1 Information flow logic for low-level concurrent code

Our information flow logic is defined over an abstract language representing assembly language constructs. A concurrent program consists of a set of sequential programs each representing a single thread, the syntax of which is given below.

$$\begin{array}{rcl} \textit{prog} & ::= & \textit{proc}^{+} \\ \textit{proc} & ::= & \textit{procId} : \textit{block}^{+} \\ \textit{block} & ::= & \textit{id} : \textit{instr}^{+} \,;\, \textit{jump} \end{array} \qquad \begin{array}{rcl} \textit{instr} & ::= & r := e \mid r := x \mid x := e \\ \textit{jump} & ::= & \textit{uncond\_jump} \mid \textit{cond\_jump} \\ \textit{cond\_jump} & ::= & \mathsf{if}\,(b)\ \textit{uncond\_jump}\ \mathsf{else}\ \textit{jump} \\ \textit{uncond\_jump} & ::= & \mathsf{goto}\ \textit{id} \mid \mathsf{call}\ \textit{procId} \mid \mathsf{indCall}\ r \end{array}$$

A *program* is defined in terms of a non-empty set of procedures: an entry procedure, and zero or more procedures reachable from it through calls. Each *procedure* consists of an identifier and a non-empty set of basic *blocks*, including an entry block. Each basic *block* comprises an identifier and a non-empty sequence of non-branching instructions, optionally followed by either an unconditional jump to a basic block or a conditional jump to one of two or more basic blocks.

Non-branching instructions include writes to registers (denoted by r, and in the following also referred to as *local variables*), and loads from and stores to locations in globally accessible memory (which includes the stack, heap and the data section of the memory, potentially shared between threads). In the following we refer to a memory location as a *global variable*. The right-hand side of local writes and stores is an expression e in terms of local variables and literals.

A conditional jump performs an unconditional jump when its condition b is true, and otherwise performs either an unconditional or conditional jump, the latter allowing more than two possible jump targets overall. Note that the condition, b is in terms of local variables and literals only.

An unconditional jump can be a simple goto instruction to the next basic block, a direct function call to a procedure, or an indirect function call in which the target of the call is encoded in a register r.

1.1 Sequential Logic: wp_{if}

Our logic is based on that of Winter et al. [8] which uses weakest precondition reasoning [4] to calculate the weakest condition that needs to hold at the beginning of a program to ensure it is secure.

The security level of information is measured (for simplicity of presentation) by a simple Boolean lattice where *true* indicates low, or public, information and *false* represents information which is high, or sensitive.[1] We specifiy for each variable the highest level of information that can be securely held by this location via its *security classification*. We assume that variables with a low classification are visible to an attacker, and high variables are hidden from attackers. Additionally, we track the flow of the *security values* for each variable which indicate whether their content is sensitive or not (similar to taint analysis).

We assume that the security classification is provided by the user. For each global variable (i.e., memory location) v, it is specified by $\mathcal{L}(v)$. The classification

[1] The approach is easily extended to general lattices (see [8]).

can be *value-dependent*, i.e., it can dynamically change depending on other global variables referred to as *control variables.* That is, $\mathcal{L}(v)$ is given as a predicate over global variables which indicates a low classification when the predicate evaluates to *true* in the current state (i.e., v is considered visible to an attacker), otherwise it is high. The classification of local variables (i.e., registers not shared with other threads) is assumed high, i.e., private.

For each variable v (local or global) we introduce an auxiliary variable $\mathit{\Gamma}_v$ to model the security value of the current content of v. It is updated whenever v is updated. In our simple lattice $\mathit{\Gamma}_v$ is a Boolean variable, corresponding to whether the information currently held in v is low or not. In a secure system, the security value $\mathit{\Gamma}_v$ must be low whenever v's security classification is low, i.e., $\mathcal{L}(v) \Rightarrow \mathit{\Gamma}_v$ is an invariant. Since we assume that local variables are not accessible by an attacker but private, they are classified as high (i.e., $\mathcal{L}(v) = \mathit{false}$) and the invariant holds by default. Thus it suffices to check the invariant on global variables only. The security value of an expression $\mathit{\Gamma}_E(e)$ is calculated as the conjunction of the security values of the local variables it references. Hence, it will be high when any of those variables holds high information.

In this section, we adapt this logic to the abstract language introduced earlier to render it applicable to low-level unstructured code as provided by the decompiler. Following Barnett and Leino [1], we apply weakest precondition reasoning to each basic block: if the basic block does not jump to another basic block we assume a postcondition *true*, otherwise the postcondition is the conjunction of the weakest preconditions of the blocks to which it jumps, i.e., we require that jump instructions to a block *id* ensure that its weakest precondition holds.

Starting at the bottom of a basic block and working backwards one instruction or jump at a time, our analysis transforms its postcondition according to the *predicate transformer for information flow* $\mathit{wp}_{\mathit{if}}$[2] which computes the weakest precondition *wp* and adds a bespoke proof obligation *po* to guarantee confidentiality (i.e., the absence of information leakage). Given a secure state, i.e., one where $\mathcal{L}(x) \Rightarrow \mathit{\Gamma}_x$ for all global variables, the proof obligation ensures that security is maintained when the instruction is executed. Hence, the failure of a proof obligation indicates that executing the associated instruction may be insecure. In the following definitions, the proof obligation is omitted when it is *true*.

