Appendices for "Verified Secure Compilation for Mixed-Sensitivity Concurrent Programs"

Robert Sison[*] [†] and Toby Murray[*]

[*]School of Computing and Information Systems, University of Melbourne, Australia
[†]CSIRO's Data61 and UNSW Sydney, Australia
(e-mail: [firstname].[lastname]@unimelb.edu.au)

A Label allocation and sequential composability

The wr-compiler fixes the label type $Lab \triangleq nat$ to allow it to ensure freshness merely by using the highest natural number reached so far on a "next label" counter (the argument nl); it then increments the counter before passing it to subsequent calls, and outputs the next available unused label on return (the return value nl').

Relative to this scheme, we prove that two *consecutively compiled* `RISC` programs—in the sense that the relevant outputs from the first call are fed directly into the second call—only ever jump to locations within themselves (and not in the other).

Specifically, we define two `RISC` programs P_1, P_2 to be joinable if they are both:

- joinable-forward: P_1 only ever jumps to labels that are either
  - labelling an instruction in P_1 itself, or
  - the label of the very first instruction in P_2.
- joinable-backward: P_2 does not jump to any of the labels of instructions in P_1.

The lemma we prove then says that two `RISC` programs output by consecutive invocations of the wr-compiler are joinable.

Proving that the control flow of programs compiled by the wr-compiler always remains self-contained in this manner facilitates reasoning about their sequential composition.

B Register allocation scheme model

Like Tedesco *et al.* (2016) we generalise over the (user-supplied) register allocation scheme, and assume there are enough registers to service the maximum depth of expressions in the source program. We leave for future work the modelling and analysis of a compiler phase that spills register contents to memory, in order to make this assumption unnecessary.

Here we model the (user-supplied) register allocation scheme with two functions *reg_alloc* and *reg_alloc_cached* on the *register record* Φ and the set A of registers whose contents are needed to evaluate the current expression. To avoid loading from memory unnecessarily, the compiler may first call *reg_alloc_cached* $\Phi\ A\ v$ to identify a register that Φ records as already containing the variable v. When the compiler needs a fresh register, it will call *reg_alloc* $\Phi\ A$. Neither function is allowed to allocate a register in A, so the allocator is permitted to fail if it cannot find any suitable register. However, registers typically become available again as expression evaluation is resolved.

C Informal descriptions of cases of refinement relation $\mathscr{R}_{\mathsf{wr}}$

C.1 Base cases

- `stop`: This case relates a terminated `While` program with a terminated `RISC` program (i.e. one where the program counter is at the length of the program text).
- `skip_nop`: This case relates the `While` program **skip** with the configuration where the program counter is at the start of the `RISC` program [**Nop**].
- `assign_expr`: This case relates the expression evaluation part (for the expression e) of the `While` program $v := e$ with the corresponding part of the `RISC` program obtained by compiling it with the wr-compiler.
- `assign_store`: As for `assign_expr`, but for the very last **Store** instruction that commits the result of the expression evaluation back to shared memory variable v.

  It asserts additionally that v must be stable if lock-governed, and non-lock-governed otherwise. This prevents threads from violating the locking discipline.
- `lock_acq`: This case relates **lock**(k) with **LockAcq** k.
- `lock_rel`: This case relates **unlock**(k) with **LockRel** k.

C.2 Inductive cases

- `seq`: This case relates the `While` program $c_1 ; c_2$ with the *concatenation* $P_1 @ P_2$ of the `RISC` programs P_1 and P_2 that are respectively the outputs of successful consecutive compilation (see Appendix A) of c_1 and c_2 by the wr-compiler. It is intended for cases where the `While` (resp. `RISC`) program is currently in c_1 (resp. P_1).

