CompCertOC: Verified Compositional Compilation of Multi-threaded Programs with Shared Stacks

LING ZHANG, Shanghai Jiao Tong University, China
YUTING WANG*, Shanghai Jiao Tong University, China
YALUN LIANG, Shanghai Jiao Tong University, China
ZHONG SHAO, Yale University, USA

It is a long-standing open problem to support verified compilation of multi-threaded programs *compositionally* when sharing of stack data between threads is allowed. Although certain solutions exist on paper, none of them is completely formalized because of the difficulty in simultaneously enabling sharing and forbidding modification of stack memory in presence of arbitrary memory operations (e.g., pointer arithmetic). We present a compiler verification framework that solves this open problem in the setting of *cooperative multi-threading*. To address the challenges of sharing stack data, we introduce *threaded Kripke memory relations* (TKMR) to support both protection and sharing of stacks in a multi-stack memory model. We further introduce *threaded forward simulations* parameterized by TKMR to capture semantics preservation for compiling program modules in multi-threaded contexts. We show that threaded forward simulations are both *horizontally composable*—thereby enabling the compositional verification of open threads and heterogeneous modules—and *vertically composable*—thereby enabling composition of compiler correctness for multiple compiler passes. Furthermore, threaded forward simulations can be converted into backward simulations. We apply this framework to 18 passes of CompCert to get CompCertOC, the first optimizing verified compiler that supports compositional verification of cooperative multi-threaded programs with shared stacks.

CCS Concepts: • **Theory of computation → Program verification**; **Concurrency**; • **Software and its engineering → Correctness**; **Formal software verification**; **Compilers**.

Additional Key Words and Phrases: Verified Compositional Compilation, CompCert, Cooperative Multithreading, Stack Sharing

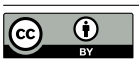
ACM Reference Format:
Ling Zhang, Yuting Wang, Yalun Liang, and Zhong Shao. 2025. CompCertOC: Verified Compositional Compilation of Multi-threaded Programs with Shared Stacks. *Proc. ACM Program. Lang.* 9, PLDI, Article 173 (June 2025), 27 pages. https://doi.org/10.1145/3729276

1 Introduction

Software often consists of modules written in different languages (i.e., heterogeneous modules). To ensure the verified properties hold for their compiled binary code, it is essential to support *verified compositional compilation* (VCC), i.e., separate verification of the compilers for heterogeneous modules and composition of compiler correctness theorems. The past decade has witnessed great

*Yuting Wang is the corresponding author.

Authors' Contact Information: Ling Zhang, John Hopcroft Center for Computer Science, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China, ling.zhang@sjtu.edu.cn; Yuting Wang, John Hopcroft Center for Computer Science, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China, yuting.wang@sjtu.edu.cn; Yalun Liang, Zhiyuan College, Shanghai Jiao Tong University, Shanghai, China, liangyalun@sjtu.edu.cn; Zhong Shao, Department of Computer Science, Yale University, New Haven, USA, zhong.shao@yale.edu.



ACM 2475-1421/2025/6-ART173
https://doi.org/10.1145/3729276

```
1 void *thread(void *p) { (*(int *)p)++; }
2 int main() {  int i = 0; pthread_t tid;
3               pthread_create(&tid,NULL,thread,(void *)&i);
4               pthread_join(tid,NULL); printf("%d\n", i);  }
```

Fig. 1. A Concurrent Program not Supported by VCC

progress towards achieving this goal [Gu et al. 2015; Koenig and Shao 2021; Song et al. 2020; Stewart et al. 2015; Wang et al. 2019; Zhang et al. 2024]. These projects target realistic optimizing compilers and are built upon CompCert, the state-of-the-art verified compiler [Leroy 2023].