For our abstract language we define the predicate transformer $\mathit{wp}_{\mathit{if}}$ as follows. For a write to a local variable r, we have no additional proof obligation since no information is put into memory accessible by another thread. The predicate transformer, however, deviates from the standard definition of weakest preconditions to include the update to auxiliary variable $\mathit{\Gamma}_r$ that is required to trace security values. At this abstract level this is implicitly covered through the substitutions in the postconditions,

$$\mathit{wp}_{\mathit{if}}(r := e, Q) \ \widehat{=}\ Q[r, \mathit{\Gamma}_r \backslash e, \mathit{\Gamma}_E(e)] \tag{1}$$

[2] Note the subscript *if* is stands for information flow and is part of the transformer's name (not a parameter).

where Q is the predicate that needs to hold following the instruction, and $Q[r, \Gamma_r \backslash e, \Gamma_E(e)]$ denotes the replacement of each occurrence of r and Γ_r in Q with e and $\Gamma_E(e)$, respectively.

For a load of global variable x to r, we take into account the fact that we are in a secure state before the load is executed. Hence, the value of Γ_x is evaluated under the assumption that $\mathcal{L}(x) \Rightarrow \Gamma_x$, i.e., we use $(\mathcal{L}(x) \Rightarrow \Gamma_x) \Rightarrow \Gamma_x$ as the security value of x. This simplifies to $\mathcal{L}(x) \vee \Gamma_x$ as follows.

$$\begin{array}{lr}
(\mathcal{L}(x) \Rightarrow \Gamma_x) \Rightarrow \Gamma_x & \\
\Leftrightarrow \neg\,(\neg\,\mathcal{L}(x) \vee \Gamma_x) \vee \Gamma_x & \text{(material implication)} \\
\Leftrightarrow (\mathcal{L}(x) \wedge \neg\,\Gamma_x) \vee \Gamma_x & \text{(De Morgan's law)} \\
\Leftrightarrow (\mathcal{L}(x) \vee \Gamma_x) \wedge (\neg\,\Gamma_x \vee \Gamma_x) & \text{(distribution)} \\
\Leftrightarrow \mathcal{L}(x) \vee \Gamma_x & \text{(complement law)}
\end{array}$$

which results in

$$wp_{if}(r := x, Q) \;\widehat{=}\; Q[r, \Gamma_r \backslash x, \mathcal{L}(x) \vee \Gamma_x]. \tag{2}$$

For a store to x, we need to check that the security value of the information being stored is low when the security classification of x is low. Additionally, having value-dependent security classifications, we need to take into account that changing x may change the security classification of one or more other global variables. These conditions are captured in $po_1(x := e)$ and $po_2(x := e)$.

$$\begin{array}{rrcl}
 & wp_{if}(x := e, Q) & \widehat{=} & Q[x, \Gamma_x \backslash e, \Gamma_E(e)] \wedge po_1(x := e) \wedge po_2(x := e) \\
\text{with} & po_1(x := e) & \widehat{=} & (\mathcal{L}(x) \Rightarrow \Gamma_E(e)) \\
 & po_2(x := e) & \widehat{=} & \forall\, y \cdot x \in vars(\mathcal{L}(y)) \Rightarrow (\mathcal{L}(y)[x \backslash e] \Rightarrow (\mathcal{L}(y) \vee \Gamma_y))
\end{array} \tag{3}$$

where $vars(\mathcal{L}(y))$ denotes the control variables of y which determine its (value-dependent) security classification. If x is a control variable of y then we need to check that if the security classification of y after e has been stored at x is low, i.e., $\mathcal{L}(y)[x \backslash e]$, then either the security classification of y was already low before the update or the current value of Γ_y is low, i.e., $\mathcal{L}(y) \vee \Gamma_y$.

For a sequence of instructions and jumps, the conjunction of the weakest precondition of each instruction or jump (including their proof obligation) is passed on as the postcondition for the preceding instructions.

$$wp_{if}(i_1 \,;\, i_2, Q) \;\widehat{=}\; wp_{if}(i_1, wp_{if}(i_2, Q)) \tag{4}$$

A goto needs to ensure id_{ok}, the weakest precondition of the block it jumps to.

$$wp_{if}(\mathsf{goto}\ id, Q) \;\widehat{=}\; id_{ok} \tag{5}$$

Each function f will have a precondition $pre(f)$ and postcondition $post(f)$ capturing any changes to security values and values of global variables which may affect security classifications. A function's postcondition holds only when its precondition is true. Hence, for a function call to result in a predicate Q, its precondition must be true and its postcondition must imply Q. A postcondition may include both the pre- and post-state values of variables $\bar{v}$ occurring in the function. We use $\bar{v}'$ to represent the post-state values of variables $\bar{v}$ and Q' to denote the predicate $Q[\bar{v} \backslash \bar{v}']$.

$$wp_{if}(\mathsf{call}\ f, Q) \ \widehat{=}\ pre(f) \wedge (\forall\, \bar{v}' \cdot post(f) \Rightarrow Q') \tag{6}$$

The rule for an indirect function call is given in terms of that for a direct call and a function *eval* which evaluates the content of a register to a procedure identifier, i.e., $eval(r)$ represents a procedure identifier/address.

$$wp_{if}(\mathsf{indCall}\ r, Q) \ \widehat{=}\ wp_{if}(\mathsf{call}\ eval(r), Q) \tag{7}$$

