \\ \forall x \in \mathrm{dom}\ \Gamma_n \cdot \Gamma_n(x) = \mathcal{R}_{P_{x=0}}(x) \\ \Gamma \cup \Gamma_n, P, D\ \ \{c\}_{M^v}\ \ \Gamma' \cup \Gamma'_n, P', D'\end{array}}{\Gamma, P, D\ \ \{c^v\}_M\ \ \Gamma', P', D'}$$

To evaluate the loop, Γ is updated with a value for each variable x in the set v based on what is known to have occurred if x were read; the (branch) condition $x = 0$ is used for this read. For $read$ in Fig. 2, Γ would be extended with a mapping from z to $\mathcal{R}_{P_{z=0}}(z)$. Since P before the loop is the initial value $True$, $\mathcal{R}_{P_{z=0}}(z) = \mathcal{R}_{True}(z)$ which is Low since $low_{True}(z)$ holds, i.e., $True \vdash \mathcal{L}(z)$ holds since $\mathcal{L}(z) = True$.

Also, M is replaced by M^v which extends M to include variables in v in the set of variables with mode $AssNoW$ and variables in $Global$ in the set of variables with mode $GuarNoW$. For $read$, we would have $z \in AssNoW$ and $\{z, x\} \in GuarNoW$. The extension of $GuarNoW$ in M^v ensures that, while in the loop, no writes can be made by the thread to any global variables. This is required in such non-blocking algorithms so that the execution can be discarded and restarted when one or more variables in v is discovered not to be stable.

For the rule to be sound, we require that the loop cannot be exited unless the variables in v are stable from the time that it is entered. This check requires reasoning about the functionality of the code and is outside of the scope of the logic (similar to the obligation that assumptions are matched by guarantees on other threads). In the case of $read$, the proof follows from the fact that the value of z is never decreased.

4.9 Non-control variable assignment

We introduce two assignment rules, distinguished based on writes to control and non-control variables. For an assignment $x := e$, where x is not a control variable, it is necessary to demonstrate a valid flow of information from the expression e to x. That is, the classification of x must be greater than the security level of the value being written. Similar to the If rule, we restrict P to those variables that are up-to-date from the perspective of $x := e$, denoted as $P_{x:=e}$. Therefore, the proof obligations are preserved under any reordering with write operations. This check is not required if modes prevent any other threads in the system from reading x.

$$\textsc{Assign}\ \frac{\begin{array}{c}x \notin \mathcal{C} \\ \Gamma, P_{x:=e} \vdash e : t \qquad x \notin M(AssNoRW) \implies t \sqsubseteq \mathcal{W}_{P_{x:=e}}(x)\end{array}}{\Gamma, P, D\ \ \{x := e\}_M\ \ \Gamma[x \mapsto t]_M, P[x := e]_M, D[x := e]}$$

The logic state is updated according to the assignment, and well-formedness enforced on the changes. Γ is updated to map x to its new type t derived from the expression e. To achieve this, we introduce the notation $\Gamma[x \mapsto y]_M$ to denote Γ updated so that x maps to y if x is a stable variable based on M. This operation ensures the domain of Γ remains equal to stable M.

We introduce $P[x := e]$ as a shorthand for the strongest postcondition of $x := e$ given a precondition P.

$$P[x := e] \mathrel{\widehat{=}} \exists v.\ P[v/x] \wedge x = e[v/x]$$

We extend this shorthand to $P[x := e]_M$ to represent the strongest postcondition restricted to stable variables based on M. This ensures the state only refers to variables in stable M ensuring well-formedness.

4.10 Control variable assignment

For an assignment $x := e$ where x is a control variable, it is necessary to consider both information flow and potential classification changes. As stated in Sect. 4.1, we constrain the security policy $\mathcal{L}$ such that control variables are always considered Low. Therefore, the instruction $x := e$ is only secure from an information-flow perspective if the expression e can demonstrate a Low classification.

We decompose the effects a control variable assignment may have on variable classifications 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-flow 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 falling classification based on the definitions introduced in Sect. 4.2.

$$falling_{P,P'}(x) = \{y : Var | x \in var(\mathcal{L}(y)) \wedge \neg low_P(y) \wedge \neg high_{P'}(y)\}$$

where P is the pre-state predicate, P' is the post-state predicate, and x the modified variable. Note that it is necessary to consider variables that are $\neg low_P$ and $\neg high_{P'}$, rather than $high_P$ and $low_{P'}$ respectively. This captures those variables for which we cannot determine their classification due to the potential unknown outcomes of low_P and $high_{P'}$. For example, if $\mathcal{L}(x)$ were $z = 0$ and P were $z < 10$ then x might be $High$. In this case, $high_P$ would be $False$ (as it is not possible to deduce $\neg\mathcal{L}(x)$). However, $\neg low_P(x)$ would be $True$ (as it is also not possible to deduce $\mathcal{L}(x)$).

We need to ensure that all falling variables hold low information. Consider a variable y, such that the assignment $x := e$ results in its classification falling. This is secure if y holds a Low value prior to the classification change. Hence we require $\Gamma(y) = Low$ at the point of the classification change. However, this is not enough. Instruction reordering may result in the corresponding write of Low information to y and the assignment $x := e$ being reordered. Consequently, y's classification may fall without its $High$ information being cleared. To account for this, it is also necessary to consult $D_W(x := e)$ to ensure all writes to y have definitely occurred.

Alternatively, if the falling variable y is in the $M(AssNoRW)$ set, then changes to its classification are inconsequential as no other thread can read its value. The required condition for falling variables is captured below.

$$falling_{P,P'}(x) \subseteq M(AssNoRW) \cup (\{y : Var | \Gamma(y) = Low\} \cap D_W(x := e)\})$$

Next we consider the rising case. This 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 $out := y; x := e$, 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 $x := e; out := y$ enabling a write of $High$ information to out if y's classification rises after $x := e$ is executed (e.g. if some other thread places classified data into y in between the assignment to x and the assignment of out).

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

$$rising_{P,P'}(x) = \{y : Var | x \in var(\mathcal{L}(y)) \wedge \neg high_P(y) \wedge \neg low_{P'}(y)\}$$

Given a variable y in the rising set, it is necessary to show that there are no reorderable reads of the variable. This can be achieved using the dependency analysis on reads, ensuring y is in $D_R(x := e)$.

$$rising_{P,P'}(x) \subseteq D_R(x := e)$$

We merge these falling and rising proof obligations into a single property.

$$\begin{aligned} &secure\_update_{\Gamma,P,U,V,M}(x := e) \mathrel{\widehat{=}} \\ &\quad falling_{P_1,P_2}(x) \subseteq M(AssNoRW) \cup (\{y : Var | \Gamma(y) = Low\} \cap U\}) \wedge \\ &\quad rising_{P_1,P_2}(x) \subseteq V \end{aligned}$$

where P_1 is the pre-state predicate $P|_U$ and P_2 is the post-state predicate $P[x := e]|_V$. We define this property in terms of variable sets U and V, representing the sets of up-to-date writes and reads respectively, rather than consulting D based on $x := e$ to enable the reuse of the definition of $secure\_update$ in the CAS rule of Sect. 4.13.

Note that it is possible for a variable to be in both the rising and falling sets, in the event that its classification is unknown in both the pre-state and post-state. The following rule is sufficient to demonstrate preservation of the security policy due to a control variable assignment.

$$\textsc{AssignC}\ \frac{x \in \mathcal{C} \qquad \Gamma, P_{x:=e} \vdash e : Low \qquad secure\_update_S(x := e)}{\Gamma, P, D \quad \{x := e\}_M \quad \Gamma[x \mapsto Low]_M, P[x := e]_M, D[x := e]}$$

where $S \mathrel{\widehat{=}} \Gamma, P, D_W[x := e], D_R[x := e], M$

4.11 Fences

ARMv8 supports a variety of fences. Below, we list the most important of these fences and their properties.

- A full fence (DMB or DSB in ARM) ensures all instructions before it take effect in memory before any instruction after it.
- A store fence (DMB.ST or DSB.ST in ARM) ensures all store instructions before it take effect in memory before any store instructions after it.
- A control fence (ISB in ARM) can be placed between a branch instruction and following loads to prevent the loads being speculatively executed. That is, a branch before a control fence cannot be reordered with it, and a load after a control fence cannot be reordered with it.

Fig. 4 Writer and reader threads using the operation secret_write and read of Fig. 3

```
writer_thread:
1  z := 0;
2  x := 0;
3  while (true)
4      z := z+1;
5      fence;
6      x := secret;
       ...
7      x := 0;
8      fence;
9      z := z+1

reader_thread:
10 while(true)
11     do
12         do
13             r1:= z;
14         while (r1 % 2 ≠ 0)
15         fence;
16         r2 := x;
17         fence;
18     while (z ≠ r1)
19     y := r2
```

Support for fences depends on suitable definitions for the $\overset{R}{\Leftarrow}$ relation. For the fences above, this relation is as follows [8].

$$\begin{aligned} fence &\overset{R}{\Leftarrow} \alpha \\ \alpha &\overset{R}{\Leftarrow} fence \\ x := e &\overset{R}{\Leftarrow} fence.st \quad \text{if } x \in Global \\ fence.st &\overset{R}{\Leftarrow} x := e \quad \text{if } x \in Global \\ [b] &\overset{R}{\Leftarrow} cfence \\ cfence &\overset{R}{\Leftarrow} x := e \end{aligned}$$

Certain invariants for the D_W and D_R sets can be derived from these definitions. For example, in the case of a $fence$, updates to D will never remove writes and reads from $D_W[fence]$ and $D_R[fence]$ respectively, as no operation can reorder with $fence$. Consequently, given D is initialised to map all instructions to Var, the invariant $D_W(fence) = D_R(fence) = Var$ will be maintained across all instructions. Therefore, whenever applying an update due to a $fence$, D is reset to initial conditions as all instructions will map via D_W and D_R to Var. A similar property can be seen for $fence.st$ as an action writing to a $Global$ variable will never reorder with it. As a result, $D_W(fence.st) \supseteq Global$ will be maintained across all instructions.

As these instructions do not manipulate the predicate P or classification context Γ, their information-flow properties are trivial. As we prevent branching based on $High$ information, it is not possible to introduce a side-channel based on the conditional execution of a fence (as in [39]).

$$\textsc{Fence}\ \frac{\alpha \in \{\mathsf{fence}, \mathsf{fence.st}, \mathsf{cfence}\}}{\Gamma, P, D \ \ \{\alpha\}_M \ \ \Gamma, P, D[\alpha]}$$

4.12 Example revisited

We now have enough of the logic to illustrate its application to the example of Fig. 3. Two threads which call the $secret\_write$ and $read$ operations are shown in Fig. 4.