It has been long believed that VCC can support concurrent programs. For *cooperative multitasking*, when a single concurrent thread (or process) is under focus, it behaves like an open module interacting with other threads through external function calls because context switches behave like function calls in this case. Therefore, compiler correctness for individual threads and processes can be proved separately and composed into that for the complete program with cooperative concurrent semantics. To apply such compiler correctness to concurrent program verification with preemptive semantics, the equivalence between cooperative and preemptive semantics is then proved by exploiting the well-behavedness of source and target programs. For example, in CASCompCert [Jiang et al. 2019a], this is done by exploiting a DRF-SC (i.e., data-race free programs are sequentially consistent) theorem to shuffle context switches to call sites to thread primitives. Since this proof is more about program verification rather than compiler verification, we shall focus on cooperative concurrency in the remaining discussion.

Although the existing works have realized the above vision to a certain extent [Gu et al. 2018; Jiang et al. 2019a; Zha et al. 2022], they are unable to support a common practice in multi-threaded programming, i.e., sharing of stack-allocated data among threads. [1] A simple example for illustrating this problem is depicted in Fig. 1. It is representative of concurrent programs that delegate tasks to child threads by sharing stack data. The main thread creates a child thread for increasing its stack variable `i` by passing its address, which effectively results in sharing of `i`. After the task is completed, the main thread prints the result. In essence, the two threads behave like two modules sharing the same code but with different entry points (`main` and `thread`). More importantly, their executions are interleaved because of concurrency. A context switch starting from one thread and ending with switching back behaves like an external call of that thread. Therefore, compiler correctness for the two threads should simply be those for modules focusing on different entry functions, which are then composed to form the correctness for compiling the whole program.

Unfortunately, none of the existing approaches to VCC supports this seemingly trivial example, because there are fundamental conflicts between supporting shared stack-data and the critical compositionality properties for compiler correctness, i.e., *horizontal and vertical compositionality*:

$$\textbf{Vertical Compositionality:} \quad L_1 \leqslant L_2 \Rightarrow L_2 \leqslant L_3 \Rightarrow L_1 \leqslant L_3$$

$$\textbf{Horizontal Compositionality:} \quad L_1 \leqslant L_1' \Rightarrow L_2 \leqslant L_2' \Rightarrow L_1 \oplus L_2 \leqslant L_1' \oplus L_2'$$

Here, a *refinement* relation $L_1 \leqslant L_2$ (usually a *forward simulation*) encodes the compiler correctness between cooperative semantics of open modules before (L_1) and after (L_2) compilation. Vertical compositionality ensures that the refinements for two adjacent compiler passes can be combined into a single refinement. It is critical for supporting multi-pass compilers. Horizontal compositionality ensures that the parallel refinements for different open modules can be combined together where $L_1 \oplus L_2$ denotes the semantic composition of L_1 and L_2. It is critical for compiling heterogeneous

[1] A solution exists on paper [Jiang et al. 2019b] which is not formalized due to excessive complexity. See §7 for details.

modules where L_1 and L_2 may denote semantics of modules written in different languages with complicated interactions (e.g., mutual calls). Both compositionalities require appropriate description of the *rely conditions* the open modules depend upon and the *guarantee conditions* they ensure. A great amount of work has been devoted to ensuring both compositionalities with or without concurrency [Gu et al. 2015; Hur et al. 2012; Koenig and Shao 2021; Neis et al. 2015; Patterson and Ahmed 2019; Song et al. 2020; Stewart et al. 2015; Zhang et al. 2024].

Continue with the example in Fig. 1. The child thread should be able to modify the publicly accessible stack variable `i` of its parent. On the other hand, it should not modify any private stack variable of its parent (e.g., `tid`). The requirement for sharing public stack data while protecting private stack data imposes additional rely-guarantee conditions. Because it is unclear how to formalize these rely-guarantee conditions so that they are preserved under horizontal and vertical composition, existing works on VCC for concurrency (among them the most well-known are Thread-safe CompCertX [Gu et al. 2018] and CASCompCert [Jiang et al. 2019a]) simply forbid stack memory of one thread (`i` for the main thread in Fig. 1) to be leaked to and modified by another thread. This solution deviates from the common assumption that stack memory should be treated no differently from other memory such as heap or global memory (e.g., POSIX thread APIs), and cannot support common programming idioms like in Fig. 1 or *scoped threads* in Rust [Library 2024].