A conditional jump transforms the postcondition Q according to the weakest precondition and proof obligation of the jump taken. A proof obligation ensures that the branching condition is not high (as required to avoid timing side-channels on concurrent programs [6]).

$$\begin{aligned} wp_{if}(\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,) \ &\widehat{=}\ (b \Rightarrow wp_{if}(j_1, Q)) \wedge (\neg\, b \Rightarrow wp_{if}(j_2, Q)) \wedge \\ &\qquad po(\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,) \\ \text{with}\quad po(\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,) \ &\widehat{=}\ \Gamma_E(b) \end{aligned} \tag{8}$$

1.2 Concurrent Logic: wp_{if}^{RG}

To account for interference from other threads in our analysis, we use rely/guarantee reasoning [5,9]. Each thread t has a rely condition $\mathcal{R}$, a predicate over two states of the global memory, which describes the transitions that may be made by the threads in t's environment. The rely condition is reflexive, since the environment may make no changes to the memory, and transitive, so that it can capture the environment making more than one change. Each thread in t's environment must guarantee that it satisfies $\mathcal{R}$. Similarly, t must guarantee that it satisfies the rely conditions of all threads in its environment. This guarantee is captured by an additional two-state predicate $\mathcal{G}$ which must hold for instructions which can change memory, i.e., store instructions.

Given such a rely and guarantee condition, two additional proof obligations for a store are required, $po_G(x := e)$ and $po_R(x := e, Q)$. The former checks whether the execution of $x := e$ will satisfy $\mathcal{G}$, and the latter ensures that the weakest precondition of the store (given postcondition Q) and its proof obligation are *stable* and thus not affected by any changes to memory satisfying $\mathcal{R}$.

To include these proof obligations we extend wp_{if} to the transformer wp_{if}^{RG} for concurrent programs[3]. Given $stable_{\mathcal{R}}(P) \mathrel{\widehat{=}} \forall\, v' \cdot P \wedge \mathcal{R} \Rightarrow P'$, we have

$$\begin{aligned} wp_{if}^{RG}(x := e, Q) \ &\widehat{=}\ wp_{if}(x := e, Q) \wedge po_G(x := e) \wedge po_R(x := e, Q) \\ \text{with}\qquad po_G(x := e) \ &\widehat{=}\ \mathcal{G}[x' \backslash e] \\ po_R(x := e, Q) \ &\widehat{=}\ stable_{\mathcal{R}}(wp_{if}(x := e, Q) \wedge po_G(x := e)) \end{aligned} \tag{9}$$

[3] Note that sub- and superscript are part of the transformer's name, not parameters.

A load does not change global memory and hence trivially satisfies the guarantee. It may add constraints on global variables to the transformed predicate, however, and hence a check on its stability under $\mathcal{R}$ is required.

$$\begin{array}{rrcl} & wp_{\mathit{if}}^{RG}(r := x, Q) & \widehat{=} & wp_{\mathit{if}}(r := x, Q) \wedge po_R(r := x, Q) \\ \text{with} & po_R(r := x, Q) & \widehat{=} & stable_{\mathcal{R}}(wp_{\mathit{if}}(r := x, Q)) \end{array} \tag{10}$$

All other instructions i, apart from direct calls which are discussed below, are not affected by and do not create environmental interference.

$$wp_{\mathit{if}}^{RG}(i, Q) \mathrel{\widehat{=}} wp_{\mathit{if}}(i, Q) \tag{11}$$

Functions are accompanied by generic rely and guarantee conditions, $\mathcal{R}_f$ and $\mathcal{G}_f$, which are *context-insensitive*, i.e., independent of the calling context in which the function is used. It is assumed (or where function code is available, previously proven) that function f satisfies its given specification, $pre(f)$ and $post(f)$, under the generic rely and guarantee conditions, $\mathcal{R}_f$ and $\mathcal{G}_f$ respectively.

For reasoning over a program c, that calls a function f, to be sound the analysis needs to ensure that c's rely and guarantee, $\mathcal{R}_c$ and $\mathcal{G}_c$, satisfy $\mathcal{R}_c \Rightarrow \mathcal{R}_f$ and $\mathcal{G}_f \Rightarrow \mathcal{G}_c$ [7]. However, to improve on the precision of the analysis (and achieve better completeness and thus a lower false positive rate), we weaken the above constraint on $\mathcal{R}_c$ to $Inv \Rightarrow (\mathcal{R}_c \Rightarrow \mathcal{R}_f)$, where Inv is defined as the part of the calling context that is invariant over the caller's rely $\mathcal{R}_c$ and the guarantee of the function $\mathcal{G}_f$ as further explained in the following.

Assuming P describes the calling context (i.e., the pre-state under which f is called) in terms of state variables $\bar{v}$ then the invariant is captured by P and all states $\sigma(\bar{v}')$ reachable from P through steps permitted by $\mathcal{R}_c$ or $\mathcal{G}_f$, i.e., steps that can be performed by the caller's environment or function f. Provided that $\mathcal{R}_c \vee \mathcal{G}_f$ is reflexive and transitive, we have

$$P \wedge (\forall\, \bar{v}' \cdot \mathcal{R}_c \vee \mathcal{G}_f \Rightarrow (\forall\, \bar{v}'' \cdot (\mathcal{R}_c \Rightarrow \mathcal{R}_f)[\bar{v}, \bar{v}' \backslash \bar{v}', \bar{v}''])) \tag{12}$$

where, due to the variable replacement, $\mathcal{R}_c \Rightarrow \mathcal{R}_f$ is interpreted on pre-state $\sigma(\bar{v}')$ corresponding to the invariant, and a post-state of the invariant $\sigma(\bar{v}'')$. This reasoning with context-insensitive rely conditions has been proven sound in Isabelle/HOL [2] for the abstract rely/guarantee proof system of [3]. Note that reflexivity of $\mathcal{R}_c \vee \mathcal{G}_f$ follows from the fact that $\mathcal{R}_c$ and $\mathcal{G}_f$ are both reflexive. Transitivity, however, must be proven separately.