  It is an inductive case of $\mathscr{R}_{\mathsf{wr}}$, in that:

  - c_1 is required to be related by $\mathscr{R}_{\mathsf{wr}}$ to the present location in P_1.
  - For all local configurations that obey the compiled-cmd-config-consistent requirements, c_2 is required to be related by $\mathscr{R}_{\mathsf{wr}}$ to the first instruction of P_2. This quantification ensures that $\mathscr{R}_{\mathsf{wr}}$ remains closed when execution progresses from the first program to the second program.

  It asserts that P_1 and P_2 are joinable (Appendix A), which is particularly relevant here to ensure that P_1 can only jump to locations within or at the end of itself (i.e. the start of P_2).

- `join`: This case relates a `While` program c with an offset $pc > \mathsf{length}\ P_1$ into a `RISC` program $P_1 @ P_2$, assuming the inductive hypothesis that c is related by $\mathscr{R}_{\mathsf{wr}}$ with the offset $pc - \mathsf{length}\ P_1$ into the `RISC` program P_2 alone.

  It is intended primarily for cases where the `While` (resp. `RISC`) program is currently in the c_2 (resp. P_2) of some consecutively compiled $c_1 ; c_2$ (resp. P_1 concatenated with P_2) but applies more broadly to allow any prepend of dead, unreachable instructions onto the front of a `RISC` program without breaking $\mathscr{R}_{\mathsf{wr}}$.

  It also asserts that P_1 and P_2 are joinable, which is important here to ensure that P_2 cannot jump back into P_1.

- `if_expr`: This case relates the expression evaluation part (for the expression e) of the `While` program **if** e **then** c_1 **else** c_2 **fi** with the corresponding part (including the conditional jump **Jz** at the end of expression evaluation) of the `RISC` program obtained by compiling it with the wr-compiler.

  It relies on both c_1 and c_2 being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled `RISC` counterparts when started with initialisation states judged valid by compiled-cmd-config-consistent.

  This case is depicted in full in Figure C.1, on page 4; for comparison, Figure C.2 depicts the relevant part of the compile-cmd implementation.

- `if_c1`: This case relates some `While` program c_1' reachable from c_1 with the corresponding part within the c_1 part of the `RISC` program obtained by compiling **if** e **then** c_1 **else** c_2 **fi** with the wr-compiler.

  It relies on c_1 being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled `RISC` counterpart at the appropriate program counter offset.

- `if_c2`: As for `if_c1`, but for c_2.

- `epilogue_step`: This case relates a terminated `While` program to the silent control flow steps navigating to the end of a `RISC` program from the end of the “then” and “else” branches of a compiled if-conditional.

  It works only for the “epilogue” step forms: **Jmp** and **Nop**.

  It is inductive in that it asserts closedness of $\mathscr{R}_{\mathsf{wr}}$ over pairwise reachability from the pair currently under consideration—the only case to do so directly.

- `while_expr`: This case relates the `While` program (**while** e **do** c **od**)’s initial intermediate step to **if** e **then** (c ; **while** e **do** c **od**) **else stop fi**, and its expression evaluation part, with the expression evaluation and conditional jump of the `RISC` program that **while** e **do** c **od** was compiled to by compile-cmd.

  It relies on c being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled `RISC` counterpart when started with initialisation states judged valid by compiled-cmd-config-consistent.

- `while_inner`: This case relates some program c_I ; **while** e **do** c **od** reachable from c ; **while** e **do** c **od** to the loop body part of the `RISC` program compiled from **while** e **do** c **od**.

It relies on c_I being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled RISC counterpart at the appropriate program counter offset.

It also carries around the same reliance on c being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled RISC counterpart for all initialisation states judged valid by compiled-cmd-config-consistent.

- while_loop: This case handles epilogue steps for the inner loop body program, and the final jump back to the beginning of the While-loop.

