Value-Dependent Information-Flow Security on Weak Memory Models

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

[1]Defence Science and Technology Group, Australia

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

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

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 it 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 contemporary multicore processors leads to vulnerabilities in such programs, and present a compositional, timing-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.

1 Introduction

Modern multicore processors utilise *weak memory models* which, for reasons of efficiency, allow instructions to take effect in an order different to that in the program text [25]. Such instruction reordering is constrained by basic principles, the key one being that the sequential semantics of each thread in the original code should be preserved [6, 7]. This ensures the effects of weak memory models can largely be ignored by programmers whose code is either not concurrent, or is concurrent but data-race free.[1] However, these effects do need to be considered by programmers writing efficient low-level code for device drivers and data structures. Such code is generally concurrent and *non-blocking*, i.e., using no, or minimal, locking, and hence inherently not data-race free [16]. It is well known that this affects the correctness of such code on weak memory models [2]. As shown by Vaughan and Milstein [26] (for the weak memory model TSO) and Mantel et al. [14] (for TSO, PSO and IBM-370), it also leads to security violations which are not detectable using the standard approaches to information-flow security.

[1] The recently discovered Meltdown [12], Spectre [11] and Foreshadow [4] vulnerabilities show that this is not strictly the case.

While TSO [24] is widely used (by chip manufacturers Intel and AMD), PSO and IBM-370 are not supported on recent processors. More relevant weak memory models are ARM [9, 22] and IBM POWER [23]; the former being widely used in mobile devices [8]. These memory models are significantly weaker (allowing more kinds of reordering) than those studied by the papers above, yet have received little attention from the security community.

Additionally, the effects of weak memory models on programs with security levels that are *value-dependent* [13, 18, 27] have not been explored to date. In such programs, the security level of a variable may depend on the values of one or more other variables. Hence, it may change as the program changes the state.

Building on Mantel et al.'s compositional information-flow logic for concurrent programs [15], Murray et al. [19, 20] provide two information-flow logics for concurrent programs which are compositional and also handle dynamic, value-dependent security levels. The latter of these has been successfully applied to a non-trivial concurrent program running on an embedded device which facilitates secure interaction with multiple classified networks.

In this paper, we take this work further by incorporating the effects of weak memory models. Our logic specifically captures the effects of the revised version of ARMv8 [22], the latest version of ARM.[2] This memory model has much in common with prior versions of ARM [9], and with IBM POWER [23]. Our logic has been proven sound with respect to a recent operational semantics of ARMv8 [6] which has been validated against approximately 10,000 litmus tests run on actual hardware.

We begin with an overview of weak memory models in Section 2 and demonstrate how they can lead to security vulnerabilities in value-dependent security systems in Section 3. In Section 4, we present a formal framework for our logic which is presented in full in Section 5. We discuss the issue of timing sensitivity in Section 6 and conclude in Section 7.

2 Weak Memory Models

Hardware weak memory models, as exemplified by TSO [24], ARM [9, 22] and IBM POWER [23], 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

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

cores. For example, if a thread t buffers the writes to variables x and y while executing the code $\mathsf{x} := 1; \mathsf{y} := 2$ 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 $\mathsf{y} := 2;\ \mathsf{x} := 1$.

Colvin and Smith [6, 7] 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 $\mathsf{x} := e$ can be reordered with an assignment $\mathsf{y} := f$ 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 $\mathsf{x} := 1;\ \mathsf{x} := 2$ has a different final value of x (and hence different behaviour) than $\mathsf{x} := 2;\ \mathsf{x} := 1$. Constraint (ii) is required since $\mathsf{x} := 1;\ \mathsf{y} := \mathsf{x}$ will result in a different value for y than $\mathsf{y} := \mathsf{x};\ \mathsf{x} := 1$ when the initial value of x is not 1. Similarly, constraint (iii) is required since $\mathsf{x} := \mathsf{y};\ \mathsf{y} := 1$ can result in a different value for x than $\mathsf{y} := 1;\ \mathsf{x} := \mathsf{y}$. Finally, constraint (iv) is required so that the order of updates and accesses of each global variable, considered individually, is maintained: $\mathsf{x} := \mathsf{z};\ \mathsf{y} := \mathsf{z}$ will not behave the same as $\mathsf{y} := \mathsf{z};\ \mathsf{x} := \mathsf{z}$ 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 $\mathsf{x} := e;\ \mathsf{y} := \mathsf{x}$ to be reordered to $\mathsf{y} := e;\ \mathsf{x} := e$ 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. They will also have reordering constraints related to other types of instructions such as branch instructions and fences (see Section 3.1 for the branch constraints on ARM). Fences are a means by which the programmer can enforce ordering where necessary in their program. For example, letting fence denote a full fence (e.g., the instruction DMB on ARM), the program $\mathsf{x} := 1;\ \mathsf{fence};\ \mathsf{y} := 2$ 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 [6, 7].