For example, let function `memset` update global variable buf, and its rely condition assume that buf is not modified by another thread, $\mathcal{R}_{memset} = (buf = buf')$. Furthermore, `memset` guarantees not to change the global variable $lock : Tid$ which synchronises the access to global variables between the threads $t_i \in Tid$ (i.e., $lock = t_c$ denotes that thread t_c holds the lock). That is, we have $\mathcal{G}_{memset} = (lock = lock')$. Assume the caller is part of thread t_c and its rely condition states that the environment never changes buf when t_c holds the lock, i.e, $\mathcal{R}_c = (lock = t_c \Rightarrow (buf = buf' \wedge lock = lock'))$. As can be seen $\mathcal{R}_c \Rightarrow \mathcal{R}_{memset}$ does not hold for such a setup. Instead, we can show that $\mathcal{R}_c \vee \mathcal{G}_{memset}$

is transitive and, deriving $Inv = (lock = t_c)$ from the calling context $P = (lock = t_c)$, $\mathcal{R}_c$ and $\mathcal{G}_{memset}$, we can show that $(lock = c) \Rightarrow (\mathcal{R}_c \Rightarrow \mathcal{R}_{memset})$. That is, when called in a context where $lock = t_c$, reasoning over the call to `memset` using the caller's rely condition $\mathcal{R}_c$ is sound.

When reasoning over a call to f, the second conjunct of (12) is added as a proof obligation $po_{RG}(\mathsf{call}\ f)$. The first conjunct of (12), the calling context P, is the weakest precondition that has to hold before $\mathsf{call}\ f$ in order to establish postcondition Q, i.e., $wp_{if}(\mathsf{call}\ f, Q)$:

$$\begin{aligned} & wp_{if}^{RG}(\mathsf{call}\ f, Q) \ \widehat{=}\ wp_{if}(\mathsf{call}\ f, Q) \wedge po_{RG}(\mathsf{call}\ f) && (13) \\ \text{with} \quad & po_{RG}(\mathsf{call}\ f) \ \widehat{=}\ \forall\, \bar{v}' \cdot \mathcal{R}_c \vee \mathcal{G}_f \Rightarrow (\forall\, \bar{v}'' \cdot (\mathcal{R}_c \Rightarrow \mathcal{R}_f)[\bar{v}, \bar{v}' \backslash \bar{v}', \bar{v}'']) \end{aligned}$$

2 Implementation in Boogie

Boogie verifies the functional correctness of programs using standard weakest precondition reasoning. It supports low-level, unstructured code via goto commands [1]. Following ideas proposed by Smith [7], we can encode programs in Boogie to also check thread-local information flow based on rely/guarantee reasoning, specifically the information flow logic defined in Section 1.

2.1 Translating Proof Obligations

We define the encoding as a mapping $[\![\cdot]\!]$ from the abstract language in Section 1 to instrumented Boogie code[4]. This encoding allows us to use the standard weakest precondition calculation wp (as implemented in Boogie) in place of our transformer wp_{if}^{RG} in Section 1.

The encoding maintains the structure of the abstract program in terms of procedures and blocks, but transforms instructions to allow analysis by Boogie. This analysis requires additional definitions corresponding to parts of the specification: a procedure *rely* and a two-state predicate $\mathcal{G}$, along with checks on their reflexivity and transitivity (following [7]), and a function L to model the security classifications. Also for each function f called by the program, procedures Inv_f and Ref_f (defined below) are added to facilitate checking the proof obligation for a direct call (13).

The soundness of this approach is proven with respect to the soundness of wp_{if}^{RG} [8,2] for each instruction in the abstract language. For the sake of readability we intersperse the definition of the encoding $[\![\cdot]\!]$ with the proof.

Theorem 1. *(Soundness)* $\quad wp_{if}^{RG}(\alpha, Q) = wp([\![\alpha]\!], Q)$

Proof *(by cases)*

[4] Note that some Boogie syntax is captured here by standard mathematical notation for the sake of presentation.

Local assignment:

$$[\![r := e]\!] \mathrel{\widehat{=}} r, \mathit{\Gamma}_r := e, \mathit{\Gamma}_E(e) \tag{14}$$

Given the Boogie notation $x, y := E, F$ denotes the simultaneous assignment of E to x and F to y, we trivially have

$$\begin{aligned} wp_{\mathit{if}}^{RG}(r := e, Q) &= Q[r, \mathit{\Gamma}_r \backslash e, \mathit{\Gamma}_E(e)] \\ &= wp([\![r := e]\!], Q) \end{aligned}$$

Memory Load:

$$\begin{aligned} [\![r := x]\!] \mathrel{\widehat{=}} & \ \mathsf{call}\ \mathit{rely}()\,; \\ & r, \mathit{\Gamma}_r := x, (L(x) \vee \mathit{\Gamma}_x) \end{aligned} \tag{15}$$

Load (15) instructions are encoded to explicitly update the security value of the register according to the security value of the loaded memory location. Additionally the encoding needs to take potential interference from other threads into account (as outlined in Section 1.2). This is handled via a stability check.