The sequential composition rule allows us to step through a program one line at a time. The values of Γ, P and D following a given line can be calculated from the applied rule. If we reach a line of code that no rule can be applied to, this indicates a potential security leak. For example, consider the $writer\_thread$ in Fig. 4 for which we will assume $M(AssNoW) = \{z, x\}$. This thread initialises the variables z and x and then repeatedly calls the $secret\_write$ operation of Fig. 3. Applying rules AssignC and Assign to lines 1 and 2, respectively, shows

that the code up to line 2 is secure. Following line 2, we have $\Gamma = \{z \mapsto Low, x \mapsto Low\}$, $P = (z = 0 \wedge x = 0)$, $D_W(z := z + 1) = \{z\}$, $D_W(x := secret) = D_W(x := 0) = \{x\}$ and $D_R(z := z + 1) = D_R(x := secret) = D_R(x := 0) = \{x, z\}$.

The CONSEQ rule can then be applied to weaken P to $z\,\%\,2 = 0 \wedge x = 0$ and leave Γ and D unchanged. These values become the starting point for evaluating lines 4 to 9. We can show that these lines are also secure by applying rules ASSIGNC, FENCE and ASSIGN. Note that without the fence at line 5, z would not be a member of $D_W(x := secret)$ and hence not in $P_{x:=secret}$. Therefore, ASSIGN would not be applicable (since, with $\mathcal{L}(x) = (z\%2 = 0)$, the value of z is required to be odd for this assignment to be secure). Hence, no rule would be applicable for line 6. This demonstrates how the leak of x would be detected by the logic if lines 4 and 6 could be reordered.

Similarly, without the fence at line 8, no rule would be applicable to line 9. In this case, since z becomes even at line 9, the variable x must hold Low data to satisfy $secure\_update$. This could not be ascertained, however, since x would not be in $D_W(z := z + 1)$. This demonstrates how the leak of x would be detected by the logic if lines 7 and 9 could be reordered.

The reasoning for the $reader\_thread$ is similar. The most interesting aspect of it is the use of the NON- BLOCKING rule. The reason that the $reader\_thread$ is secure, is that it only reaches line 19 when z is stable from line 13 (when it is assigned to $r1$) until line 18 (where it is checked to be equal to $r1$). The algorithm works on the principle that there is a high chance of z being stable while these lines are executed, and hence the $reader\_thread$ will reach line 19 without too many iterations of the outer do-loop. Hence, we annotate the outer-do loop with the set $\{z\}$ so that the NON- BLOCKING rule can be applied.

4.13 Compare-and-swap

A compare-and-swap CAS instruction is an atomic operation for updating a variable based on its current value. $r := \mathsf{CAS}(g, l_1, l_2)$ updates the $Global$ variable g to the value of expression l_2 if $g = l_1$, and otherwise leaves g unchanged. The local variable r records whether or not the write occurred. Due to the atomic nature of the instruction, we restrict expressions l_1 and l_2 to only refer to $Local$ variables.

The rule is structured as a composition of the IF and ASSIGN rules. We assume the action of conditionally modifying g may introduce a timing side-channel, and therefore restrict the boolean expression $g = l_1$ to be Low.

Moreover, it is necessary to demonstrate that g can hold the value of l_2, given $g = l_1$. To account for this, we introduce an intermediate predicate $P' \mathrel{\widehat{=}} P \wedge g = l_1$, capturing the state where a write occurs, and use that state to determine the classification of g. Note that we do not need to ensure P' is well-formed due to the atomic nature of the CAS.

Finally, it is necessary to consider the implications of classification changes if g is a control variable. We reuse the definition of $secure\_update$ to enforce these constraints. Note that r is a $Local$ variable and, therefore, cannot be a control variable.

The rule for CAS instructions is as follows where $op \mathrel{\widehat{=}} r := \mathsf{CAS}(g, l_1, l_2)$.

$$\text{CAS}\ \frac{\begin{array}{cc}\Gamma, P_{op} \vdash g = l_1 : Low & g \notin M(AssNoRW) \implies t \sqsubseteq \mathcal{W}_{P'_{op}}(g) \\ \Gamma, P'_{op} \vdash l_2 : t & secure\_update_S(g := l_2)\end{array}}{\Gamma, P, D \quad \{op\}_M \quad \Gamma[g \mapsto t, r \mapsto Low]_M, P'', D[op]}$$

where P' is the intermediate predicate described above, P'' is a post-state predicate described below, and $S \mathrel{\widehat{=}} \Gamma, P', D_W[op], D_R[op], M$.

Γ, P and D are updated according to the semantics of the operation. The Γ update captures assignments to both g and r. For r, this is trivial, as it encodes the outcome of the comparison $g = l_1$, which must be Low. The value of g varies depending on the outcome of $g = l_1$. Therefore, it is necessary to determine both possible security levels of the value held in g and take the highest classification, as done in the MERGEIF rule. When $g = l_1$ is false, the value of g remains unmodified. As $g = l_1$ is restricted to being Low, in this case $\Gamma(g) = Low$. Otherwise, when $g = l_1$ is true, the value of g corresponds to t, the type of l_2. The final type of g is the highest of the two, which is trivially t.

To compute the post-state predicate P'', we introduce a ternary operator $b\,?\,e_1 : e_2$, evaluating to e_1 when b holds, and e_2 otherwise. Moreover, we introduce the following pairwise assignment operator given the two written variables are distinct.

$$P[(x, y) := (e_1, e_2)] \mathrel{\widehat{=}} \\ \exists v_1, v_2.\ P[v_1/x][v_2/y] \wedge x = e_1[v_1/x][v_2/y] \wedge y = e_2[v_1/x][v_2/y]$$

In the context of the CAS instruction, we can show $g \neq r$ as g is $Global$, while r is $Local$. P'' is hence defined as follows where we restrict the free variables of the resulting predicate to those that are in stable M to preserve well-formedness.

$$P'' \mathrel{\widehat{=}} P[(g, r) := (g = l_1\,?\,l_2 : g, g = l_1)]_M$$

To facilitate updates to D, it is necessary to define the reordering rules for a CAS instruction. To ensure a sound analysis, we introduce the weakest reordering constraints for a CAS based on its sub-instructions. That is, we allow the most possible reorderings and hence the greatest scope for information leaks.

For forwarding, we always apply the conditional store $g := l_2$ as it will always remove reordering constraints. Inversely, we do not forward the store to r as this introduces references to g preventing additional reorderings.

$$x := e_{\langle r := \mathsf{CAS}(g, l_1, l_2)\rangle} = x := e[l_2/g]$$

For the reordering relation, the CAS is only guaranteed to load g and evaluate l_1 storing the result in r. Therefore, we define the reordering relation to be consistent with that for these instructions.

$$\alpha \overset{R}{\not\Leftarrow} r := \mathsf{CAS}(g, l_1, l_2) \qquad \text{if } \alpha \overset{R}{\not\Leftarrow} r := (g = l_1) \\ r := \mathsf{CAS}(g, l_1, l_2) \overset{R}{\not\Leftarrow} \alpha \qquad \text{if } r := (g = l_1) \overset{R}{\not\Leftarrow} \alpha$$

4.14 Load-acquire/store-release

ARMv8 supports load and store instructions with acquire/release memory orderings. The load-acquire $\mathsf{acq}(r, g)$ operation loads a $Global$ variable g into the $Local$ variable r and ensures no memory operations later in program order may reorder before it. The store-release $\mathsf{rel}(g, l)$ operation stores the local expression l to the $Global$ variable g and ensures all memory operations earlier in program order have been completed. These operations may be forwarded, using the corresponding forwarding definitions for their traditional assignment forms, $g := l$ and $r := g$ for $\mathsf{rel}(g, l)$ and $\mathsf{acq}(r, g)$ respectively. We introduce appropriate

$\overset{R}{\Leftarrow}$ definitions for these instructions below.

$$\begin{aligned} \mathsf{acq}(r, g) &\overset{R}{\Leftarrow} \alpha \\ \alpha &\overset{R}{\Leftarrow} \mathsf{acq}(r, g) && \text{if } \alpha \overset{R}{\Leftarrow} r := g \\ \alpha &\overset{R}{\Leftarrow} \mathsf{rel}(g, l) \\ \mathsf{rel}(g, l) &\overset{R}{\Leftarrow} \alpha && \text{if } g := l \overset{R}{\Leftarrow} \alpha \end{aligned}$$

We support these instructions as variants of the traditional assignment rules, but with modified reordering relations. This results in different restrictions on P and modified updates to D for the instructions. First, we introduce a rule for $\mathsf{rel}(g, l)$ where $g \notin \mathcal{C}$.

$$\textsc{Release}\ \frac{\begin{array}{c} g \notin \mathcal{C} \\ \Gamma, P_{\mathsf{rel}(g,l)} \vdash l : t \qquad g \notin M(AssNoRW) \implies t \sqsubseteq \mathcal{W}_{P_{\mathsf{rel}(g,l)}}(g) \end{array}}{\Gamma, P, D \quad \{\mathsf{rel}(g, l)\}_M \quad \Gamma[g \mapsto t]_M, P[g := l]_M, D[\mathsf{rel}(g, l)]}$$

A similar definition is possible in the event $g \in \mathcal{C}$.

$$\textsc{ReleaseC}\ \frac{\begin{array}{c} g \in \mathcal{C} \\ \Gamma, P_{\mathsf{rel}(g,l)} \vdash l : Low \qquad secure\_update_S(g := l) \end{array}}{\Gamma, P, D \quad \{\mathsf{rel}(g, l)\}_M \quad \Gamma[g \mapsto t]_M, P[g := l]_M, D[\mathsf{rel}(g, l)]}$$

where $S \mathrel{\widehat{=}} \Gamma, P, D_W[\mathsf{rel}(g, l)], D_R[\mathsf{rel}(g, l)], M$.

Load-acquire instructions load a *Global* variable into a *Local*. Since *Local* variables are always classified $High$ and cannot be control variables, such an assignment can never result in an information-flow violation or a change in classification. Therefore, the rule is a simplified variant of the ASSIGN rule.

$$\textsc{Acquire}\ \frac{\Gamma, P_{\mathsf{acq}(r,g)} \vdash g : t}{\Gamma, P, D \quad \{\mathsf{acq}(r, g)\}_M \quad \Gamma[r \mapsto t]_M, P[r := g]_M, D[\mathsf{acq}(r, g)]}$$

4.15 Arrays

We introduce limited support for modelling arrays to the logic. This allows us to model address shifting; something which is common in assembly-level programs. We constrain arrays to have a known, static length. Moreover, it is assumed that the arrays all refer to distinct memory regions and remain allocated throughout execution.

To enable reuse of existing logic properties and rules, arrays are modelled as a collection of variables. We introduce the notation A_n to refer to the nth element of array A, where n is restricted to a valid index in A's domain and must be an integer constant. As traditional *Global* variables, they may appear anywhere in the logic state or program that a *Global* would be expected. For example, they may appear as control variables and have tracked classifications in Γ.

We let the expression $A[l]$ refer to an access to A based on the result of l. To support logic rules over such array accesses, we must resolve the result of $A[l]$ into the form A_n. As the outcome of l may be unknown, this may result in multiple possible resolved variants of the instruction. If this is the case, all variants must be proved secure.