3 Weak Memory Models and Security

We are interested in evaluating the security of assembly code running on ARMv8 processors [9]. For ease of presentation, we adopt a high-level language to represent assembly commands (as in [6,7]). The syntax of a command, i.e., a program, c in this language is as follows:

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

where x is a (global or local) variable, b a Boolean condition, and e an expression. In our examples, we also allow $\mathsf{do}\ c\ \mathsf{while}\ (b)$ as a shorthand for $c\ ;\ \mathsf{while}\ (b)\ \mathsf{do}\ c$.

This simple language has only one kind of fence (a full fence which requires all updates to be seen by all threads before proceeding). ARM additionally supports store fences (which maintain an order on stores only) and a control fence (for restricting speculative execution beyond branch points). Extending our approach to cater for such additional constructs is beyond the scope of this paper which focusses on the interplay between value-dependent security levels and instruction reordering in weak memory models.

To illustrate this interplay we introduce the example of Figure 1. 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 the data is read (how this is detected is elided in the abstract representation of Figure 1), the operation enters some unclassified data in x (the value 0) before setting z back to an even number by incrementing it again. As we will see, the setting of z ensures that the classified input is not readable by all applications running on the computer to which the keyboard is attached.

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

Fig. 1. 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.

We call z a *control variable* because it controls the security level of x; when it is even x may only contain unclassified data, but when it is odd it may also contain classified data. The use of such control variables provides us with *value-dependent security* [13, 18, 27].

Next consider the operations which read from the buffer. We have a secret_read operation which only applications which are allowed access to classified information can call, 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 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.

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* [16]. 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 Figure 1 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 [3], and is a typical example of a non-blocking algorithm.

3.1 Value-Dependent Security and Reordering

The implementation in Figure 1 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 constraints presented in Section 2.

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 Figure 2.

Similar issues arise with the read operation. To understand these, we first provide the rules for reordering involving branch instructions on ARM processors [6, 7].

1. 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.
2. 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.
3. Two branch instructions can be reordered if, and only if, their branching conditions do not reference common global variables.

In case 1 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.

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

Fig. 2. Versions of the operations (secret_write) and (read) which are secure when run on the ARMv8 memory model.

In the read operation two problems arise due to these reorderings. Firstly, since r2 is a local variable, the assignment to r2 could be reordered with the first branch instruction (case 1) 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.

Secondly, if the assignment to r2 is reordered with the second branch condition (case 2) then it is possible that a secret_write operation begins after the check of that branch condition and hence r2 is loaded with classified data. Again, a fence can prevent the reordering. A secure version of read is included in Figure 2.

4 Formal Framework

In this section, we provide a formal framework on which we build our logic in Section 5. 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 $Local \cap Global = \varnothing$. We let $var(e)$ denote the set of variables which occur free in an expression e.

4.1 Assumptions and Guarantees

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 [5, 10]. 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 z then all other threads must guarantee that they do not.

Mantel et al. [15] adopt this approach in their concurrent information-flow logic by assigning variables referenced by a thread to one or more of the following *modes*.

- *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 another thread)
- *GuarNoW* - this thread does not write to the variable (but may read it)
- *GuarNoRW* - this thread does not read or write to the variable

In our logic, such modes are represented by a function $M : \mathit{Mode} \rightarrow \mathcal{P}\,\mathit{Var}$ mapping each mode to the set of variables which have that mode. Local variables are always non-readable and non-writable by other threads, i.e., $\mathit{Local} \subseteq M(\mathit{AssNoRW})$.

4.2 Value-Dependent Security Levels

Murray et al. [19, 20] extend the approach of Mantel et al. [15] to include value-dependent security levels. As in that work, we adopt a two-point lattice of security levels with values *Low* and *High* such that $\mathit{Low} \sqsubseteq \mathit{High}$ and $\mathit{High} \not\sqsubseteq \mathit{Low}$ (meaning that information classified *High* should not flow to a variable classified *Low*).