Stability check: Recall that the application of wp_{if}^{RG} to a load results in the additional proof obligation po_R which is of the form $P \wedge \mathit{stable}_{\mathcal{R}}(P)$. In the translation we add a call to the procedure *rely* which nondeterministically updates the current state according to the rely condition $\mathcal{R}$, i.e., $\mathit{pre}(\mathit{rely}) = \mathit{true}$ and $\mathit{post}(\mathit{rely}) = \mathcal{R}$. As shown below, the use of the call to $\mathit{rely}()$ adds po_R to the current weakest precondition P.

$$\begin{aligned} wp(\mathsf{call}\ \mathit{rely}(), P) = &\ \forall\, v' \cdot \mathcal{R} \Rightarrow P' \\ &= P \wedge \forall\, v' \cdot \mathcal{R} \Rightarrow P' && \{\mathcal{R} \text{ reflexive}\} \\ &= P \wedge \forall\, v' \cdot P \wedge \mathcal{R} \Rightarrow P' && \{v' \text{ not free in } P\} \\ &= P \wedge \mathit{stable}_{\mathcal{R}}(P) && \{\text{definition of } \mathit{stable}_{\mathcal{R}}\} \end{aligned}$$

Hence

$$\begin{aligned} wp_{\mathit{if}}^{RG}(r := x, Q) &= wp_{\mathit{if}}(r := x, Q) \wedge po_R(r := x, Q) \\ &= Q[r, \mathit{\Gamma}_r \backslash x, \mathcal{L}(x) \vee \mathit{\Gamma}_x] \wedge \mathit{stable}_{\mathcal{R}}(Q[r, \mathit{\Gamma}_r \backslash x, \mathcal{L}(x) \vee \mathit{\Gamma}_x]) \\ &= wp(\mathsf{call}\ \mathit{rely}, Q[r, \mathit{\Gamma}_r \backslash x, \mathcal{L}(x) \vee \mathit{\Gamma}_x]) \\ &= wp([\![r := x]\!], Q). \end{aligned}$$

Memory Store:

Memory stores create more complex proof obligations as defined in (3) ensuring that the classification of the updated variable is high enough to hold the value to be stored ($po_1(x := e)$), as well as ensuring that the classification of all variables controlled by the updated one are still secure ($po_2(x := e)$). We encode a memory store in Boogie as follows.

$$[\![x := e]\!] \mathrel{\widehat{=}} \mathsf{call}\ \mathit{rely}()\,; \tag{16}$$

$$\begin{array}{l}\textsf{assert}\ (po_1^B(x := e))\,;\\ \bar{u}_{old} := \bar{u}\,;\ \bar{v}_{old} := \bar{v}\,;\\ x, \mathit{\Gamma}_x := e, \mathit{\Gamma}_E(e)\,;\\ \textsf{assert}\ (po_2^B(x := e))\,;\\ \textsf{assert}\ (\mathcal{G}[\bar{v}, \bar{v}'\backslash\bar{v}_{old}, \bar{v}])\end{array}$$

with $po_1^B(x := e)\ \widehat{=}\ L(x) \Rightarrow \mathit{\Gamma}_E(e)$ and
$po_2^B(x := e)\ \widehat{=}\ \forall\, y.\, (x \in vars(L(y)) \Rightarrow L(y) \Rightarrow (L(y)[\bar{u}\backslash\bar{u}_{old}] \vee \mathit{\Gamma}_y))$.

Given the weakest precondition of an assertion, $wp(\textsf{assert}\ A, Q) = A \wedge Q$, we encode the required proof obligations $po_1(x := e)$ and $po_2(x := e)$ as assertions. The latter appears after the store in (16) but captures the property that needs to hold before the store as folllows. Let $\bar{u}$ denote the list of variables which appear in the security classification for any y, i.e., in $vars(L(y))$. Then $\bar{u}_{old}$ are the same variables before the store. For all y where $x \in vars(L(y))$, $L(y)[\bar{u}\backslash\bar{u}_{old}]$ refers to the classification of y before the update of (control variable) x (i.e., $L(y)[\bar{u}\backslash\bar{u}_{old}] = \mathcal{L}(y)$) and $L(y)$ to the classification after the update (i.e., $L(y) = \mathcal{L}(y)[x\backslash e]$). Hence, with $c = \bar{u}_{old} := \bar{u};\ \bar{v}_{old} := \bar{v};\ x, \mathit{\Gamma}_x := e, \mathit{\Gamma}_E(e)$, we have $wp(c, po_2^B(x := e)) = po_2(x := e)$. Note that this holds whether or not $\bar{v}$ (described below) includes variables in $\bar{u}$.

Additional to the encoding of proof obligations, the encoding of a memory store needs to include a stability check (identical to the check in the encoding of a memory load) as well as a guarantee check which is encoded as follows.