To enable such a judgement, we first introduce a function to resolve array accesses in an expression, $resolve_P$. The function returns the set of possible instructions coupled with a predicate P constraining the index expression. The function is defined by a recursive traversal

over expressions with global references. We include representative definitions of this function below, where $\oplus$ is an arbitrary binary operator.

$$
\begin{aligned}
&resolve_P(A[l]) := \{(P \wedge l = n, A_n) | n \in \text{dom } A \wedge \neg(P \vdash l \neq n)\} \\
&\quad resolve_P(x) := \{(P, x)\} \\
&\quad ... \\
&\quad resolve_P(e_1 \oplus e_2) := \\
&\quad \{(P_2, e_1' \oplus e_2') | (P_1, e_1') \in resolve_P(e_1) \wedge (P_2, e_2') \in resolve_{P_1}(e_2)\} \\
&\quad ... \\
&\quad resolve_P(x := e) := \\
&\quad \{(P_2, x' := e') | (P_1, x') \in resolve_P(x) \wedge (P_2, e') \in resolve_{P_1}(e)\} \\
&\quad resolve_P(r := \mathsf{CAS}(g, l_1, l_2)) := \\
&\quad \{(P', r := \mathsf{CAS}(g', l_1, l_2)) | (P', g') \in resolve_P(g)\}
\end{aligned}
$$

Note that the $resolve_P$ function will reject impossible indices based on P via the $\neg(P \vdash l \neq n)$ test.

Moreover, it is necessary to demonstrate all array accesses are based on Low information. If this is not the case, cache and reordering effects may be influenced by $High$ data, potentially introducing side-channels. Thus our logic enforces a *constant-time* security guarantee [2] (which we return to later in Sect. 5.3). To establish this, we collect all array access expressions for an instruction using a function $indices$.

$$
\begin{aligned}
&indices(A[l]) := \{l\} \\
&indices(x) := \emptyset \\
&... \\
&indices(e_1 \oplus e_2) := indices(e_1) \cup indices(e_2)
\end{aligned}
$$

Given these definitions, we introduce a generic rule for supporting any instruction in set a (defined at the beginning of Sect. 4) in which an unresolved array access may be found.

$$
\textsc{Array}\ \frac{\begin{array}{c}\alpha \in a \\ \forall i \in indices(\alpha) \cdot P, \Gamma \vdash i : Low \\ \forall (P', \beta) \in resolve_P(\alpha) \cdot \Gamma, P', D \quad \{\beta\}_M \quad \Gamma', P'', D'\end{array}}{\Gamma, P, D \quad \{\alpha\}_M \quad \Gamma', P'', D'}
$$

This general rule can be specialised to improve automation. For example, if the $resolve$ function returns only one possible instruction, e.g., if it were given $A[0] := 1$, then only a single judgement must be shown. Moreover, the post-state of the Array rule would be equivalent to the post-state of the singular possible access. Otherwise, it is necessary to demonstrate valid judgements for all possible instructions and merge their post-states. A specialised rule may employ a similar strategy to the IfMerge rule (introduced later in Sect. 7) in which the highest Γ mapping for all variables is maintained; the disjunction of all possible predicates is taken; and D consists of the intersection of all possible Ds.

This technique has been employed in the symbolic execution tool (see Sect. 7). Evidently, it does not scale well with large arrays due to the significant increase in necessary judgements. However, security properties are typically independent of array length. Therefore, properties shown over small arrays will in many cases hold over those of any size.

4.16 Weaker memory model concepts

ARM and POWER processors employ a number of additional techniques to reorder memory operations and improve performance. These techniques can be seen as a weakening of the constraints (i) to (iv), under particular situations. To account for these, we weaken the $\overset{R}{\Leftarrow}$ relation accordingly, resulting in fewer up-to-date writes and reads in D.

The first optimisation is referred to as *squashing* and is capable of weakening constraint (i). Constraint (i) prevents repeated writes to the same variable from reordering, such as $x := 2; x := 5$. However, the processor can squash the first write, skipping it entirely. This is not an immediate issue for the information-flow logic as such a behaviour is a subset of possible thread interleavings where no other thread interleaves to read the first instance, although it can weaken instruction dependencies.

To illustrate, consider $r := x; c := r; c := 1; l := r$, where x is controlled by c and only Low information must be written to l. If squashing were not possible, the assignments to c would be restricted to execute after the read of x due to the dependency via r. Therefore, it would not be necessary to consider potential rises in x's classification due to these operations. However in the presence of squashing, the first write to c may be skipped, eliminating the dependency via r. This enables the execution $c := 1; r := x; l := r$, in which $c := 1$ may result in x's classification becoming $High$ and subsequently resulting in a flow of $High$ information to l. To account for this case, we weaken $\overset{R}{\Leftarrow}$ between assignments by removing constraint (i).

Processors may also weaken constraint (iii), which prevents reordering of $x := e$ and $y := f$ given y is referenced in e. This remains true if y is a $Global$ variable, as the processor must read its value before modifying it. However if y is a $Local$ variable, this represents a false dependency between the two operations as their outcomes are only linked due to register reuse. The processor may decide to *rename* the use of y in $y := f$ and future operations to another $Local$ variable, thus breaking the false dependency between the two operations. To account for this, constraint (iii) only applies to cases where y is a $Global$ variable.

Finally, processors may weaken constraint (iv), which prevents reordering of $x := e$ and $y := f$ given e and f refer to the same $Global$ variables, via *load speculation*. Given two memory load operations, where determining the first memory address requires a costly computation relative to the second, the processor may speculate that the two addresses are distinct and perform the second earlier. If this speculation does not hold, the processor must ensure the two loads behave as if constraint (iv) held, potentially rolling back operations. Consequently, any instructions executed due to the speculation cannot have effects observable to other threads, and therefore cannot involve $Global$ writes. As the proposed logic is oblivious to the order of $Global$ reads, reorderings introduced due to load speculation cannot introduce an information-flow violation. Therefore, constraint (iv) can be preserved.

We restate the weakened $\overset{R}{\Leftarrow}$ relation between assignments below.

$$
\begin{aligned}
x := e &\overset{R}{\Leftarrow} y := f && \text{(ii) if } f \text{ refers to } x \\
x := e &\overset{R}{\Leftarrow} y := f && \text{(iii) if } e \text{ refers to } y \text{ and } y \text{ is a } Global \text{ variable} \\
x := e &\overset{R}{\Leftarrow} y := f && \text{(iv) if } e \text{ and } f \text{ refer to the same } Global \text{ variables} \\
x := e &\overset{R}{\Leftarrow} y := f && \text{otherwise}
\end{aligned}
$$

5 Soundness

Our logic has been encoded in Isabelle/HOL [30] and proven sound with respect to a definition of value-dependent non-interference suitable for compositional reasoning [29]. We use formalisation techniques derived from a series of prior logic encodings [23,28,29], in which a successful application of the logic's rules, along with suitable initial conditions, are shown to establish a strong bisimulation over a pair of executions of a thread. This bisimulation preserves the desired security property between its constituent pairs, expressed as a form of low-equivalence. Low-equivalence is a property that constrains all variables classified as Low to be equal in both executions. As a result, it is not possible to distinguish two low-equivalent executions via inspection of only Low variables. The bisimulation is also closed under global modifications, such that interference from parallel threads may not invalidate local reasoning. Finally, given compatible thread specifications, these local bisimulations can be composed to establish a security property across a concurrent system.

The formalisation builds directly on the encoding of CoVern [28], preserving its definitions of security and compositionality, whilst replacing its language and logic rules with those detailed in Sect. 4. Encoding decisions are made to minimise modifications to these existing theories.

The theories files are available at https://bitbucket.org/wmmif/wmm-if, along with a series of applications of the logic to small examples. These snippets illustrate the difficulty of applying the logic rules in Isabelle/HOL, motivating the external automation described in Sect. 7.

5.1 Compositional Security

To successfully reuse the existing compositional security theory, it is necessary to prove that the logic's rules establish a thread-local bisimulation with a series of properties. We briefly summarise these properties to motivate verification effort, with a full description of the underlying theory available in foundational work [28].

First, the bisimulation must be preserved across thread-local operations, as is standard for a bisimulation definition. Moreover, the bisimulation must be symmetric, to prevent any distinction between the two executions. These two properties together constitute a *strong bisimulation*.

Definition 1 (*Strong Bisimulation)*

$$\begin{aligned}
&\mathsf{sbisim}\ \mathcal{B} \equiv (\forall s, s' \cdot s\ \mathcal{B}\ s' \implies s'\ \mathcal{B}\ s) \wedge \\
&\quad \forall c_1, mem_1, c_2, mem_2 \cdot \langle c_1, mem_1\rangle\ \mathcal{B}\ \langle c_2, mem_2\rangle \implies \\
&\qquad \forall c_1', mem_1' \cdot \langle c_1, mem_1\rangle \rightarrow \langle c_1', mem_1'\rangle \implies \\
&\qquad\quad \exists c_2', mem_2' \cdot \langle c_2, mem_2\rangle \rightarrow \langle c_2', mem_2'\rangle \wedge \langle c_1', mem_1'\rangle\ \mathcal{B}\ \langle c_2', mem_2'\rangle
\end{aligned}$$

where $\langle c, mem\rangle$ represents a pair of program and memory state, $s\ \mathcal{B}\ s'$ represents a bisimulation $\mathcal{B}$ that relates the states s and s', and $s \rightarrow s'$ represents a transition from state s to s' via a thread-local operation.

The bisimulation must enforce low-equivalence on bisimilar memory states. We define this relation in terms of the security policy $\mathcal{L}$, ensuring equivalence between the two memories for any variables considered to be Low. Recall that a variable is considered Low whenever its security policy evaluates to $true$. The low-equivalence definition may evaluate these

policies over either of the two bisimilar memories, as the Low classification constraint on control variables ensures equivalent value-dependent classifications. Moreover, the low-equivalence definition explicitly allows for unreadable variables to hold arbitrary data, as they cannot influence other threads. This excludes control variables to preserve the aforementioned symmetry.

Definition 2 (*Low Equivalence*)

$$mem_1 \approx_M mem_2 \equiv \\ \forall x \notin M(AssNoRW) - \mathcal{C} \cdot mem_1 \in \mathcal{L}\, x \implies mem_1\, x = mem_2\, x$$

where M refers to the variable modes for the current thread, and $m \in P$ represents evaluation of the predicate P to $true$ for a memory m.

Finally, to enable compositional reasoning, the bisimulation must be closed under global modifications that satisfy the variable modes and preserve low-equivalence. This is formalised by quantifying over all possible low-equivalent memories, mem'_1 and mem'_2, constrained such that they do not modify any variables assumed to be stable. To be closed under global operations, these new memories must be considered bisimilar.