Also following [19, 20], we let $\mathcal{L}(x)$, for a variable x, be a predicate which is true precisely when x has security level *Low*. For example, $\mathcal{L}(\mathsf{x}) = (\mathsf{z} \,\%\, 2 = 0)$ in the example of Section 3, i.e., the security level of x depends on the parity of z. $\mathcal{L}$ is provided by the user and is independent of the program's state. In order to determine the security level of a variable in our logic, we introduce the following.

- A partial function $\Gamma : \mathit{Var} \nrightarrow \{\mathit{Low}, \mathit{High}\}$ whose domain is the set of *stable* variables, i.e., variables in $M(\mathit{AssNoRW}) \cup M(\mathit{AssNoW})$, and which returns the security level of data held by those variables. This data can be at a lower level than the variable's security level, i.e., a variable with a *High* security level may hold *Low* data. The data referred to by Γ at any point in the execution of a program assumes that precisely the instructions up to that point have been executed, i.e., instruction reordering due to a weak memory model is not considered.
- A predicate P on the program's variables (capturing the current state). We let $\mathit{low}_P(x) \mathrel{\widehat{=}} P \vdash \mathcal{L}(x)$ denote that x's security level is provably *Low* when P holds, and $\mathit{high}_P(x) \mathrel{\widehat{=}} P \vdash \neg\, \mathcal{L}(x)$ denote that x's security level is provably *High* when P holds. As for Γ, at any point in the execution of a program, P assumes that precisely the instructions up to that point have been executed, i.e., instruction reordering is not considered.

Based on these we define the following shorthand for determining the security level t of an expression e (as the highest level of any free variable in e).

$$\Gamma, P \vdash e : t \quad \widehat{=} \quad t = \textstyle\bigsqcup_{x \in var(e)} \Gamma_P(x)$$

$$\text{where} \qquad \Gamma_P(x) \quad \widehat{=} \quad \begin{cases} \Gamma(x) & \text{if } x \in \operatorname{dom} \Gamma \\ \mathit{Low} & \text{if } x \notin \operatorname{dom} \Gamma \text{ and } \mathit{low}_P(x) \\ \mathit{High} & \text{otherwise} \end{cases}$$

Note that when the security level of one or more variables in an expression is unknown (i.e., neither specified in Γ nor derivable from P), Γ_P will default to security level *High* for those variables. This ensures that an expression which we are assigning to a variable is given its highest possible security level.

When determining the security level of a variable x to which we assign a value, on the other hand, we want to default to *Low*.

$$\mathit{eval}_P(x) \quad \widehat{=} \quad \begin{cases} \mathit{High} & \text{if } \mathit{high}_P(x) \\ \mathit{Low} & \text{otherwise} \end{cases}$$

Following Murray et al. [19, 20], we assume control variables are always *Low*, i.e., $\mathcal{L}(\mathsf{z}) = \mathit{true}$ for each control variable z. As a result, it is not necessary to include them in Γ when they are stable.

4.3 Weak Memory Models

Γ and P ignore the effects of reordering possible under a weak memory model. This is not a problem for Γ under the defined reordering constraints, as it is only consulted for the reads of an instruction. If an instruction containing such an expression e is reordered before a prior write to a variable x then, according to the constraints in Section 2, 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 evaluation of e's security level. 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. To use P we need to consider guarantees on the ordering of program instructions. To capture these guarantees in our logic, we introduce a function known_W where, for a given instruction a, $\mathit{known}_W(a)$ is the set of variables whose most recent prior write in the program is known to have occurred. Hence, these variables' values in P can be used when determining the security level of x (using $\mathit{eval}_P(x)$) and the expression assigned to x (using $\Gamma, P \vdash e : t$).

The value of $\mathit{known}_W(a)$ evolves as the program progresses. For example, given the code $\mathsf{z} := \mathsf{x}$; $\mathsf{y} := \mathsf{x}$; $\mathsf{z} := 0$; $\mathsf{y} := \mathsf{x}$ where z, y and x are global variables, after the first assignment $\mathit{known}_W(\mathsf{y} := \mathsf{x})$ contains z since the first assignment must occur before the second due to constraint (iv) of Section 2. However, after the third assignment $\mathit{known}_W(\mathsf{y} := \mathsf{x})$ does not contain z since the fourth assignment can be reordered before the third.