In this paper, we present a formalized solution to the open problem of sharing stack-data in VCC for *cooperative multi-threaded* programs. Our contributions are summarized as follows:

- We introduce *threaded Kripke memory relations* (TKMR), a formalization of rely-guarantee conditions for multi-threaded programs that simultaneously supports sharing of public stack data and protection of private data. On top of that, we introduce *threaded forward simulations*, a notion of compiler correctness as forward simulations for *cooperative concurrent semantics* that supports shared stacks, open threads written in heterogeneous modules, and both horizontal and vertical compositionality. Moreover, they can be flipped into backward simulations in the end by following the same idea of CompCert.
- We verify that 18 of CompCert's total 20 passes satisfy threaded forward simulations[2]. By vertical compositionality we get CompCertOC, the first verified optimizing compiler that supports multi-threaded open programs with stack sharing. The formal development is based on two existing extensions of CompCert: Nominal CompCert [Wang et al. 2022] for supporting thread-local stacks and CompCertO [Koenig and Shao 2021; Zhang et al. 2024] for supporting VCC. It is fully formalized in Coq based on CompCert v3.13.
- To demonstrate the effectiveness of the above framework, we apply it to verify non-trivial multi-threaded programs that combine heterogeneity (modules written in different languages such as C and assembly) and stack sharing (sharing stack data between parent and child threads using POSIX thread APIs). For this, we develop semantics for describing programs using POSIX thread APIs and the techniques for composing compiler correctness to derive both forward and backward simulations between complete multi-threaded programs.

Note that our compiler correctness theorems include both forward and backward simulation of cooperative semantics. As mentioned above, following the existing approaches [Gu et al. 2018; Jiang et al. 2019a], the connection with preemptive semantics should be handled in program verification frameworks and left for future work (see the discussion in §7 for more details). In the rest of the paper, we first introduce the key ideas for enabling stack sharing in VCC in §2. We then discuss the technical background and challenges in §3. We elaborate on our ideas and contributions in §4, §5 and §6. Finally, we discuss the related work in §7.

[2]The two missing passes are `SimplExpr` and `Unusedglob`. We explain why they are omitted in §5.

```c
/* client.c */
#define N 5
typedef struct {
  int *input, *result, size; } Arg;
void* server(void *a);

int main() {
  pthread_t a;
  int input[N]={1,2,3,4,5}, result[N];
  int mask = 0, i;
  Arg arg = {input,result,N};

  pthread_create(&a,0,server,&arg);
  for (i = 0;i < N;i++)
    { mask += input[i]; yield(); }
  pthread_join(a, NULL);
  for (i = 0;i < N;i++) {
    result[i] = result[i] & mask;
    printf("%d; ", result[i]);  }
}
```

(a) Client running on the main thread

```c
/* server.c */
void encrypt (int i, int *r);
void* server(void *a) {
  int *i = ((Arg *)a)->input;
  int *r = ((Arg *)a)->result;
  int size = ((Arg *)a)->size;
  for (int j = 0;j < size;j++) {
    encrypt(i[j], r+j);
    yield();  }
  return NULL;
}
/* encrypt.s */
key: .long 42
encrypt:
Pallocframe 16 8 0 // allocate frame
Pmov key RAX
Pxor RAX RDI   // result = i XOR key
Pmov RDI (RSI) // stores to (RSI)
Pfreeframe 16 8 0 // free frame
Pret
```

(b) Server running on the child thread

Fig. 2. An Example of Multi-Threaded Program with Stack Sharing

2 Key Ideas

2.1 A Running Example

To illustrate the key ideas, we introduce a running example of multi-threaded program composed of open modules in Fig. 2. It consists of a client written in C (Fig. 2a) which runs on the main thread and a server written in C and assembly (Fig. 2b) which runs on the child thread. We use POSIX primitives `pthread_create` and `pthread_join` to implement thread creation and waiting. As we work with cooperative concurrency, we use a `yield` primitive to encode explicit context switches.