Definition 3 (*Closed under Global Modifications*)

$$\mathsf{closed}_M\ \mathcal{B} \equiv \forall c_1, mem_1, c_2, mem_2 \cdot \langle c_1, mem_1\rangle\ \mathcal{B}\ \langle c_2, mem_2\rangle \implies \\ \forall mem'_1, mem'_2 \cdot mem'_1 \approx_M mem'_2 \implies \\ (\forall x \in \mathsf{stable}\ M \cdot mem_1\, x = mem'_1\, x \wedge mem_2\, x = mem'_2\, x) \implies \\ \langle c_1, mem'_1\rangle\ \mathcal{B}\ \langle c_2, mem'_2\rangle$$

5.2 Compositionality theorem

Given $\langle c_1, mem_1\rangle\ \mathcal{B}\ \langle c_2, mem_2\rangle \implies mem_1 \approx_M mem_2$, $\mathsf{sbisim}\ \mathcal{B}$ and $\mathsf{closed}_M\ \mathcal{B}$, a formal *compositionality theorem* states how to establish a global bisimulation across a series of threads with compatible modes. That global bisimulation guarantees security for the parallel composition of the threads. This global bisimulation enforces low-equivalence and satisfies a variant of sbisim with thread-local state transitions replaced by interleaved thread operations (parallel composition). The order of the interleaving is defined by a fixed but arbitrary *schedule* shared by both executions, as in prior work [28,29], thereby making scheduling deterministic. Consequently, we adopt the familiar assumption [28,29] that scheduling decisions are not influenced by $High$ information.

At a high level, the proof of the compositionality theorem is structured as an induction over the scheduler trace, using the sbisim and low-equivalence properties for the executing thread to preserve both its local bisimulation and the global notion of low-equivalence, whilst the closed_M properties for all other threads are used to preserve their own bisimulations.

Note that this proof relies on all threads conforming to their guaranteed variable modes as well as compatibility of these modes between threads. While this would be trivial for the proposed logic, as modes do not change throughout the execution of a thread, demonstrations of sound mode use and compatibility have been excluded from the verification, as done in prior formalisations [23,29].

5.3 Semantics

Colvin and Smith [8] provide an operational semantics of ARMv8 which has been validated against approximately 10,000 litmus tests developed by Alglave et al. [1]. Specifically, the semantics has been compared with running these tests on actual hardware. Its definition is similar to that of a standard small-step semantics for a Kleene algebra, with the introduction of instruction reordering based on the behaviours introduced in Sect. 3. Additionally, weak memory behaviours are modelled at a thread-local scope, enabling a traditional thread interleaving interpretation of parallel composition. Consequently, verification against such a semantics enables reuse of existing compositionality theorems and language structures, in contrast to alternative weak memory semantics that rely on axiomatic approaches [1] or significantly modify the state encoding [17].

Notably, this encoding introduces non-determinism into the language to capture the various reordering and speculation choices that can be made during execution. However, the underlying bisimulation theory requires a deterministic language to ensure bisimilar programs observe the same instruction traces. Consequently, our language encoding includes a reordering schedule, det, that determines when instruction reordering and speculation takes place, re-establishing deterministic behaviour.

The schedule is encoded as a list of L and R values, as the semantics only features two possible choices for each language structure. For instance, $\alpha; c$ may chose to execute α, encoded as L in the schedule, or reorder an instruction later in c before α, encoded as R. Note that the semantics also features non-deterministic choice to support if with speculation, which makes similar use of det to establish deterministic behaviour.

As a result, our bisimulation only establishes low-equivalence between two program executions in which the same reordering behaviours occur. This could result in an information leak if the underlying hardware performs reordering decisions based on potentially classified information and such decisions are observed by an attacker. For example, this can occur when reordering array operations, as their indexing calculations may prevent or allow reordering depending on whether they evaluate to equivalent indices. We eliminate this side-channel by constraining such indexing operations to be based on public information, as detailed in Sect. 4.15, thus enforcing a constant-time security guarantee [2]. In general, we observe that constant-time security appears to be necessary to ensure the absence of leakage via reordering effects—an observation that, despite much prior work on noninterference for weak memory models, we believe is novel.

We assume that, under the enforcement of constant-time security, the hardware will not reorder based on secret information and, hence, it is safe to reason under all possible deterministic reorderings, as captured by det.

Additionally, the semantics models reordering in the presence of branching language structures, such as if and while, via refinement to a trace of instructions with appropriate guards, as detailed in Sect. 4. We refer to a program that has been fully refined to a trace of instructions as flat. For example, the program $\mathsf{if}\ (b)\ \mathsf{then}\ \alpha\ \mathsf{else}\ \gamma$ could be silently rewritten to $[b]; \alpha$, modelling speculation of b at the hardware level. Consequently, if α can reorder with $[b]$ then it may execute prior to the evaluation of the if's condition. If the later evaluation of the if's condition fails, e.g., $[b]$ does not evaluate to true, then the speculation failed, triggering a rollback that reverts the effects of α at the hardware level.

The semantics does not model this rollback behaviour and considers a trace with failed speculation *magic*. Consequently, these traces are ignored and it is assumed that only successful speculation cases are observed. This is formalised by ensuring a trace exists that satisfies all speculated guards at each execution step. As we do not model rollback behaviours, we must

assume the hardware implements a valid rollback implementation that completely reverts the speculated actions. As demonstrated in other work [18], such an assumption may not hold on modern hardware, as speculated operations may be observed across a rollback via the cache. We leave such issues to future work.

We introduce the operation refine $c\ det$ to resolve all non-deterministic program structures in c up to the next instruction chosen for execution, based on det. Moreover, we capture the possibility of an action α reordering prior to a program c via a new definition $\alpha' < c < \alpha$, where α can reorder with all instructions in c and their cumulative forwarding effects produce the instruction α'.

Lemma 1 (Program Split) *Given a transition of the form* $(c, det) \rightarrow_\alpha (c', det')$*, the program* c *must* refine *to a program* $c_a; \alpha'; c_b$ *based on* det*, such that* c_a *is* flat *and* $\alpha < c_a < \alpha'$*. Additionally,* c' *must be equivalent to the remaining program* $c_a; c_b$*.*

$$(c, det) \rightarrow_\alpha (c', det') \implies \exists c_a, c_b, \alpha' \cdot$$
$$\mathsf{refine}\ c\ det = c_a; \alpha'; c_b \wedge \mathsf{flat}\ c_a \wedge \alpha < c_a < \alpha' \wedge c' = c_a; c_b$$

We prove Lemma 1 via structural induction over the program semantics. All cases resolve trivially, as the property definitions in the consequent closely resemble the structure of the semantics. This lemma is crucial for the soundness proof, as it allows for the decomposition and recomposition of logic judgements over the subprograms.

5.4 Logic

We encode all rules seen in this paper, with the exception of the CAS, acq and rel instructions which are not covered by the semantics, and the NonBlocking rule whose soundness relies on properties of non-blocking algorithms which fall outside the logic. For the three instructions not covered by the semantics, our rules are based on other rules (namely, those for if statements and assignments) which have been proven sound.

A deeply embedded predicate language is used to encode the memory state, P, and proof obligations. This deep embedding facilitates the variable-based queries and operations seen in the rules, such as determining free variables in a security policy and performing existential quantification for a set of variables. Γ is encoded as a partial map from variables to their classifications, either Low or $High$. To more closely reflect an executable implementation of the logic, the formalisation encodes an over-approximation of the W and R update operations seen in Sect. 3.2. This implementation reduces the domain of these mappings from all possible instructions to reads and writes of individual variables and approximates the former based on the later. Details of this implementation are included in Sect. 7.

The formalisation encodes a set of core logic rules, from which all others are derived. This core set covers all supported instruction types, such as assignments, fences and guard operations. Additionally, it includes the Seq and Conseq rules detailed in Sect. 4, allowing for composition and rewriting of logic judgements. Finally, the core set includes rules for Kleene algebra operations representing non-deterministic choice and iteration, as illustrated in Fig. 5.

Note that the core set does not include rules for language structures such as if and while. As the semantics handles guards and control flow as separate concepts, it is simpler to develop rules for each and then consider their composition to verify these language structures. For example, recall the rule for if (b) then c_1 else c_2, detailed in Sect. 4.6, which can be rewritten as $([b]; c_1) \sqcap ([\neg b]; c_2)$. This rule requires $\Gamma, P_b \vdash b : Low$, which implies

$$\frac{\Gamma, P, D \ \{c_1\}_M \ \Gamma', P', D' \quad \Gamma, P, D \ \{c_2\}_M \ \Gamma', P', D'}{\Gamma, P, D \ \{c_1 \sqcap c_2\}_M \ \Gamma', P', D'} \qquad \frac{\Gamma, P, D \ \{c\}_M \ \Gamma, P, D}{\Gamma, P, D \ \{c^*\}_M \ \Gamma, P, D}$$

Fig. 5 Rules for Choice and Iteration

$\Gamma, P_b \vdash \neg b : Low$. These properties are sufficient to establish $\Gamma, P, D\ \{b\}\ \Gamma, P \wedge b, D[b]$ and $\Gamma, P, D\ \{\neg b\}\ \Gamma, P \wedge \neg b, D[b]$, covering both variants of the guard. These can be composed with the other proof obligations of the if rule using SEQ, to establish judgements for each outcome of the if. These are then composed via the rule for choice, verifying the rewritten if.

Additionally, we show judgements are preserved across a variety of program transforms, the most notable being refine. This holds as the refine operation effectively unfolds choice and iteration operators into an execution trace of instructions. Therefore, we can establish $\Gamma, P, D\ \{c\}\ \Gamma', P', D' \implies \Gamma, P, D\ \{\mathsf{refine}\ c\ det\}\ \Gamma', P', D'$ for any det.

We now consider the implications of instruction reordering on logic judgements. First, we introduce $D[\beta]_f$, a *forced* update to D, such that the reordering relation is strengthened to ensure all operations are considered ordered after β and may therefore observe its effects. We also introduce the definition guards $c\ det$ to compute the weakest precondition capable of ensuring all speculated guards in c due to the decisions in det are eventually satisfiable. Notably, we are able to capture the effects of a single speculated instruction as guards $(\alpha)\ [R]$, where $[R]$ is det with a single element R and encodes the speculation of the single instruction program α.

Lemma 2 (Instruction Judgement Reordering) *Given two instructions, α and β, such that $\alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle}$, and a logic judgement over $\alpha;\beta$, a logic judgement must exist for $\beta_{\langle\alpha\rangle}$ over the original precondition strengthened to capture speculation, and a logic judgement must exist for α over a new intermediate state where the effects of $\beta_{\langle\alpha\rangle}$ are visible. Moreover, α's postcondition must match that of the original judgement.*

$$\begin{aligned}
&\alpha \overset{R}{\Leftarrow} \beta_{\langle\alpha\rangle} \wedge \Gamma, P, D\ \{\alpha;\beta\}_M\ \Gamma', P', D' \implies \\
&\quad \exists \Gamma_i, P_i, D_i \cdot \\
&\qquad \Gamma, P \wedge \mathsf{guards}\ (\alpha)\ [R], D\ \{\beta_{\langle\alpha\rangle}\}_M\ \Gamma_i, P_i, D_i \wedge \\
&\qquad \Gamma_i, P_i, D[\beta_{\langle\alpha\rangle}]_f\ \{\alpha\}_M\ \Gamma', P', D'
\end{aligned}$$

Proof First consider the new early judgement over $\beta_{\langle\alpha\rangle}$. This is relatively trivial to establish, as the proof obligations associated with β are shown to hold regardless of α's effects on the state. We demonstrate this via the constraints imposed by $D[\alpha]_W[\beta]$ on β's original proof obligations, which must be agnostic to any variables written by α due to their removal from D and subsequent quantification.