We similarly introduce a function $\mathit{known}_R(a)$ to denote the set of variables whose most recent prior read in the program is known to have occurred. This set is required in cases where a read of a variable may be reordered with an instruction which changes the variable's security level (see Section 5 for details).

We define known_W and known_R in terms of four other functions each of type $\mathit{Var} \to \mathcal{P}\,\mathit{Var}$ capturing the dependencies between writes and reads:

- $W_w(x)$ returns the set of variables whose prior writes, if any, have occurred when we reach an instruction which writes to x;
- $W_r(x)$ returns the set of variables whose prior writes, if any, have occurred when we reach an instruction which reads x;
- $R_w(x)$ returns the set of variables whose prior reads, if any, have occurred when we reach an instruction which writes to x; and
- $R_r(x)$ returns the set of variables whose prior reads, if any, have occurred when we reach an instruction which reads x.

Given these definitions, we define $\mathit{known}_\psi(a)$ where ψ stands for either W or R as

$$\mathit{known}_\psi(a) = \begin{cases} \mathit{Var} & \text{if } a = \mathsf{fence} \\ \bigcup_{y \in \mathit{wr}(a)} \psi_w(y) \cup \bigcup_{y \in \mathit{rd}(a)} \psi_r(y) & \text{otherwise} \end{cases}$$

where $\mathit{wr}(a)$ is the set of variables written to by instruction a and $\mathit{rd}(a)$ the set of variables read by a.

Initially the functions W_w, W_r, R_w and R_r map all variables to Var. At other points in the program their values are defined in terms of allowable instruction reorderings. We define $\mathit{later}_w(a)$ to return the set of variables whose *writes* cannot be reordered before a. Similarly, we define $\mathit{later}_r(a)$ to return the set of variables whose *reads* cannot be reordered before a. For example, $y \in \mathit{later}_w(x := e)$ implies writes of y cannot be reordered before the instruction $x := e$. This will be the case when $y = x$ (due to constraint (i) of Section 2) or $y \in \mathit{var}(e)$ (due to constraint (iii) of Section 2). Similarly, $y \in \mathit{later}_r(x := e)$ implies reads of y cannot be reordered before $x := e$. This will be the case when $y = x$ and e contains global variables (due to the weakened constraint (ii) of Section 2) or $y \in \mathit{var}(e) \cap \mathit{Global}$ (due to constraint (iv) of Section 2). The full definitions are:

$$\begin{aligned} \mathit{later}_w(\mathit{Fence}) &= \mathit{Var} \\ \mathit{later}_w(x := e) &= \{x\} \cup \mathit{var}(e) \\ \mathit{later}_w(b) &= \mathit{var}(b) \cup \mathit{Global} \end{aligned} \qquad \begin{aligned} \mathit{later}_r(\mathit{Fence}) &= \mathit{Var} \\ \mathit{later}_r(b) &= \mathit{var}(b) \cap \mathit{Global} \end{aligned}$$

$$\mathit{later}_r(x := e) = \begin{cases} \{x\} \cup (\mathit{var}(e) \cap \mathit{Global}) & \text{if } \mathit{var}(e) \cap \mathit{Global} \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

where an argument b denotes the guard of an if or while instruction. The 'otherwise' case of the definition of $\mathit{later}_r(x := e)$ allows for forwarding.

Let $f[a]$ denote the update of function f (which may be W_w, W_r, R_w or R_r) when instruction a occurs, and ψ stand for either W or R. Then

$$\psi_w[a](x) = \begin{cases} \psi_w(x) \cup \mathit{known}_\psi(a) & \text{if } x \in \mathit{later}_w(a) \\ \psi_w(x) \setminus \mathit{kill}_\psi(a) & \text{otherwise} \end{cases}$$

$$\psi_r[a](x) = \begin{cases} \psi_r(x) \cup \mathit{known}_\psi(a) & \text{if } x \in \mathit{later}_r(a) \\ \psi_r(x) \setminus \mathit{kill}_\psi(a) & \text{otherwise} \end{cases}$$

where $\mathit{kill}_W(a) = \mathit{wr}(a)$ and $\mathit{kill}_R(a) = \mathit{rd}(a)$.