This example encrypts `N` input integers, where the encryption happens both at the server side (XORing inputs with a fixed encryption key 42) and at the client side (encoding the results of the server with a mask calculated from the inputs). The `main` function uses variables `input` and `result` as its buffers. The addresses and sizes of these buffers are passed to `server` through `pthread_create`. Concurrent to the execution of `server`, the client calculates a mask from the input values which is used to encode the results after the `pthread_join`. The function `encrypt` is written in assembly. It receives input value i and the pointer to result r in `RDI` and `RSI` respectively, does the encryption using XOR and stores the result. The function `server` simply calls the encryption function written in assembly for each input value. We assume that context switches happen at the end of each iteration by calling `yield`. As we can see, although the `input` and `result` arrays reside in the client stack, they are shared and modified by the server thread.

Despite being small, our running example contains all the features we care about in this work, including heterogeneous modules, dynamic thread creation, stack sharing, and context switches between threads. Therefore, it is non-trivial and none of the existing approaches can support its VCC. We shall use this example to illustrate the *feasibility* of our approach in VCC of concurrency with stack-sharing. We leave the application to larger-scale program verification for future work.

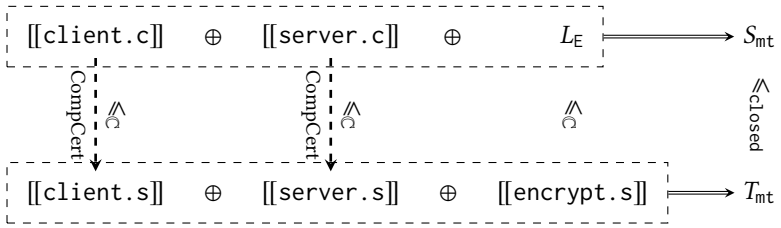


Fig. 3. Verification of the Running Example

2.2 Motivations

Following the standard practice of VCC, the schema for verifying the running example is depicted in Fig. 3. We first verify the correctness for compiling individual modules which is denoted by a refinement relation $\leqslant_{\mathbb{C}}$ between source and target semantics ($[\![M]\!]$ denotes the semantics of module M). We then compose individual refinements into $\leqslant_{\texttt{closed}}$ for the whole program. In the first step, we treat all thread primitives such as `yield`, `thread_create` and `thread_join` as external calls, which are turned into internal executions like other external calls in the second step.

Unfortunately, existing techniques for VCC of sequential programs do not work for thread primitives. To see that, we first review the most recent approach to VCC developed by Zhang et al. [2024]. In this approach, semantics of open modules are implemented as open *labeled transition systems*. Refinements between them are defined as forward simulations (which may be flipped into backward simulations) using rely-guarantee conditions which have been widely studied [Koenig and Shao 2021; Liang et al. 2012; Song et al. 2020; Stewart et al. 2015; Zhang et al. 2024].

Fig. 4 depicts a simulation between two open semantics L_1 and L_2. The source semantics L_1 is invoked by a query (i.e., function call) q_1 and may issue external call q_1' after internal execution. When the external function returns with a reply r_1', L_1 continues to execute. After several external calls, it finally returns a reply r_1 to the initial query. The target semantics L_2 shares the same structure. Its queries and replies are related to the source via simulation invariants denoted by the vertical double arrows. Certain *rely conditions* for external calls are assumed so that the invariant between r_1' and r_2' still holds when they return. In turn, *guarantee conditions* from q_1 and q_2 to r_1 and r_2 need to be proved so that L_1 and L_2 satisfy the rely conditions for their callers. It is exactly those rely-guarantee conditions that enable the composition of refinements in Fig. 3.