Note that the logic does not ignore all effects α may have on β, specifically in the event that α is a guard. In this case, the logic allows any proof obligations associated with β to assume the guard condition for α holds, even if they may execute out-of-order. This is based on the assumption that only valid speculation is considered, therefore, even if β executes before α, α's guard condition must eventually be true, constraining the state β executes under. This is captured by the conjunction with the guards condition, which computes the weakest precondition to ensure successful speculation.

Additionally, it is necessary to consider the implications of forwarding, as β and $\beta_{\langle\alpha\rangle}$ may not be equivalent. This is trivial, as the forwarding operation does not introduce new

behaviour, rather the $\beta_{\langle\alpha\rangle}$ represents one of the possible executions of β under the original ordering.

As a result, it is possible to establish an early judgement over $\beta_{\langle\alpha\rangle}$. We then consider the late judgement over α, from a new intermediate state between the two reordered instructions. Note that we consider $D[\beta_{\langle\alpha\rangle}]_f$ rather than D_i for the precondition in this judgement, as $\beta_{\langle\alpha\rangle}$ is considered to have been executed once it has reordered. Therefore, its effects should be visible to α.

Demonstrating this judgement relies on the use of $secure\_update$ for β, as the proof obligation ensures the early execution of β, or its forwarded variant $\beta_{\langle\alpha\rangle}$, cannot adversely increase or decrease variable classifications. Consequently, any reasoning used to ensure valid information flow for α before β must still hold after. Additionally, it is necessary to ensure $secure\_update$ holds for α, in the event it is a control variable assignment. This relies on the notion that the $falling$ and $rising$ sets must decrease given the early execution of β, as this constrains reorderings and maintains α's prior classification reasoning.

Finally, it is necessary to show that the same postcondition can be established, regardless of the execution order. This holds for P and Γ, as the reordering relation preserves thread-local reasoning and interference from other threads remains consistent regardless of reordering. For D, we show that $D[\beta_{\langle\alpha\rangle}]_f[\alpha]$ is stronger than D' and use the CONSEQ rule to establish the desired postcondition. □

We then extend this notion to consider the reordering of β prior to a trace of earlier instructions, via induction over the trace and repeated application of Lemma 2.

Lemma 3 (Program Judgement Reordering) *Given an instruction β and a* flat *program c, such that $\beta' < c < \beta$, and a logic judgement over c; β, a logic judgement must exist for β' over the original precondition strengthened to capture speculation, and a logic judgement must exist for c over a new intermediate state where the effects of β' are visible. Moreover, c's postcondition must match that of the original judgement.*

$$\begin{aligned}
&\beta' < c < \beta \wedge \mathsf{flat}\ c \wedge \Gamma, P, D\ \{c; \beta\}_M\ \Gamma', P', D' \implies \\
&\quad \exists \Gamma_i, P_i, D_i \cdot \\
&\qquad \Gamma, P \wedge \mathsf{guards}\ c\ det, D\ \{\beta'\}_M\ \Gamma_i, P_i, D_i \\
&\qquad \Gamma_i, P_i, D[\beta']_f\ \{c\}_M\ \Gamma', P', D'
\end{aligned}$$

5.5 Local bisimulation

Having established the logic's compatibility with reordering, we now turn to the proof that the logic guarantees security for each thread. Recall that this is captured by the existence of a strong bisimulation for each: the soundness proof for the logic constructs such a bisimulation, which we now describe.

We define the thread-local relation $\mathcal{B}_M$, parameterised by the thread's variables modes, and prove that it satisfies the necessary properties for compositional security. The relation is defined to require equivalent programs between its states, as well as a successful logic judgement over this program. The precondition P and the classification context Γ of the judgement must only refer to stable variables, to prevent interference from concurrent threads. Additionally, the two related memories must satisfy the precondition P and conform to the classification context Γ, mapping Low variables to the same value. Finally, the relation must also enforce low-equivalence.

Definition 4 (*Thread-Local Relation* $\mathcal{B}_M$)

$$
\begin{aligned}
&\langle c_1, mem_1\rangle\ \mathcal{B}_M\ \langle c_2, mem_2\rangle \equiv \\
&\quad \exists \Gamma, P, D, \Gamma', P', D' \cdot \\
&\qquad c_1 = c_2 \wedge \Gamma, P, D\ \{c_1\}_M\ \Gamma', P', D' \wedge \\
&\qquad \mathsf{vars}\ P \cup \mathrm{dom}\ \Gamma \subseteq \mathsf{stable}\ M \wedge \\
&\qquad mem_1 \in P \wedge mem_2 \in P \wedge \\
&\qquad \forall x \in \mathrm{dom}\ \Gamma \cdot \Gamma\ x = Low \implies mem_1\ x = mem_2\ x \wedge \\
&\qquad mem_1 \approx_M mem_2
\end{aligned}
$$

To establish a strong bisimulation, we must show that $\mathcal{B}_M$ is symmetric. This can be achieved by demonstrating symmetry for each of its sub-properties, which is only non-trivial for low-equivalence. As mentioned earlier, the Low classification constraint on control variables guarantees consistent value-dependent classifications between the two memories, resulting in a symmetric low-equivalence relation. Consequently, $\mathcal{B}_M$ is symmetric.

Next, we demonstrate $\mathsf{closed}\ \mathcal{B}_M$, ensuring the relation is closed under global operations. This is achieved by re-establishing the relation on a new pair of low-equivalent related memories that agree on the values of all stable variables. Evidently, this is trivial for all properties that do not specify the related memories. Moreover, the new memories are already known to be low-equivalent. Therefore, it is only necessary to establish that they satisfy the precondition P and conform to the classification context Γ. As these properties only constrain the unmodified stable variables, they can be re-established on the new related memories, re-establishing the relation.

As $\langle c_1, mem_1\rangle\ \mathcal{B}_M\ \langle c_2, mem_2\rangle \implies mem_1 \approx_M mem_2$ holds by definition, it only remains to show that $\mathcal{B}_M$ is a bisimulation. We phrase this property in terms of the deterministic weak memory semantics and only consider execution traces for which speculation is known to succeed. To capture this, we define $\mathsf{spec}\ c\ det\ mem$, which holds true when any speculation required to execute the next instruction in c, as determined by det, can eventually be satisfied from a memory mem, and $\mathsf{eval}\ \alpha$ to represent the deterministic evaluation of an instruction α.

Lemma 4 (Thread-Local Bisimulation)

$$
\begin{aligned}
&\forall det, c_1, mem_1, c_2, mem_2 \cdot \\
&\quad \langle c_1, mem_1\rangle\ \mathcal{B}_M\ \langle c_2, mem_2\rangle \wedge \\
&\quad (c_1, det) \rightarrow_\alpha (c_1', det') \wedge \mathsf{spec}\ c_1\ det\ mem_1 \wedge (mem_1, mem_1') \in \mathsf{eval}\ \alpha \implies \\
&\qquad \exists c_2', mem_2' \cdot \\
&\qquad\quad \langle c_1', mem_1'\rangle\ \mathcal{B}_M\ \langle c_2', mem_2'\rangle \wedge \\
&\qquad\quad (c_2, det) \rightarrow_\alpha (c_2', det') \wedge \mathsf{spec}\ c_2\ det\ mem_2 \wedge (mem_2, mem_2') \in \mathsf{eval}\ \alpha
\end{aligned}
$$

Proof We demonstrate this property via structural induction over the deterministic weak memory semantics, with most cases resolving trivially. The base case considers the execution of an instruction without reordering, $(\alpha; c, L\#det) \rightarrow_\alpha (c, det)$ (where $x\#xs$ denotes the list whose head is x and whose tail is xs). We first establish the transition and speculation properties over c_2 and mem_2, which are trivial due to $c_2 = \alpha; c$ and the lack of speculation. Then we split the judgement over $\alpha; c$ into $\Gamma, P, D\ \{\alpha\}_M\ \Gamma_i, P_i, D_i$ and $\Gamma_i, P_i, D_i\ \{c\}_M\ \Gamma', P', D'$ for some new context.

Given the logic judgement over α, we show a successful evaluation of α on mem_1 implies its evaluation must be defined for the low-equivalent memory mem_2, due to various constraints enforced by the logic. Moreover, these constraints allow us to demonstrate the preservation of low-equivalence across the evaluation of α. Finally, we prove the strongest postcondition corresponds to the instruction's effects on the state and demonstrate the resulting state only references stable variables. Combined with the judgment over c this is sufficient to solve the base case.

Next, we consider the case of instruction reordering, given as a transition of the form $(\alpha; c, R\#det) \rightarrow_{\beta_{\langle\alpha\rangle}} (\alpha; c', det')$. Using Lemma 1, we can obtain subprograms c_a and c_b, such that $\mathsf{refine}\ (\alpha; c)\ (R\#det) = \alpha; c_a; \beta'; c_b$ and $\alpha; c' = \alpha; c_a; c_b$. Given $\Gamma, P, D\ \{\alpha; c\}_M\ \Gamma', P', D'$, we use the preservation of logic judgements across refine to show $\Gamma, P, D\ \{\alpha; c_a; \beta'; c_b\}_M\ \Gamma', P', D'$. It is then possible to split and reorder this judgement using Lemma 3, establishing $\Gamma, P \wedge \mathsf{guards}\ (\alpha; c_a)\ (R\#det), D\ \{\beta_{\langle\alpha\rangle}\}_M\ \Gamma_i, P_i, D_i$ in addition to $\Gamma_i, P_i, D[\beta_{\langle\alpha\rangle}]_f\ \{\alpha; c'\}_M\ \Gamma', P', D'$ for some new intermediate context. Evidently, these judgements parallel that of the base case, where properties of the executing instruction are known and a successful application of the logic is known for the remaining program.

Mirroring the structure of the base case, we establish the transition and speculation properties over c_2 and mem_2. The transition is again trivial, as $c_2 = \alpha; c$, however, as speculation may take place, it is necessary to demonstrate a successful speculation trace exist for mem_2, given one exists for mem_1. Such a property must hold as guards are restricted to be based on Low information, as seen in rules for if and while. Consequently, any successful guard evaluation on mem_1 must also hold on the low-equivalent mem_2. This property is demonstrated via an induction over the flat program $\alpha; c_a$, proving the equivalence between these guard evaluations based on successful application of the logic's rules.

Given there exists a trace with successful speculation for both mem_1 and mem_2, we can then establish that $\mathsf{guards}\ (\alpha; c_a)\ (R\#det)$ holds for both, as guards encodes a weakest precondition calculation equivalent to spec. As a result, mem_1 and mem_2 satisfy the precondition for the reordered logic judgement over $\beta_{\langle\alpha\rangle}$, allowing it to be used to define and relate mem_1' and mem_2' via $\mathcal{B}_M$ following the same reasoning illustrated in the base case, consequently solving the reordering case.