For example when an instruction a occurs, for any instruction a_1, $\mathit{known}_W(a_1)$ is updated by changes to $W_w(y)$ for any variable y written in a_1, and $W_r(y)$ for any variable y read in a_1. These changes reflect the instruction reorderings captured in $\mathit{later}_w(a)$ and $\mathit{later}_r(a)$. If the instruction a_1 cannot be reordered before a (due to a variable written or read in a_1 being in $\mathit{later}_w(a)$ or $\mathit{later}_r(a)$, respectively) then any writes that are known to have occurred before a will also be known to have occurred before a subsequent instruction a_1. Hence, they are added into $W_w(y)$ or $W_r(y)$. If, on the other hand, a_1 can be reordered before a then any writes in a are removed from those known to have occurred before a_1 (by removing them from both $W_w(y)$ and $W_r(y)$).

5 The Logic

In this section, we present our logic in which a thread c is secure when a judgement $\Gamma, P, D \;\; \{c\}_M \;\; \Gamma', P', D'$ can be derived from the logic's rules under modes M where D is the tuple (W_w, W_r, R_w, R_r) capturing the dependencies between instructions. Initially, $\Gamma(x)$ is *Low* for those stable variables x for which $\mathcal{L}(x)$ is true and *High* otherwise, P is *true*, and all functions in D map each variable to *Var*. In the logic, we let $\mathcal{C} \subseteq \mathit{Var}$ represent the set of control variables.

The rules for skip, sequential composition, if statements and while loops (see Figure 3) are based on those of Murray et al. [19, 20]. The most significant modification is the introduction of P_a, a version of P restricted to the writes which are guaranteed to have occurred prior to reaching instruction a. We define $P_a = P \mid_{\mathit{known}_W(a)}$, where

$$P \mid_S \;\; \widehat{=} \;\; \exists\, y_1, ...y_n \cdot P \;\; \text{where } \{y_1, .., y_n\} = \mathit{Var} \setminus S$$

As in Murray et al. [19, 20], rules with branching conditions restrict the expression to be *Low*. This is necessary to ensure our logic is timing-sensitive (see Section 6). Additionally, we introduce an update to D based on an instruction a, $D[a]$, which updates all of its components as described in Section 4.

These rules also uses the notation $[b]_M$, which is the condition b with all free occurrences of unstable variables removed.

$$[b]_M \;\; \widehat{=} \;\; \exists\, y_1, ...y_n \cdot b$$
$$\text{where } \{y_1, ..., y_n\} = \mathit{Var} \setminus (M(\mathit{AssNoRW}) \cup M(\mathit{AssNoW}))$$

The removal of free occurrences of unstable variables is required as these variables may be changed by another thread at any time (invalidating the relationship between them and stable variables). For example, if variables x and y

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

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

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

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

$$\textsc{Rewrite}\ \frac{\Gamma_1, P_1, D_1\ \ \{c\}_M\ \ \Gamma_1', P_1', D_1' \qquad \begin{array}{c} \Gamma_2 \geq \Gamma_1 \\ P_2 \Rightarrow P_1 \\ D_2 \supseteq D_1 \end{array} \qquad \begin{array}{c} \Gamma_1' \geq \Gamma_2' \\ P_1' \Rightarrow P_2' \\ D_1' \supseteq D_2' \end{array}}{\Gamma_2, P_2, D_2\ \ \{c\}_M\ \ \Gamma_2', P_2', D_2'}$$

$$\textsc{Assign}\ \frac{x \notin \mathcal{C} \qquad \Gamma, P_{x:=e} \vdash e : t \qquad x \notin M(\mathit{AssNoRW}) \Rightarrow t \sqsubseteq \mathit{eval}_{P_{x:=e}}(x)}{\Gamma, P, D\ \ \{x := e\}_M\ \ \Gamma[x \mapsto t], P[x := e]_M, D[x := e]}$$

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

$$\textsc{Fence}\ \frac{}{\Gamma, P, D\ \ \{\mathsf{fence}\}_M\ \ \Gamma, P, D[\mathsf{fence}]}$$

Fig. 3. Rules of the logic.

are stable but z is not, the guard expression $x = y + z$ should add the predicate $\exists\, z \cdot x = y + z$ to P, rather than $x = y + z$.