Guarantee check: Like proof obligation po_2^B, the guarantee check is encoded after the store in (16). Let $\bar{v}$ denote the variables that appear in the guarantee relation $\mathcal{G}$. Then $\bar{v}_{old}$ are the same variables before the store and thus $wp(c, \mathcal{G}[\bar{v}, \bar{v}'\backslash\bar{v}_{old}, \bar{v}]) = po_G(x := e)$ where c is defined as above.
Hence each of the proof obligations and rely/guarantee checks are captured by our encoding and we can conclude that

$$\begin{aligned} wp_{if}^{RG}(x := e, Q) &= wp_{if}(x := e, Q) \wedge po_G(x := e) \wedge po_R(x := e, Q)\\ &= wp(\textsf{call}\ rely, Q[x, \mathit{\Gamma}_x\backslash e, \mathit{\Gamma}_E(e)] \wedge po_1^B(x := e) \wedge\\ &\qquad\qquad wp(c, po_2^B(x := e)) \wedge wp(c, \mathcal{G}[\bar{v}, \bar{v}'\backslash\bar{v}_{old}, \bar{v}]))\\ &= wp([\![x := e]\!], Q)\end{aligned}$$

Goto Instructions:

$$[\![\textsf{goto}\ id]\!]\ \widehat{=}\ \textsf{goto}\ id \tag{17}$$

The weakest precondition of a goto [1] is identical to that under wp_{if}^{RG} (5), i.e., we trivially have

$$wp_{if}^{RG}(\textsf{goto}\ id, Q) = wp([\![\textsf{goto}\ id]\!], Q).$$

Conditional Jumps:

Conditional jumps are similarly defined in terms of Boogie's goto command.

$$[\![\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,]\!] \mathrel{\widehat{=}} \mathsf{assert}\ (\mathit{\Gamma}_E(b))\,; \qquad \mathsf{goto}\ id_1, id_2 \tag{18}$$

where id_1 identifies the basic block $\quad id_1 : \mathsf{assume}\ b\,;\ [\![j_1]\!]$,
and id_2 the basic block $\quad id_2 : \mathsf{assume}\ \neg\, b\,;\ [\![j_2]\!]$.

The Boogie notation $\mathsf{goto}\ id_1, \ldots, id_n$ defines a nondeterministic choice between target blocks $id_1, \ldots, id_n$. As such $wp(\mathsf{goto}\ id_1, \ldots, id_n, Q) = wp(\mathsf{goto}\ id_1, Q) \wedge \ldots \wedge wp(\mathsf{goto}\ id_n, Q)$.

The assertion in (18) captures the proof obligation $po(\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,)$ and, given that $wp(\mathsf{assume}\ A, Q) = A \Rightarrow Q$, we can conclude that

$$\begin{aligned}
wp^{RG}_{\mathit{if}}(\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,, Q) = {} & (b \Rightarrow wp_{\mathit{if}}(j_1, Q)) \wedge (\neg\, b \Rightarrow wp_{\mathit{if}}(j_2, Q)) \wedge \\
& po(\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,, Q) \\
= {} & (b \Rightarrow wp([\![j_1]\!], Q)) \wedge (\neg\, b \Rightarrow wp([\![j_2]\!], Q)) \wedge \mathit{\Gamma}_E(b) \\
= {} & wp(\mathsf{assert}\ (\mathit{\Gamma}_E(b)), wp(\mathsf{goto}\ id_1, Q) \wedge wp(\mathsf{goto}\ id_2, Q)) \\
= {} & wp([\![\mathsf{if}\,(b)\,j_1\,\mathsf{else}\,j_2\,]\!], Q)
\end{aligned}$$

Direct Call instructions:

$$\begin{aligned}
[\![\mathsf{call}\ f]\!] \mathrel{\widehat{=}} \mathit{if}\,(*)\{\ & \mathsf{call}\ \mathit{Inv}_f\,; \\
& \mathsf{call}\ \mathit{Ref}_f\,; \\
& \mathsf{assert}\ (\mathit{false})\,;\quad \} \\
\mathsf{call}\ f &
\end{aligned} \tag{19}$$

Reasoning over direct calls to a procedure f requires the proof obligation $po_{RG}(\mathsf{call}\ f)$ (13) to relate the rely condition of the caller $\mathcal{R}_c$ to the rely condition of the callee $\mathcal{R}_f$. (19) encodes this by asserting that any state contradicting po_{RG} is unreachable. These states are established within a branch body by calls to two Boogie procedures, Inv_f and Ref_f. Calling Inv_f assumes any state reachable from the current state via the caller's rely and the callee's guarantee (12), i.e., $\mathit{pre}(\mathit{Inv}_f) = \mathit{true}$ and $\mathit{post}(\mathit{Inv}_f) \mathrel{\widehat{=}} \mathcal{R}_c(\bar{v}, \bar{v}') \vee \mathcal{G}_f(\bar{v}, \bar{v}')$. Calling Ref_f assumes the negation of the implication $(\mathcal{R}_c \Rightarrow \mathcal{R}_f)$ used in (12), i.e., $\mathit{post}(\mathit{Ref}_f) \mathrel{\widehat{=}} \neg(\mathcal{R}_c(\bar{v}, \bar{v}') \Rightarrow \mathcal{R}_f(\bar{v}, \bar{v}'))$. The two calls are followed by an assertion of false to establish unreachability. This encoding of $po_{RG}(\mathsf{call}\ f)$ in Boogie avoids the nested quantification over the post-states in Eqn (13) which simplifies the SMT reasoning within Boogie. The branch is on an unspecific condition (denoted $*$ in Boogie) which allows the backwards reasoning to proceed in all possible states despite the false assertion. Let $c = \mathsf{call}\ \mathit{Inv}_f\,;\ \mathsf{call}\ \mathit{Ref}_f\,;\ \mathsf{assert}\ (\mathit{false})$. Then