The remaining cases solve trivially. As an illustration, consider the execution of the left program, c_1, in a non-deterministic choice, $c_1 \sqcap c_2$. Such a case requires decomposition of the four properties on the left hand side of the implication, defined over $c_1 \sqcap c_2$ and $L\#det$, such that they can be shown to hold over c_1 and det. For instance, we demonstrate $\langle c_1, mem_1\rangle\ \mathcal{B}_M\ \langle c_1, mem_2\rangle$ given $\langle c_1 \sqcap c_2, mem_1\rangle\ \mathcal{B}_M\ \langle c_1 \sqcap c_2, mem_2\rangle$ due to $\Gamma, P, D\ \{c_1 \sqcap c_2\}_M\ \Gamma', P', D' \implies \Gamma, P, D\ \{c_1\}_M\ \Gamma', P', D'$. Using these properties over c_1, it is then possible to define and relate mem_1' and mem_2' via the inductive hypothesis. □

With all necessary properties demonstrated for the local bisimulation, the compositionality theorem (Sect. 5.2) can be used to establish the global bisimulation and, consequently, the preservation of low-equivalence as defined by the security policy $\mathcal{L}$ throughout execution, i.e. that the concurrent program is secure.

6 Completeness

The logic compromises the completeness of its reasoning to simplify its application in the presence of weak memory models. This is evident in the use of W and R, as they are only

capable of expressing the absence of a possible reordering rather than enabling a detailed analysis of the potential executions. For example, the logic is not able to verify programs with control variable writes that do not alter classifications but can reorder with controlled operations. Consider the snippet $c := 3$; fence; $c := 2$; $x := High$ with a security policy $\mathcal{L}(x) = (c \leq 1)$. It is evident that $x := High$ will execute in a state where $c > 1$, validating the information flow. However, the logic is not able to verify this case, as the possible reordering of $c := 2$ and $x := High$ results in existential quantification of any references to c in the precondition associated with $x := High$.

A similar issue can be observed with the $secure\_update$ proof obligations for control variable writes, in which the classifications of reorderable reads and writes must not rise and fall respectively. False positives may arise as these checks occur even if such changes in classification would not alter the original information flow outcomes. For instance, consider the snippet $c := 2$; fence; $High := x$; $c := 0$ with a security policy $\mathcal{L}(x) = (c > 1)$. The classification of x does not influence the outcome of $High := x$, as it is always secure. However, the $secure\_update$ proof obligation for $c := 0$ will fail, as the classification of x rises in the presence of a reorderable read of x.

These cases in general describe situations where a control variable write and an operation exhibiting related information flow can reorder, however, their reordering does not change the information flow outcomes. Consequently, they result in false positives, as the logic does not track sufficient information to handle such benign cases. We believe such cases are sufficiently rare that they do not motivate the additional complexity required. Moreover, it is possible to transform these program by reordering operations and introducing fences to enable verification. For instance, the prior two snippets can be verified by reordering the last two instructions.

The logic may also produce false positives in the event of a classification test after a variable read. For example, consider the following snippet, where $\mathcal{L}(x) = (c > 1)$ and c is stable.

$$r := x; \text{ if } (c > 1) \text{ then } Low := r$$

The example is secure as $Low := r$ will only execute if $r := x$ wrote Low data to r. However, the logic is not able to establish the classification of x when considering $r := x$, due to insufficient information concerning c. Consequently, $\Gamma(r)$ is updated to $High$ and the if statement cannot be verified.

Other value-dependent logics [28] handle such cases by preserving value-dependent classifications in Γ. For example, in this case $\Gamma(r)$ would be updated to $c > 1$ for the operation $r := x$. However, under weak memory models, enabling state dependencies within Γ significantly increases complexity, as it is not clear how these value-dependent classifications and the evolving state P relate, given operations may have reordered. Consequently, the logic only supports programs that test control variables before accessing controlled variables. We believe that this captures a significant set of programs, with program transformations potentially supporting more.

Notably, our logic only supports static variables modes, due to the difficulty of coordinating these mode changes between threads, even under traditional memory models. Prior work [23] suggests the use of inline annotations to specify changes to variable modes, with additional analysis stages to verify their compatibility. However, in complex situations, extracting the implied synchronisation behaviours that demonstrate such compatibility may not be straightforward. Other work [28] has coupled the variable modes with synchronisation operations, such as locks. However, we are interested in the verification of non-blocking algorithms where locks are avoided, such as the algorithm described in Fig. 3. We anticipate an approach based

on more general rely/guarantee reasoning to express such synchronisation behaviours [10]. Consequently, to limit the complexity of the logic, we only support static variable modes and rely on additional rules such as NONBLOCKING, with underlying rely/guarantee reasoning, to handle intricate cases.

Other constraints, such as not supporting $High$ guards and the Low classification constraint for control variables, can be seen in similar work without the complexity of weak memory models [28]. We focus our efforts on supporting a similar set of features under the new semantics.

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 [12] utilising Scala, to take advantage of Scala's powerful pattern matching and compatibility with Java through the JVM, and the SMT solver Z3 [26], 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. The prototype tool is available at https://github.com/l-kent/wemelt.

Using the CONSEQ rule with the IF rule requires user intervention and is hence not amenable to automation. To enable our logic to be implemented in our symbolic execution tool, we developed the following specialisation of the IF rule, which is capable of automatically deriving a post-state. This rule is essentially a combination of the IF rule with an application of the CONSEQ rule.

$$\textsc{IfMerge}\ \frac{\Gamma, P_b \vdash b : Low \qquad \begin{array}{c}\Gamma, P \wedge [b]_M, D[b]\ \{c_1\}_M\ \Gamma_1, P_1, D_1 \\ \Gamma, P \wedge [\neg b]_M, D[b]\ \{c_2\}_M\ \Gamma_2, P_2, D_2\end{array}}{\Gamma, P, D\ \{\mathsf{if}\ (b)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\}_M\ \Gamma', P_1 \vee P_2, D'}$$

where $\mathrm{dom}\ \Gamma' = \mathrm{dom}\ \Gamma_1 = \mathrm{dom}\ \Gamma_2$ and $\forall x : \mathrm{dom}\ \Gamma' \cdot \Gamma'(x) = \Gamma_1(x) \sqcup \Gamma_2(x)$ and $D' = (\lambda\alpha \cdot D_{1W}(\alpha) \cap D_{2W}(\alpha), \lambda\alpha \cdot D_{1R}(\alpha) \cap D_{2R}(\alpha))$.

The rule ensures that the judgement on the if statement is correct no matter which branch is taken as the final state is weaker than both branch outcomes. Specifically, Γ' maps each variable to its highest value following one of the branches; P' is the disjunction of the predicates resulting from each branch; and D' maps each action to the intersection of the respective dependency analysis (W or R) for each branch.

For our symbolic execution tool, we also include an additional rule which combines the WHILE rule with an application of the CONSEQ rule.

$$\textsc{WhileHoare}\ \frac{\Gamma', P'_b \vdash b : Low \qquad \begin{array}{c}\Gamma, P, D \geq_M \Gamma', P', D' \\ \Gamma', P' \wedge [b]_M, D'[b]\ \{c\}_M\ \Gamma', P', D'\end{array}}{\Gamma, P, D\ \{\mathsf{while}\ (b)\ \mathsf{do}\ c\}_M\ \Gamma', P' \wedge [\neg b]_M, D'[b]}$$

This rule is based on the standard Hoare-logic rule for loops [15]

$$\textsc{While(Hoare- logic)}\ \frac{pre \implies inv \qquad inv \wedge b\ \{c\}\ inv}{pre\ \{\mathsf{while}\ (b)\ \mathsf{do}\ c\}\ inv \wedge \neg b}$$

with the precondition, pre, represented by Γ, P, D and the loop invariant, inv, represented by Γ', P', D'. The rule ensures that the judgement on the first iteration of the loop is correct by requiring that the pre state is stronger than the inv state. This also ensures the judgement

is correct for the case where there is no iteration of the loop. The symbolic execution is then only required to consider one iteration of the loop corresponding to the proof obligation to show the loop invariant is preserved at the end of the loop body c.

The rule requires the user to provide, in advance of running the tool, suitable values for Γ' and P' in the same way the user must provide the loop invariant in Hoare logic. The tool is able to compute D' based on a data flow analysis. If the user provides values for Γ' and P' which are too weak, the tool may produce false positives but is still sound. The rule is not applicable with values for Γ' and P' which are too strong.

In general, the logic has been structured to enable automation via symbolic execution. This is evident in its restrictions on the logic state. For example, P only tracks stable variables and, therefore, does not dramatically increase in complexity as symbolic execution proceeds over the program. Moreover, Γ maps variables to classifications, rather than predicates as seen in prior work [28,29], thereby reducing its complexity significantly.

The only part of the logic that cannot be readily automated is D. Therefore our tool tracks a safe abstraction of it instead. It is infeasible to track a set of variables for all possible instructions as is required for W and R. Instead, D is implemented as a mapping from memory operations (i.e., reads or writes to global variables) to variable sets. For example, W is implemented as W_R (which maps a variable x to all those variables whose writes must occur before a read of x) and W_W (which maps a variable x to all those variables whose writes must occur before a write of x). $W(\alpha)$ can be derived from these functions, e.g., given x and y are global variables, $W(x := 0) = W_W(x)$ and $W(x := y) = W_W(x) \cup W_R(y)$. The details of this implementation are derived from our prior work [37], and the equivalence between the two implementations has been verified in the Isabelle/HOL encoding.

8 Information flow on POWER

POWER processors allow the same reorderings as ARM, as well as the additional optimisations discussed in Sect. 4.16. This has been validated by Colvin and Smith [8] using approximately 8,000 litmus tests developed for POWER by Alglave et al. [1]. Hence, the logic developed so far can also be used for POWER.

In addition to the fences supported by ARMv8, POWER has the following fences.

- An eieio fence prevents one memory or I/O operation from starting until the previous memory or I/O operation completed. Based on the discussion in [1] we treat this as a barrier on stores only.
- A lightweight fence maintains order between loads, loads then stores, and stores, but not stores and subsequent loads (i.e., $load;load$, $load;store$, $store;store$, but not $store;load$).

Following Colvin and Smith [8], we model a lightweight fence in terms of two invented fences: a *loadgate* and a *storegate*.

$$lwfence; c \mathrel{\widehat{=}} storegate; loadgate; c$$

The $storegate$ allows stores to "move backwards" (away form the start of the program) and the $loadgate$ allows loads to "move forwards" (towards the start of the program). For instance, assume the following sequence of instructions, where l_i are loads and s_i are stores.

$$l_1; s_1; storegate; loadgate; l_2; s_2$$

Thread 1:	**Thread 2:**	**Thread 3:**

z := 1;	while(z≠1){}
	x := secret
		if (z=0) then
		  cfence;
		  r := x;
		  fence;
		if (z=0) then
		  y := r

Fig. 6 Non-multi-copy atomicity example

Assuming all loads and stores are to different variables and hence there are no pairwise constraints on reordering, the following reordering is possible.

$$l_1; storegate; l_2; s_1; loadgate; s_2$$

Note that the order between loads, between stores, and between loads then stores has been maintained, but load l_2 may be reordered before the store s_1.