The Rewrite rule is typically required when using the If and While rules to ensure both branches have corresponding analysis states and to establish loop invariants, respectively. The rule allows for the introduction of stronger Γ, P and D on the left-hand side, and weaker Γ', P' and D' on the right. To express this, we introduce a relation $\supseteq$ between values of D.

$$D \supseteq D' \ \widehat{=}\ \forall\, x : \mathit{Var} \cdot \psi_w(x) \supseteq \psi_w'(x) \wedge \psi_r(x) \supseteq \psi_r'(x) \qquad \text{where } \psi \in \{W, R\}$$

Additionally, we introduce a relation $\geq$ between values of Γ. This relation constrains the entries in the weaker Γ to be higher or equal to those in the stronger, as any expressions that pass the logic's rules with a *High* read will also succeed with a *Low* read. Such a rewriting property allows for branches that consider the same variable at different security levels to merge, rewriting to the highest level.

$$\Gamma \geq \Gamma' \ \widehat{=}\ \operatorname{dom} \Gamma = \operatorname{dom} \Gamma' \wedge \forall\, x : \operatorname{dom} \Gamma \cdot \Gamma(x) \sqsubseteq \Gamma'(x)$$

While the use of the Rewrite rule requires user interaction, its application can be automated based on the context, e.g., through the introduction of a specialised If rule as in Murray et al. [19, 20].

There are two rules for assignment. The first rule, Assign, corresponds to the assignment of an expression e to a non-control variable x. If another thread can read x, the expression's security level should not be higher than x's security level when considered under $P_{x:=e}$.

Γ, P and D are updated to reflect the assignment. The notation $\Gamma[x \mapsto t]$ denotes reassignment, where the function Γ is updated so that x maps to t provided $x \in \operatorname{dom} \Gamma$. For P we use a shorthand that denotes the strongest postcondition, *sp*, of the assignment $x := e$ from a state satisfying P with all free occurrences of unstable variables removed.

$$P[x := e]_M \;\;\widehat{=}\;\; sp(x := e, P) \mid_{M(AssNoRW) \cup M(AssNoW)}$$

The second assignment rule, AssignC, corresponds to an assignment to a control variable. In this case, the expression must have a *Low* security level to conform with the restriction on control variables introduced in Section 4.2. Moreover, the effect of the assignment on the security level of controlled variables must be taken into account. If the security level of a readable controlled variable y falls from *High* to *Low*, it is necessary that any earlier writes to y are guaranteed to have occurred, and that the final such write has set y to *Low*.

If, on the other hand, the security level of y rises from *Low* to *High*, it is necessary that any earlier reads of y are guaranteed to have occurred. To see why this is required, consider the code z:=0; x:=y; z:=1 where $\mathcal{L}(\mathsf{y}) = (z = 0)$ and $\mathcal{L}(\mathsf{x}) = \mathit{true}$. If the assignment z=1 occurs before x=y then another thread may update y to a *High* value before x=y occurs. This would result in a *High* value being passed to x which has a *Low* security level.

The required condition for assignment to control variables is captured by the shorthand $\mathit{secure\_update}_{\Gamma,P,D,M}(x := e)$ defined below.

$$\begin{aligned}
&\mathit{secure\_update}_{\Gamma,P,D,M}(x := e) \;\;\widehat{=}\\
&\quad (\forall\, y : \mathit{falling}(x, P, P') \setminus M(AssNoRW) \cdot\\
&\qquad y \in \mathit{known}_W(x := e) \wedge \Gamma, P' \vdash y : \mathit{Low}) \wedge\\
&\quad (\forall\, y : \mathit{rising}(x, P, P') \cdot y \in \mathit{known}_R(x := e))
\end{aligned}$$

where $P' = P[x := e]_M$ is the predicate after the assignment and

$$\begin{aligned}
\mathit{falling}(x, P, P') \;\;&\widehat{=}\;\; \{y : \mathit{Var} \mid x \in \mathit{var}(\mathcal{L}(y)) \wedge \neg\,(\mathit{low}_P(y)) \wedge \neg\,(\mathit{high}_{P'}(y))\}\\
\mathit{rising}(x, P, P') \;\;&\widehat{=}\;\; \{y : \mathit{Var} \mid x \in \mathit{var}(\mathcal{L}(y)) \wedge \neg\,(\mathit{high}_P(y)) \wedge \neg\,(\mathit{low}_{P'}(y))\}
\end{aligned}$$

The sets *falling* and *rising* identify all variables that could change security level due to the modification of a control variable x. As not all information may be available in P or P' to determine security levels, for soundness the definitions default to assume a change has occurred.