  It requires $\mathscr{R}_{\mathsf{wr}}$ to relate the terminated While program to the end of the compiled loop body, and furthermore also carries around the same reliance on c being related by $\mathscr{R}_{\mathsf{wr}}$ to its compiled RISC counterpart for all initialisation states judged valid by compiled-cmd-config-consistent.

$$\frac{\begin{array}{c} c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} \qquad \mathsf{compile\text{-}cmd\text{-}input\text{-}reqs}\ C\ l\ nl\ c \\ (PCs, l', nl_2, C', \mathsf{False}) = \mathsf{compile\text{-}cmd}\ C\ l\ nl\ c \qquad (P_e, r, C_1, \mathsf{False}) = \mathsf{compile\text{-}expr}\ C\ \varnothing\ l\ e \\ (P_1, l_1, nl_1, C_2, \mathsf{False}) = \mathsf{compile\text{-}cmd}\ C_1\ \mathsf{None}\ (\mathsf{Suc}\ (\mathsf{Suc}\ nl))\ c_1 \qquad pc \le \mathsf{length}\ P_e \\ (P_2, l_2, nl_2, C_3, \mathsf{False}) = \mathsf{compile\text{-}cmd}\ C_1\ (\mathsf{Some}\ nl)\ nl_1\ c_2 \qquad C_{pc} = (\mathsf{map\ snd}\ PCs)[pc] \\ \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C_{pc}\ regs\ mds\ mem \qquad \mathsf{regrec\text{-}stable}\ C_{pc} \\ \forall mds'\ mem'\ regs'.\ \mathsf{compiled\text{-}cmd\text{-}config\text{-}consistent}\ C_1\ regs'\ mds'\ mem' \wedge \mathsf{regrec\text{-}stable}\ C_1 \\ \longrightarrow ((\langle c_1, mds', mem'\rangle_{\mathsf{w}}, \langle(((0, \mathsf{map\ fst}\ P_1), regs'), mds', mem'\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}} \wedge \\ (\langle c_2, mds', mem'\rangle_{\mathsf{w}}, \langle(((0, \mathsf{map\ fst}\ P_2), regs'), mds', mem'\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}}) \end{array}}{(\langle c, mds, mem\rangle_{\mathsf{w}}, \langle(((pc, \mathsf{map\ fst}\ PCs), regs), mds, mem\rangle_{\mathsf{r}}) \in \mathscr{R}_{\mathsf{wr}}}$$

Figure C.1: Introduction rule for case if_expr of refinement relation $\mathscr{R}_{\mathsf{wr}}$.
This case pertains to the expression-evaluation part of an **if**-conditional compiled by compile-cmd (see Figure C.2). Variables ignored are in gray.

```
compile_cmd C l nl (If e c₁ c₂) =
  (let (Pₑ, r, C₁, failₑ) = (compile_expr C {} l e);
      (br, nl') = (nl, Suc nl); (ex, nl'') = (nl', Suc nl');
      (P₁, l₁, nl₁, C₂, fail₁) = (compile_cmd C₁ None nl'' c₁);
      (P₂, l₂, nl₂, C₃, fail₂) = (compile_cmd C₁ (Some br) nl₁ c₂);
      (∗ Pre−compilation check ensures asmrec C₂ = asmrec C₃ ∗)
      C' = (regrec C₂ ⊓_R regrec C₃, asmrec C₂)
   in (Pₑ @ [((if Pₑ = [] then l else None, Jz br r), C₁)] @
      P₁ @ [((l₁, Jmp ex), C₂)] @ P₂ @ [((l₂, Nop'), C₃)],
      Some ex, nl₂, C', failₑ ∨ fail₁ ∨ fail₂))
```

Figure C.2: Excerpt of wr-compiler implementation: case for **if**-conditionals.
This case of the Isabelle/HOL function compile-cmd compiles the While command **if** e **then** c_1 **else** c_2 **fi**. Here, @ denotes concatenation between two RISC program texts, and $\Phi \sqcap_R \Phi'$ denotes the subset of mappings on which the register records Φ and Φ' agree.

References

Tedesco, F. D., Sands, D. and Russo, A. (2016) Fault-resilient non-interference. *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016* pp. 401–416. IEEE Computer Society.