For these fences, the $\overset{R}{\Leftarrow}$ relation is as follows [8].

$$\begin{aligned}
eieio &\overset{R}{\not\Leftarrow} \alpha && \text{if } \alpha \text{ is a store}\\
\alpha &\overset{R}{\not\Leftarrow} eieio && \text{if } \alpha \text{ is a store}\\
\alpha &\overset{R}{\Leftarrow} storegate && \text{if } \alpha \text{ is a store}\\
\alpha &\overset{R}{\not\Leftarrow} storegate && \text{otherwise}\\
loadgate &\overset{R}{\Leftarrow} \alpha && \text{if } \alpha \text{ is a load}\\
loadgate &\overset{R}{\not\Leftarrow} \alpha && \text{otherwise}
\end{aligned}$$

Based on these definitions, it is then possible to determine suitable appropriate updates to D for these fences, as seen in Sect. 4.11. Moreover, the rule for these fences is identical to those of ARMv8.

$$\textsc{FenceP}\ \frac{\alpha \in \{\mathsf{eieio}, \mathsf{loadgate}, \mathsf{storegate}\}}{\Gamma, P, D\ \ \{\alpha\}_M\ \ \Gamma, P, D[\alpha]}$$

9 Non-multi-copy atomicity

ARMv8 [31] is multi-copy atomic, meaning updates made by a thread are seen by all other threads at the same time. This is not the case for POWER [33] and older versions of ARM [14]. Under these architectures, writes may be propagated to some threads earlier than others, via mechanisms such as shared buffers and inter-thread communication. Consequently, these architectures expose new behaviours when two or more threads attempt to synchronise their executions based on writes from another.

For example, consider the code in Fig. 6 in which there are 3 threads: the first sets z to 1, the second waits for z to become 1 then assigns classified information to x, and the third uses a non-blocking read operation to read a non-classified value of x, i.e., a value before z is set to 1. The fences in the third thread ensure that the value of x is read into r after the first branch condition and before the second branch condition is checked; and hence while $z = 0$. Despite this careful placement of fences, under non-multi-copy atomicity the following scenario is possible:

1. The first thread sets z to 1. This new value becomes available to the second thread, but not yet to the third.
2. The second thread updates x to the classified value $secret$. This new value becomes available to the third thread.
3. The third thread, based on the original value of z, updates the value of y to the new (classified) value of x.

This vulnerability illustrates the case where two threads attempt to synchronise their executions based on another's write to z, with the second thread's accesses to x intended to occur after $z := 1$ and the third thread's prior. Therefore, non-multi-copy atomicity can introduce a security leak by breaking this synchronisation and allowing the third thread to observe the effects of the second.

Fortunately, our logic prevents such leaks as it is not possible to establish non-trivial information flow between two or more threads based on the write of another. This can be attributed to the use of static variable modes, which prevent the analysis state, P and Γ, from retaining information regarding a variable that may be written by another thread at any stage of the execution. Consequently, it is not possible to discharge an instruction's proof obligations if they are dependent on this synchronisation behaviour.

To illustrate, consider the verification of the second thread of Fig. 6. It would not be possible for this thread to contain z in its stable set, due to the first thread's write to z. As a result, the condition for the loop exit, $z = 1$, would not be retained in P. Therefore, the write to x would only be considered secure if the written expression was classified as Low, preventing any insecure behaviour when considering interactions with the third thread. As this is not the case, no logic judgement can be established for the second thread.

Notably, the NONBLOCKING rule allows for variables to be added to the stable set for a thread, potentially allowing for the effects of writes from concurrent threads to be observed and used for thread-local reasoning. For instance, such a rule would allow for the verification of a variation of the third thread in Fig. 6. However, the rule disallows $Global$ writes during the period of gained stability to ensure unsuccessful executions aren't observed. Therefore, it would not be possible to influence the execution of another thread based on this gained information.

As a result, the execution behaviour introduced by non-multi-copy atomicity invalidates only synchronisation reasoning our logic currently does not support. Hence, the logic's soundness argument is preserved on non-multi-copy atomic architectures.

10 Case study: Cross-domain work-stealing deque

To illustrate our logic on a larger example, we apply it to a version of the Chase-Lev work-stealing deque [7]. Work-stealing deques (double-ended queues) are often used for load balancing in multicore systems. Each worker process has a deque, which it uses to record tasks to be performed. Thus, a worker executes put and $take$ operations that, respectively, add tasks to and remove tasks from its deque. Load balancing is achieved by allowing other, so-called "thief" processes, whose own deques are empty to execute $steal$ operations that remove elements from the deque. To avoid contention between the worker and thief processes, put and $take$ operate at the opposite end of the deque from $steal$ operations—a worker adds and removes tasks at the tail, whereas thieves steal tasks from the head. Contention between the worker and thieves, therefore, only occurs when the deque has one element.

```
put(v, u)
    int t;
    t := tail;
 ▷ levels[t mod L] := u;
    tasks[t mod L] := v;
    fence;
    tail := t+1;
    return;
```

```
take
    int h, t, task;
    t := tail-1;
    tail := t;
    fence;
    h := head;
    if (h <= t)
        task := tasks[t mod L];
        if (h=t)
            if !CAS(head, h, h + 1) then
                task := empty;
            tail := tail+1;
    else
        task := empty;
        tail := tail+1;
    return task;
```

```
steal
    int h, t, task, level, r;
    h := head;
    fence;
    t := tail;
    if (h < t)
        cfence;
     ▷ level := levels[h mod L];
     ▷ if (level=Low)
            task := tasks[h mod L];
     ▷ else
     ▷      task := fail;
       if (!CAS(head, h, h+1))
            task := fail;
    else
        task := empty;
    return task;
```

Fig. 7 Insecure version of cross-domain work-stealing deque. The code extends that of [9]. Additional lines are marked with ▷

The Chase-Lev deque is implemented as a circular array of size L with a *head* and *tail* pointer. The pointers are non-wrapping, i.e., if a pointer has the value i, it points to the array element at position $i \mod L$.

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

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

A version of the Chase-Lev deque developed specifically for ARM was presented in [19]. It includes, for example, a full fence in the *put* operation so that the increment of the tail pointer does not take effect before the element is placed in the array, and in the *take* operation to ensure publication of its intent to take the task. Errors in the placement of control fences in the *steal* operation of this version of the deque were corrected by Colvin and Smith in [9]. We extend their version of the deque to operate in a *cross-domain* environment where tasks are given a security level, and processes are only allowed to access tasks for which they have the appropriate permissions. Specifically, we examine the scenario where we have a single worker thread which is allowed to access high and low tasks, and several thief threads which are allowed only to access low tasks.

A first attempt at the cross-domain deque is shown in Fig. 7. As well as a circular array of tasks, the deque has a circular array of security levels. This array is also of size L and records

```
put(v, u)
    int t;
    t := tail;
 ▷ z := z+1;
 ▷ fence.st;
    levels[t mod L] := u;
    tasks[t mod L] := v;
    fence;
 ▷ z := z+1;
    tail := t+1;
    return;
```

```
take
    int h, t, task;
    t := tail-1;
    tail := t;
    fence;
    h := head;
    if (h <= t)
        task := tasks[t mod L];
        if (h=t)
            if !CAS(head, h, h + 1) then
                task := empty;
            tail := tail+1;
    else
        task := empty;
        tail := tail+1;
    return task;
```

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

Fig. 8 Secure version of cross-domain work-stealing deque. The code extends that of Fig. 7. Additional lines are marked with ▷

in position i the security level of the task in position i of the task array. The *put* operation has two inputs, a task v and security level u, and updates both arrays. The *steal* operation reads the security level of the task it is trying to acquire and returns $fail$ when that task is high.

We applied our ARMv8 logic to this code using our symbolic execution tool. The tool reported an error due to the ASSIGN rule failing for the assignment $task := task[h \bmod L]$ in $steal$. This correctly identified an information leak which arises due to $tasks$ being a finite circular array. Successive *put* operations can cause $tail$ to wrap-around to the start of the array and then catch up to $head$. In this situation, it is possible that $steal$ reads Low from the $levels$ array, and then, before it reads from the $tasks$ array, the *put* operation occurs putting a high task in $tasks$.

To avoid this problem, we could prevent *put* from overwriting values that have not been read yet. However, in many applications such overwriting is desirable (to lose old tasks, which may no longer be relevant, rather than new tasks). Instead, we ensure that the $steal$ operation cannot read tasks which have been concurrently overwritten. To accomplish this, we use an approach inspired by seqlock again (as we did in the secure version of the IO-driver in Fig. 3). The resulting code is shown in Fig. 8. If z changes at any time while $steal$ is reading a level and associated task, the read is restarted. We also applied our ARMv8 logic to this code using our symbolic execution tool and, in this case, no information leaks were identified.

Each of the operations in Figs. 7 and 8 were checked by the tool in isolation (as if they were each being called by a different thread). The code was annotated with the following specifications:

– For each variable, a value-dependent security classification, i.e., a predicate, was supplied. For most variables (all local variables and all global variables apart from $tasks$ and the input v to put), this predicate was simply $true$ (Low) or $false$ ($High$).

– For each global variable, a mode was supplied.
– Each operation also required an initial P and Γ constraining what states the operation could be called from.

These specifications were largely the same for all operations (differing only on local variables and inputs and outputs). For the shortest operation *put* of Fig. 7 with an array of length 2, there were 17 lines of specification for 7 lines of code. However, 6 of these lines of specification were simply stating that the security classification of a variable was $True$ (something that could be automatically generated as a default), and another 6 were modes (which could also be automatically generated by syntactically checking which variables are read and written by each thread). This would reduce the specification to 5 lines for 7 lines of code. For the longest operation *steal* of Fig. 8, there were 22 lines of specification for 18 lines of code. Defaulting to $True$ security classifications, and automatically generating modes would reduce this to 5 lines of specification for 18 lines of code. Further reductions in the number of lines of specification per line of code could be made by sharing the annotations for global variables between threads. No such optimisations were employed in our prototype tool.

In addition to the specifications, the *steal* operation of Fig. 8 required invariants for each of its loops (and an annotation of z for the outer loop to enable the use of the NonBlocking rule). The invariants on the program state, P, simply ensured the array size L remained constant, and that on the security levels, Γ, that each of the local variables was low. Our experience with similar non-blocking algorithms, indicates that such simple invariants are common (in many cases, the invariant $True$ suffices for program state). Finding ways to generate base invariants which the user could build on is an interesting area for future work.

No optimisations were made for the performance of the prototype tool. Each of the operations of Figs. 7 and 8 could be checked instantaneously (within milliseconds) for an array of length 2. However, the execution time increased exponentially as the size of the array increased. This was due to the running time of the Z3 solver which had to deal with the tool's representation of predicates increasing exponentially. While this lack of optimisation is acceptable for this prototype tool, implemented as a proof-of-concept, further optimisations would be required to allow the logic to be applied to larger examples.

11 Conclusion

In this paper, we have presented a comprehensive information-flow logic for ARM and POWER multicore processors. Our logic supports dynamic, value-dependent security classifications, and is compositional, flow-sensitive, and enforces a constant-time guarantee. It has been proven sound with respect to existing, validated operational semantics of ARM and POWER, and implemented in a symbolic execution tool. The latter was enabled by designing the logic for automation; it is both thread-local (allowing reasoning about one thread at a time), and step-local (allowing reasoning about one line of code at at time).