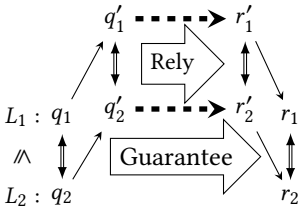
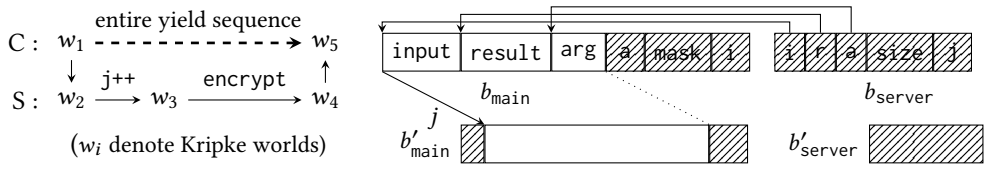


Fig. 4. Rely-Guarantee Simulation

The most important job of the rely-guarantee conditions is to distinguish between *private* memory local to modules or threads and *public* memory shared between them, and to protect the former from modification by the environment. For example, the rely condition for calling `encrypt` in `[[server.c]]` requires that the local variable `j` is unchanged during the call because `j` could be optimized into a callee-saved register. The simulation no longer holds if `j` is changed by `encrypt`.

For this, *Kripke memory relations*, or KMRs are proposed by CompCertO [Koenig and Shao 2021]. In a KMR, a world type W is given s.t. each world $w = (j, m_1, m_2) \in W$ contains a pair of source and target memory states m_1 and m_2 related by a memory injection function j (a structure-preserving embedding of source memory states into target [Leroy et al. 2012]). The evolution of memory across external calls is then defined as the *accessibility* relation between worlds (written as $w \rightsquigarrow w'$) which enforces necessary memory protection. The most general KMR is called `injp`. Its key idea is to



(a) A call to yield (b) Memory injection at the beginning of yield

Fig. 6. An Example Showing Inadequacy of `injp` for Multithreading

define the *private* and *public* regions for the source memory m_1 and the target m_2 w.r.t. the domain and range of injection j for any given world (j, m_1, m_2). For example, in Fig. 5, m_1 and m_2 contain memory regions to the left of the dashed vertical line. They are related by j before an external call. The regions in the domain and range of j (white areas) are *public* because they represent memory reachable from source and target function arguments. The regions in m_1 *unmapped* by j (shaded areas in m_1) and those *out-of-reach* from j in m_2 (shaded areas in m_2) are *private* memory of the caller because they are either removed from the source memory or inserted into the target memory to store private data (e.g., spilled registers) by the compiler. During the external call, new memory blocks may be allocated, resulting in updated memories m'_1 and m'_2 and injection j'. The accessibility $(j, m_1, m_2) \rightsquigarrow (j', m'_1, m'_2)$ ensures that the private regions in m_1 and m_2 are unchanged in m'_1 and m'_2, i.e., they cannot be modified by the external call (callee).

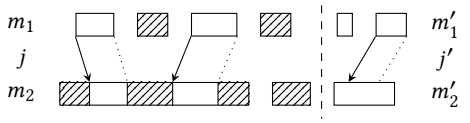


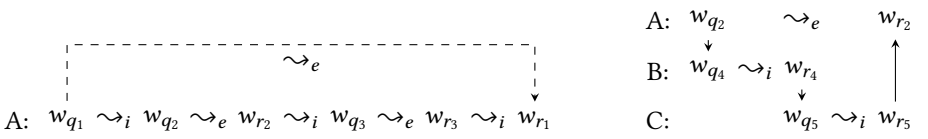
Fig. 5. Kripke Worlds Related by `injp`

Although `injp` works well for sequential programs, it does not correctly describe the rely-guarantee conditions for context switches when they are treated as external calls. For example, assume the execution of our running example has reached the following state: the server thread has previously switched to the client by calling `yield` at line 9 in Fig. 2b, and the client thread is about to switch back to the server at line 15 in Fig. 2a. The subsequent call to `yield` executes `j++`, calls `encrypt` in the server thread, and returns to the client by another `yield`, as shown in Fig. 6a where the vertical arrows stand for context switches.