The final rule is for fences. After a fence, it is guaranteed that the earlier reads and writes of all variables have occurred.

5.1 Soundness

The logic has been encoded in Isabelle/HOL [21] and proved sound with respect to an encoding of the operational semantics of ARMv8 [6]. The soundness proof follows the structure of prior proofs for sequentially consistent logics [15, 19, 20] and proves that programs that pass the logic's rules will satisfy a compositional non-interference property. That compositional property requires showing that whenever two copies of the program each perform an execution step from states that agree on the values of *Low* variables, then the resulting states also agree on their *Low* variables. The main extra complexity of the proof concerns the case in which one copy performs a step that is out-of-order. In this case we must prove that the other copy must also perform this out-of-order step. To do so, we encode into the operational semantics the assumption that the choice about *when* to reorder instructions never depends on sensitive information, akin to prior work that made a similar assumption about the thread schedule [19, 20] by quantifying over all deterministic interleavings of threads. The theories are available at https://bitbucket.org/wmmif/wmm-if.

5.2 Example Revisited

```
   writer_thread:                reader_thread:
1  z := 0;                     10  while(true)
2  x := 0;                     11    do
3  while (true)                12      do
4    z := z+1;                 13        r1:= z;
5    fence;                    14      while (r1 % 2 ≠ 0)
6    x := secret;              15      fence;
     ...                       16      r2 := x;
7    x := 0;                   17      fence;
8    fence;                    18    while (z ≠ r1)
9    z := z+1                  19    y := r2
```

Fig. 4. Writer and reader threads using the operation secret_write and read of Figure 2.

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 Figure 4 for which we will assume $M(AssNoW) = \{\mathsf{z}, \mathsf{x}\}$. This thread initialises the variables z and x and then repeatedly calls the secret_write operation of Figure 2. 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 $\Gamma = \{\mathsf{z} \mapsto Low, \mathsf{x} \mapsto Low\}$, $P = (\mathsf{z} = 0 \wedge \mathsf{x} = 0)$, and D comprises $W_w = \{\mathsf{z} \mapsto \{\mathsf{z}\}, \mathsf{x} \mapsto \{\mathsf{x}\}\}$, $W_r = \{\mathsf{x} \mapsto \varnothing, \mathsf{z} \mapsto \varnothing\}$ and $R_w = R_r = \{\mathsf{z} \mapsto \{\mathsf{x}, \mathsf{z}\}, \mathsf{x} \mapsto \{\mathsf{x}, \mathsf{z}\}\}$.

The Rewrite rule can then be applied to weaken P to $\mathsf{z}\,\%\,2{=}0 \wedge \mathsf{x}{=}0$ 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 $known_W(\mathsf{x} := \mathsf{secret})$ and hence not in $P_{x:=secret}$. Therefore, Assign would not be applicable (since 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 $\mathit{secure\_update}_{\Gamma,P_{\mathsf{z:=z+1}},D,M}$. This could not be ascertained, however, since x would not be in known_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 situation for the reader_thread is not as straightforward. Even with the fences (as suggested in Section 3), the logic cannot be used to show that the code is secure. This is because z is not stable and hence the assignment at line 16 cannot guarantee that r2's value is *Low*. Although the logic is sound, it is not precise enough to determine that reader_thread's code is secure.

5.3 A More Precise Logic

The reason that the reader_thread of Figure 4 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. This reliance on stability is common among non-blocking algorithms.

To allow for us to check the security of such algorithms, we allow non-blocking loops, such as the outer loop in reader_thread, to be annotated with a variable which we expect to be stable (z in this example). The annotation allows local reasoning to assume that the nominated variable is stable using the following rule (where c can be a while or do loop).

$$\textsc{NonBlocking}\ \frac{\begin{array}{c}\Gamma, P\ |_{\mathit{known}_W(z)} \vdash z : t \\ \Gamma \cup \{z \mapsto t\}, P, D \ \ \{c\}_{M^z} \ \ \Gamma' \cup \{z \mapsto t'\}, P', D'\end{array}}{\Gamma, P, D \ \ \{c^z\}_M \ \ \Gamma', P', D'}$$

where Γ is updated with a value for z (based on what is known to have occurred if the variable were read) and M is extended to $M^z = M[\mathit{AssNoW} \mapsto M(\mathit{AssNoW}) \cup \{z\}, \mathit{GuarNoW} \mapsto M(\mathit{GuarNoW}) \cup \mathit{Global}]$.