$$\begin{aligned}
wp(\mathit{if}(*)\{c\}\,;\,\mathsf{call}\ f, Q) &= wp(c, wp(\mathsf{call}\ f, Q)) \wedge wp(\mathsf{call}\ f, Q) \\
wp(c, wp(\mathsf{call}\ f, Q)) &= \forall\, v'.\ \mathit{post}(\mathit{Inv}_f(v, v')) \Rightarrow \\
&\qquad (\forall\, v''.\ \mathit{post}(\mathit{Ref}_f(v, v'))[v, v' \backslash v', v''] \Rightarrow \mathit{false}) \\
&\quad \{\text{since } \mathit{false} \wedge wp(\mathsf{call}\ f, Q) = \mathit{false}\ \}
\end{aligned}$$

$$
\begin{aligned}
&= \forall\, v'.\ \mathcal{R}_c(\bar{v}, \bar{v}') \vee \mathcal{G}_f(\bar{v}, \bar{v}') \Rightarrow \\
&\qquad (\forall\, v''.\ (\mathcal{R}_c(\bar{v}, \bar{v}') \Rightarrow \mathcal{R}_f(\bar{v}, \bar{v}'))[v, v' \backslash v', v'']) \\
&= po_{RG}(\mathsf{call}\, f, Q) \\
wp(\mathsf{call}\, f, Q) &= wp_{\mathit{if}}(\mathsf{call}\, f, Q) \\
&\qquad \{\text{since } wp(\mathsf{call}\, f, Q) = pre(f) \wedge (\forall\, \bar{v}' \cdot post(f) \Rightarrow Q')\}
\end{aligned}
$$

Hence, it follows that

$$
\begin{aligned}
wp_{\mathit{if}}^{RG}(\mathsf{call}\, f, Q) &= wp_{\mathit{if}}(\mathsf{call}\, f, Q) \wedge po_{RG}(\mathsf{call}\, f, Q) \\
&= wp([\![\mathsf{call}\, f]\!], Q).
\end{aligned}
$$

Indirect Call instructions:

Let the set of potential targets of an indirect call, determined by over-approximating static analyses, be $\{t_1, \ldots, t_{n-1}\}$. We then define

$$[\![\mathsf{indCall}\ r]\!] \mathrel{\widehat{=}} \mathsf{goto}\ id_1, \ldots, id_n \tag{20}$$

where id_i for all $i \in \{1, \ldots, n-1\}$, and id_n respectively identify the basic blocks

$$
\begin{aligned}
id_i &: \mathsf{assume}\ (eval(r) = t_i)\ ;\ \mathsf{call}\ t_i \\
id_n &: \mathsf{assume}\ (eval(r) \notin \{t_1, \ldots, t_{n-1}\})\ ;\ \mathsf{assert}\ (false)
\end{aligned}
$$

Each of the blocks which contain a jump to potential target t_i use an assume statement to guard the jump with the predicate that r holds the address value of the target. The default block where $eval(r) \notin \{t_1, \ldots, t_{n-1}\}$ follows its assumption with an assertion of *false*. This asserts that $\{t_1, \ldots, t_{n-1}\}$ enumerates at least all the possible targets of the indirect call. We have that

$$
\begin{aligned}
wp_{\mathit{if}}^{RG}(\mathsf{indCall}\ r, Q) &= wp_{\mathit{if}}^{RG}(\mathsf{call}\ eval(r), Q) \\
&= wp([\![\mathsf{call}\ eval(r)]\!], Q) \\
wp([\![\mathsf{indCall}\ r]\!], Q) &= (eval(r) = t_1 \Rightarrow wp([\![\mathsf{call}\ t_1]\!], Q)) \wedge \ldots \wedge \\
&\quad\ (eval(r) = t_{n-1} \Rightarrow wp([\![\mathsf{call}\ t_{n-1}]\!], Q)) \wedge \\
&\quad\ (eval(r) \notin \{t_1, ..., t_{n-1}\} \Rightarrow false)
\end{aligned}
$$

which are equal when $eval(r)$ is one of the targets $t_1, \ldots, t_{n-1}$. If Boogie reasoning reaches the default case, the analysis reports an error which maintains soundness and additionally indicates (to the BASIL user) that the call resolution performed by the static analyses needs to be refined. $\square$

2.2 From IL to Boogie: a simple example

We illustrate the Boogie encoding of our logic via the program *secret_write* (shown in Fig. 1). This program allows a classified value held in global variable `secret` to be shared with another thread by storing it in global variable `x`, provided global variable `z` (which is initialised to 0) is odd. A concurrent reader thread (not shown) outputs values of `x` only when read while `z` is even.

```
int z = 0;
int x;
int secret;

int main() {
  z = z + 1;
  x = secret;
//concurrent access
  x = 0;
  z = z + 1;
}
```

```
Globals: x, z, secret: int
   L: z -> true,   secret -> false,
      x -> z mod 2bv32 == 0bv32
Rely: z == old(z)
      && old(Gamma_x) ==> Gamma_x
Guarantee: z >= old(z)

Subroutine: main
Requires: Gamma_x == true
         && Gamma_z == true
         && Gamma_secret == false
         && z == 0bv32
```

Fig. 1. C source code and specification for secret_write

```
1  procedure main {
2    lmain:
3      ...
4      R0 := 131072bv64                       // R0 = baseAddr
5      R0 := (R0 add 20bv64)                  // R0 = &z
6      R1 := ZeroExtend((R0[32:0] add 1bv32), 32) // R1 = z + 1
7      ...
8      mem[R0] := R1[32:0]                    // z = R1
9      ...
10   }
```

Fig. 2. The IL representation of secret_write, after being compiled by GCC.