A snapshot of the memory state right before `yield` is given in Fig. 6b where $b_{\texttt{main}}$ and $b_{\texttt{server}}$ depict the client and server stack frames at the source, respectively; $b'_{\texttt{main}}$ and $b'_{\texttt{server}}$ are target frames after compilation. Note that for simplicity we omit the stack frame for `encrypt`. Because `input`, `result` and `arg` are leaked public memory, they are preserved by compilation and reside in the domain of injection j. By definition, `injp` already enables sharing of those public stack regions across threads. Therefore, `encrypt` is allowed to modify their contents. However, except for `input`, `result` and `arg`, all the remaining memory regions are private: source private variables including `j` are turned into register values and target ones are newly added private stack regions. Therefore, `j++` is illegal by `injp`. Since `yield` as an external call violates the rely condition that private memory cannot be modified, we fail to prove the compiler correctness.

2.3 Key Idea 1: Threaded Kripke Memory Relations

It is easy to observe that calling `yield` is actually not the same as calling a regular function. The yield sequence in Fig. 6a begins and ends in the middle of a function call. Such internal execution of a function call should be allowed to modify the private variables on its own stack frame (e.g.,



(a) Accessibilities for function calls (b) Verifying context switches

Fig. 7. Rely-Guarantee Reasoning for Function Calls and Context Switches

`j`). The problem is that `injp` only captures the rely-guarantee of an entire function call, not its internal steps, which is *too strong* for context switches.

Our solution is to revise KMR and the memory model so that they capture the desired rely-guarantee memory protection for both regular function calls and context switches (caused by `yield`, `thread_create`, `thread_join`, etc.). The enhanced KMR is called *threaded Kripke memory relations* (TKMR). In TKMR, a new accessibility relation called *internal accessibility* (denoted by $\rightsquigarrow_i$) is introduced whose only purpose is to describe the guarantees provided by internal executions. That is, internal steps in the *current thread* can modify *its own private stack*, but never modify the private stack of *the other threads*. For example, the server can perform `j++` internally, but it can never modify `mask`. Now, as a *rely* condition for `yield`, the original accessibility $\rightsquigarrow$ is in conflict with $\rightsquigarrow_i$ as it does not allow modification to *any* private memory. To solve this problem, TKMR relaxes $\rightsquigarrow$ to only protect the private memory of the *current* thread, resulting in *external accessibility* (denoted by $\rightsquigarrow_e$). This relaxation is possible because simulation invariants for the current thread can be preserved by context switches or external function calls if its own private state is not modified by them; modification of the private stacks of the other threads is harmless (e.g., `j++` in Fig. 6a). In summary, the new accessibility relations are defined as follows:

(1) $w \rightsquigarrow_e w'$ holds if the *private* memory of the *current* thread in w is unchanged in w';
(2) $w \rightsquigarrow_i w'$ holds if the *private* memory of the *other* threads in w is unchanged in w'.

To formalize the above definitions, we need to distinguish stack memory blocks allocated for different threads. This is difficult in the original CompCert where memory blocks are assigned positive numbers sequentially as their identifiers. We adopt a Nominal Memory Model [Wang et al. 2022] to divide the memory space into global memory and individual stack memory for each thread. Each stack block is named additionally using the thread id of its owner. The memory state also includes a current thread id to distinguish thread local blocks from the other stack blocks.