The extension of $\mathit{GuarNoW}$ in M^z 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 z is discovered not to be stable.

For the rule to be sound, we also require that the loop cannot be exited unless the variable is 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 reader_thread, the proof follows from the fact that the value of z is never decreased (as described in Section 3).

6 Timing Sensitivity

In earlier work on information-flow security on weak memory models [14, 26], an auxiliary variable is introduced (called *wt* in [26] and *pt* in [14]) to record the lowest security level of a pending write, i.e., one that has occurred locally but has not necessarily been flushed to global memory. This is argued to be necessary to prevent *Low* variables being flushed on a *High* path (i.e., a path entered depending on the value of a *High* variable) and thus revealing that the program has taken that path. In our logic, we do not allow *High* paths and hence do not require such a variable.

Our 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 [19], this is not possible in the presence of *High* paths. For example, consider the program in Figure 5 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: 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 $high = 1$.

Thread 1:

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

Thread 2:

```
output=low;
```

Fig. 5. Example illustrating the need for timing-sensitive security.

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 *High* paths from code can be achieved using program transformations as described, for example, in [1, 17].

7 Conclusion

In this paper, we have presented the first information-flow logic for the ARMv8 weak memory model; a memory model which is significantly weaker than those such as TSO for which prior information-flow logics have been considered. Our logic supports dynamic, value-dependent security levels and is compositional and timing-sensitive. It has been proven sound with respect to an operational semantics of ARMv8 which has been validated against extensive test suites.

This work, focusing on instruction reordering, is a first step towards a more extensive logic in terms of its coverage of both ARM instructions and behaviours, and potential security vulnerabilities. We also anticipate improving the completeness of the logic, in particular by supporting more general rely/guarantee conditions, and adapting it for other weak memory models including those of IBM POWER and prior versions of ARM.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16*, pages 53–70. USENIX Association, 2016.
2. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 7–18. ACM, 2010.
3. H. Boehm. Can seqlocks get along with programming language memory models? In L. Zhang and O. Mutlu, editors, *Proceedings of the 2012 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '12*, pages 12–20. ACM, 2012.
4. J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018*, pages 991–1008. USENIX Association, 2018.
5. K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.
6. R. J. Colvin and G. Smith. A high-level operational semantics for hardware weak memory models. *CoRR*, abs/1812.00996, 2018.
7. R. J. Colvin and G. Smith. A wide-spectrum language for verification of programs on weak memory models. In K. Havelund, J. Peleska, B. Roscoe, and E. P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018*, volume 10951 of *Lecture Notes in Computer Science*, pages 240–257. Springer, 2018.
8. J. Fitzpatrick. An interview with Steve Furber. *Commun. ACM*, 54(5):34–39, 2011.
9. S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pages 608–621. ACM, 2016.
10. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
11. P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.
12. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018*, pages 973–990. USENIX Association, 2018.
13. L. Lourenço and L. Caires. Dependent information flow types. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 317–328. ACM, 2015.
14. H. Mantel, M. Perner, and J. Sauer. Noninterference under weak memory models. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 80–94. IEEE Computer Society, 2014.

15. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011*, pages 218–232. IEEE Computer Society, 2011.
16. M. Moir and N. Shavit. Concurrent data structures. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications.* Chapman and Hall/CRC, 2004.
17. D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In D. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.
18. T. C. Murray. Short paper: On high-assurance information-flow-secure programming languages. In M. Clarkson and L. Jia, editors, *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2015*, pages 43–48. ACM, 2015.
19. T. C. Murray, R. Sison, and K. Engelhardt. COVERN: A logic for compositional verification of information flow control. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pages 16–30. IEEE, 2018.
20. T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016*, pages 417–431. IEEE Computer Society, 2016.
21. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.
22. C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL*, 2(POPL):19:1–19:29, 2018.
23. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 175–186. ACM, 2011.
24. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
25. D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.
26. J. A. Vaughan and T. D. Millstein. Secure information flow for concurrent programs under Total Store Order. In S. Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 19–29. IEEE Computer Society, 2012.
27. L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007.