Our immediate future work will focus on two tasks. First, we will extend the logic with general rely/guarantee conditions allowing assumptions to be used in thread-local reasoning that include arbitrary constraints between program variables (see [10] for progress in this direction). This will widen the logic's applicability by allowing assumptions which hold only under certain conditions, and hence can vary as the program executes. Second, we will adapt our logic to a suitable intermediate representation into which we can lift actual ARM and POWER assembly code. Such an intermediate representation will need to be accurate enough to maintain all ordering and dependencies on assembly instructions.

Longer term we will focus on improving the efficiency and scalability of tool support for our logic, and its adaptation to other processors such as the open-source RISC-V architecture.

Finally, we note that we expect our logic could also be extended to reason about programs that intentionally reveal secret information, i.e. *secure declassification*. Specifically, recent work [34] has shown how declassification policies can be reasoned about from functional correctness annotations on concurrent programs. These annotations take the form of predicates P that annotate each program statement. Our program logic already provides a mechanism for computing such annotations in the form of the predicates P in its state Γ, P, D, at each point of the program text. We leave investigation of this intriguing possibility for future work.

Acknowledgements This work was supported by Australian Research Council Discovery Grant DP160102457, and a combination of Next Generation Technologies Fund (NGTF) and Strategic Research Initiative (SRI) funding from the Defence Science and Technology Group, Australia. Thanks to Liam Kent for implementing the symbolic execution tool.

References

1. Alglave J, Maranget L, Tautschnig M (2014) Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans Program Lang Syst 36(2):7:1-7:74. https://doi.org/10.1145/2627752
2. Almeida JB, Barbosa M, Barthe G, Dupressoir F, Emmi M (2016) Verifying constant-time implementations. In: Holz T, Savage S (eds) 25th USENIX Security Symposium, USENIX Security 16, pp 53–70. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida
3. Barthe G, Blazy S, Grégoire B, Hutin R, Laporte V, Pichardie D, Trieu A (2020) Formal verification of a constant-time preserving C compiler. Proc ACM Program Lang (PACMPL) 4(POPL):7:1-7:30. https://doi.org/10.1145/3371075
4. Boehm H (2012) Can seqlocks get along with programming language memory models? In: Zhang L, Mutlu O (eds.) Proceedings of the 2012 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '12, pp 12–20. ACM. doi: https://doi.org/10.1145/2247684.2247688
5. Casinghino C, Paasch JT, Roux C, Altidor J, Dixon M, Jamner D (2019) Using binary analysis frameworks: The case for BAP and angr. In: Badger JM, Rozier KY (eds) NASA Formal Methods—11th International Symposium, NFM 2019, Lecture Notes in Computer Science, vol. 11460, pp 123–129. Springer, Berlin. https://doi.org/10.1007/978-3-030-20652-9_8
6. Chandy KM, Misra J (1981) Asynchronous distributed simulation via a sequence of parallel computations. Commun ACM 24(4):198–206. https://doi.org/10.1145/358598.358613
7. Chase D, Lev Y (2005) Dynamic circular work-stealing deque. In: ACM symposium on parallelism in algorithms and architectures (SPAA'05), pp 21–28. ACM Press, New York. https://doi.org/10.1145/1073970.1073974
8. Colvin RJ, Smith G (2018) A high-level operational semantics for hardware weak memory models. CoRR abs/1812.00996
9. Colvin RJ, Smith G (2018) A wide-spectrum language for verification of programs on weak memory models. In: Havelund K, Peleska J, Roscoe B, de Vink EP (eds.) Formal Methods—22nd International Symposium, FM 2018, Lecture Notes in Computer Science, vol. 10951, pp 240–257. Springer. https://doi.org/10.1007/978-3-319-95582-7_14
10. Coughlin N, Smith G (2020) Rely/guarantee reasoning for noninterference in non-blocking algorithms. In: 33rd IEEE Computer Security Foundations Symposium, CSF 2020, pp 380–394. IEEE. doi: https://doi.org/10.1109/CSF49147.2020.00034
11. D'Silva V, Payer M, Song DX, (2015) The correctness-security gap in compiler optimization. In (2015) IEEE Symposium on Security and Privacy Workshops, SPW 2015, pp 73–87. IEEE Computer Society. https://doi.org/10.1109/SPW.2015.33
12. Ernst G, Murray T (2019) SecCSL: Security concurrent separation logic. In: Dillig I, Tasiran S (eds) Computer Aided Verification—31st International Conference, CAV 2019, Proceedings, Part II, Lecture Notes in Computer Science, vol. 11562, pp 208–230. Springer, Berlin. https://doi.org/10.1007/978-3-030-25543-5_13

13. Fitzpatrick J (2011) An interview with Steve Furber. Commun ACM 54(5):34–39. https://doi.org/10.1145/1941487.1941501
14. Flur S, Gray KE, Pulte C, Sarkar S, Sezgin A, Maranget L, Deacon W, Sewell P (2016) Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In: Bodík R, Majumdar R (eds) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp 608–621. ACM. https://doi.org/10.1145/2837614.2837615
15. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580. https://doi.org/10.1145/363235.363259
16. Jones CB (1983) Specification and design of (parallel) programs. In: Proceedings of IFIP'83, pp 321–332. North-Holland
17. Kang J, Hur C, Lahav O, Vafeiadis V, Dreyer D (2017) A promising semantics for relaxed-memory concurrency. In: Castagna G, Gordon AD (eds) Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages, POPL 2017, pp 175–189. ACM. http://dl.acm.org/citation.cfm?id=3009850
18. Kocher P, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2018) Spectre attacks: Exploiting speculative execution. CoRR abs/1801.01203. http://arxiv.org/abs/1801.01203
19. Lê N, Pop A, Cohen A, Zappa Nardelli F (2013) Correct and efficient work-stealing for weak memory models. In: Principles and Practice of Parallel Programming (PPoPP'13), pp 69–80. ACM. doi: https://doi.org/10.1145/2442516.2442524
20. Leroy X, Blazy S, Kästner D, Schommer B, Pister M, Ferdinand C (2016) CompCert—a formally verified optimizing compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. SEE. https://hal.inria.fr/hal-01238879
21. Lourenço L, Caires L (2015) Dependent information flow types. In: Rajamani SK, Walker D (eds.) Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2015, pp 317–328. ACM. https://doi.org/10.1145/2676726.2676994
22. Mantel H, Perner M, Sauer J (2014) Noninterference under weak memory models. In: IEEE 27th computer security foundations symposium, CSF 2014, pp 80–94. IEEE Computer Society. https://doi.org/10.1109/CSF.2014.14
23. Mantel H, Sands D, Sudbrock H (2011) Assumptions and guarantees for compositional noninterference. In: Proceedings of the 24th IEEE computer security foundations symposium, CSF 2011, pp 218–232. IEEE Computer Society. https://doi.org/10.1109/CSF.2011.22
24. Moir M, Shavit N (2004) Concurrent data structures. In: Mehta DP, Sahni S (eds) Handbook of data structures and applications. Chapman and Hall/CRC, Boca Raton. https://doi.org/10.1201/9781420035179.ch47
25. Molnar D, Piotrowski M, Schultz D, Wagner DA (2005) The program counter security model: automatic detection and removal of control-flow side channel attacks. In: Won D, Kim S (eds.) Information security and cryptology - ICISC 2005, 8th international conference, lecture notes in computer science, vol. 3935, pp 156–168. Springer, Berlin. https://doi.org/10.1007/11734727_14
26. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Ramakrishnan CR, Rehof J (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, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin. https://doi.org/10.1007/978-3-540-78800-3_24
27. Murray TC (2015) Short paper: On high-assurance information-flow-secure programming languages. In: Clarkson M, Jia L (eds.) Proceedings of the 10th ACM workshop on programming languages and analysis for security, PLAS@ECOOP 2015, pp 43–48. ACM. https://doi.org/10.1145/2786558.2786561
28. Murray TC, Sison R, Engelhardt K, (2018) C overn: A logic for compositional verification of information flow control. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, pp 16–30. IEEE. https://doi.org/10.1109/EuroSP.2018.00010
29. Murray TC, Sison R, Pierzchalski E, Rizkallah C (2016) Compositional verification and refinement of concurrent value-dependent noninterference. In: IEEE 29th computer security foundations symposium, CSF, pp 417–431. IEEE Computer Society (2016). https://doi.org/10.1109/CSF.2016.36
30. Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL—A proof assistant for higher-order logic. lecture notes in computer science, vol. 2283. Springer, Berlin. https://doi.org/10.1007/3-540-45949-9
31. Pulte C, Flur S, Deacon W, French J, Sarkar S, Sewell P (2018) Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proceedings of the ACM on programming languages (PACMPL) 2(POPL):19:1-19:29. https://doi.org/10.1145/3158107
32. Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE J Sel Areas Commun 21(1):5–19. https://doi.org/10.1109/JSAC.2002.806121

33. Sarkar S, Sewell P, Alglave J, Maranget L, Williams D (2011) Understanding POWER multiprocessors. In: Hall MW, Padua DA (eds.) Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, PLDI 2011, pp 175–186. ACM. https://doi.org/10.1145/1993498.1993520
34. Schoepe D, Murray T, Sabelfeld A (2020) VERONICA: Expressive and precise concurrent information flow security. In: IEEE Computer Security Foundations Symposium (CSF), pp 79–94
35. Sewell P, Sarkar S, Owens S, Nardelli FZ, Myreen MO (2010) x86-TSO: a rigorous and usable programmers model for x86 multiprocessors. Commun ACM 53(7):89–97. https://doi.org/10.1145/1785414.1785443
36. Sison R, Murray T (2019) Verifying that a compiler preserves concurrent value-dependent information-flow security. In: Harrison J, O'Leary J, Tolmach A (eds.) International conference on interactive theorem proving (ITP 2019), Leibniz international proceedings in informatics, vol. 141, pp 27:1–27:19. Schloss Dagstuhl-Leibniz-Zentrum für Informatik
37. Smith G, Coughlin N, Murray T (2019) Value-dependent information-flow security on weak memory models. In: ter Beek MH, McIver A, Oliveira JN (eds) Formal Methods—The Next 30 Years—Third World Congress, FM 2019, Lecture Notes in Computer Science, vol 11800, pp 539–555. Springer, Berlin. https://doi.org/10.1007/978-3-030-30942-8_32
38. Tan YK, Myreen MO, Kumar R, Fox ACJ, Owens S, Norrish M (2019) The verified CakeML compiler backend. J. Funct. Program. 29:e2. https://doi.org/10.1017/S0956796818000229
39. Vaughan JA, Millstein TD (2012) Secure information flow for concurrent programs under Total Store Order. In: Chong S (ed.) 25th IEEE Computer Security Foundations Symposium, CSF 2012, pp 19–29. IEEE Computer Society. doi: https://doi.org/10.1109/CSF.2012.20
40. Zheng L, Myers AC (2007) Dynamic security labels and static information flow control. Int. J. Inf. Sec. 6(2–3):67–84. https://doi.org/10.1007/s10207-007-0019-9