The specification file provides the information needed to derive the proof obligations: global variables, the security classification of global variables L, rely and guarantee conditions *Rely* and *Guarantee*, and the function specifications, in this case just of *main*. Note $n$bv32 denotes a 32-bit bitvector representation of the literal n. If any of the information is omitted, default values are used instead: for function specifications' `requires` and `ensures` clauses are assumed *true*; for $\mathcal{R}$ the environment is assumed to not modify any shared variables and $\mathcal{G}$ defaults to no constraints. The security classification of global variables defaults to low to render the information flow analysis sound.

The IL representation for part of secret_write, obtained by lifting the binary compiled by GCC, is shown in Fig. 2. All instructions apart from the initial `z = z + 1` are elided. The registers `R0` and `R1` refer to the 64-bit AArch64 registers `X0` and `X1`, which are encoded as global variables storing 64-bit bitvectors. Line 4 sets `R0` to the base address of the global memory, from which the code finds the address of `z` at offset `20` (added to the base address). Line 6 adds `1` to the content of `R0` and extends the result to a 64-bit value before assigning it to `R1`. At Line 8 the incremented value in `R1` is stored back to `z`.

```
1  procedure main()
2   modifies Gamma_R0,Gamma_R1,Gamma_mem,R0,R1,mem;
3   requires(gamma_load32(Gamma_mem, $x_addr)==true);
4   ...
5  {
6   var Gamma_x_old: bool; var z_old: bv32;
7   lmain:
8    ...
9    call rely();
10   assert (L(mem, R0) ==> Gamma_R1);
11   z_old := memory_load32_le(mem, $z_addr);
12   Gamma_x_old := (gamma_load32(Gamma_mem, $x_addr)
13                  L(mem, $x_addr));
14   mem, Gamma_mem := memory_store32_le(mem, R0, R1[32:0]),
15                     gamma_store32(Gamma_mem, R0, Gamma_R1);
16   assert ((R0 == $z_addr) ==> (L(mem, $x_addr) ==> Gamma_x));
17   assert bvsge32(memory_load32_le(mem,$z_addr),z_old);
18   ...
19 }
```

Fig. 3. `main` procedure for secret_write translated to Boogie

For each IL procedure, block, and statement, a corresponding Boogie procedure, block or statement, is generated as defined in Sec. 2.1. Each register assignment is translated to a simultaneous register and Γ update (14). Snippets of the generated Boogie code for secret_write, given in Fig 3, showcase how the Boogie code is instrumented to check for potential information leakage. The encoding is of the wp_{if}^{RG} transformer for the first occurrence of `z = z + 1`.

To support readability of the code we generate for each global variable x a declaration `const $`x`_addr:bv64` and an axiom (`axiom ($`x`_addr ==` n`bv64)`) where n is the address of the global variable extracted from the binary's symbol table via the `readelf` command.

Information flow security is verified through the assertions which correspond to the proof obligations po_1^B and po_2^B of the store, as defined in (16). Line 10 checks the security value of `R1` is low if the security classification at the store address (previously stored in `R0`) is low. Line 16 checks that if the store was to `z`, the only variable in `L` which is used to control `x`, then the security classification of `x` after the update still needs to imply the security value of `x`, `Gamma_x`.

In a concurrent setting, the specified functionality and security constraints have to be maintained by the rely condition, which is encoded as a call to `rely` in Line 9. Additionally, the program guarantees to other threads that it only changes `z` by increasing it, which is checked in Line 17.

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Ernst, M.D., Jensen, T.P. (eds.) Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05. pp. 82–87. ACM (2005). https://doi.org/10.1145/1108792.1108813
2. Coughlin, N., Lam, K., Winter, K.: Weak memory rely/guarantee logic (Jan 2024), https://github.com/UQ-PAC/wmm-rg/tree/capture-sec
3. Coughlin, N., Winter, K., Smith, G.: Rely/guarantee reasoning for multicopy atomic weak memory models. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) Formal Methods - 24th International Symposium, FM 2021. Lecture Notes in Computer Science, vol. 13047, pp. 292–310. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_16
4. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer-Verlag, Berlin, Heidelberg (1990)
5. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577
6. Murray, T.C., Sison, R., Engelhardt, K.: Covern: A logic for compositional verification of information flow control. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018. pp. 16–30. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00010
7. Smith, G.: A Dafny-based approach to thread-local information flow analysis. In: 11th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE 2023. pp. 86–96. IEEE (2023). https://doi.org/10.1109/FormaliSE58978.2023.00017
8. Winter, K., Coughlin, N., Smith, G.: Backwards-directed information flow analysis for concurrent programs. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00017
9. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects of Computing **9**(2), 149–174 (1997). https://doi.org/10.1007/BF01211617