With the devices of TKMR in place, we first show what the rely-guarantee conditions for regular function calls look like. An example of memory evolution is depicted in Fig. 7a, which describes a function call by thread A. w_{q_1} represents the world from initial calls (e.g., q_1 and q_2 in Fig. 4). There are two external calls q_2 and q_3 during the function execution. Assume q_2 is a `yield` and q_3 is a regular function call (e.g., `encrypt`). We use $\rightsquigarrow_e$ to uniformly describe their *rely* conditions, fulfilling our purpose to treat context switches as regular calls. $\rightsquigarrow_i$ captures the *internal guarantee* which defines legal memory evolution of internal execution starting from either the initial call (q_1) to an external call (q_2), or a function reply (r_2) to another external call (q_3), or from a function reply (r_3) to the final return (r_1). Finally, the *external guarantee* $w_{q_1} \rightsquigarrow_e w_{r_1}$ for the entire function call is proved by reasoning about its behavior like in CompCertO.

We then illustrate how rely conditions for context switches are verified using internal accessibility. Fig. 7b shows a possible execution sequence for the yield from q_2 to r_2 in Fig. 7a. Context switches between different threads are denoted by vertical arrows. For example, w_{q_2} and w_{q_4} contain the same memory states except that the current thread is changed from A to B. Between the initial call

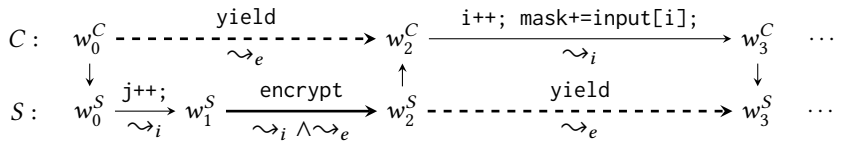


Fig. 9. TKMR Accessibility Relations for the Running Example

to yield by thread A and the return to A, the rely condition $\rightsquigarrow_e$ requires the private and local stack memory of thread A unchanged. This condition is satisfied by the internal guarantee of thread B and C as follows. By definition, $w_{q_4} \rightsquigarrow_i w_{r_4}$ ensures the private stacks of threads A and C are not modified by B. Similarly, $w_{q_5} \rightsquigarrow_i w_{r_5}$ ensures the private stacks of threads A and B are not modified by C. Therefore, the private stack of A is not modified throughout the entire yield sequence. Note that we assume context switches will not further trigger external calls. This is ensured by first performing linking of modules and then linking of threads, as we shall discuss below.

2.4 Key Idea 2: Threaded Forward Simulations

To formalize the correctness for compiling individual threads where thread primitives are treated as external functions, we introduce an enhancement to the rely-guarantee simulation in Fig. 4 called *threaded forward simulations*. As depicted in Fig. 8, it is a forward simulation of *cooperative concurrent semantics* that relates the source semantics L_1 and target semantics L_2 through a TKMR. Its rely condition embeds external accessibility $\rightsquigarrow_e$ to uniformly describe the requirement for sharing and protecting stack memory for both function calls and context switches. The internal guarantee (I.G.) embeds internal accessibility $\rightsquigarrow_i$ for memory protection by internal steps. Finally, the external guarantee embeds $\rightsquigarrow_e$ to capture the guarantee for regular function calls.

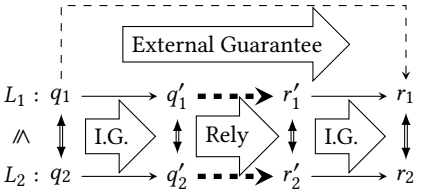


Fig. 8. Threaded Forward Simulation

Threaded forward simulations satisfy the vertical compositionality as described in the introduction. It means that the correctness of individual compiler passes can be composed into a single threaded forward simulation (e.g., $\leqslant_{\mathbb{C}}$ in Fig. 3). Moreover, threaded forward simulations are also horizontally composable, thereby enabling the compiler correctness of individual threads or modules to be composed into correctness for compiling whole programs. Technically speaking, this composition has two steps: *module linking* and *thread linking*. The first step links regular function calls, resulting in refinements for source and target semantics whose only external interaction is calling thread primitives. The second step links thread primitives: by exploiting the internal guarantee and external rely conditions of TKMR, a forward simulation for *closed multi-threaded semantics* which concretely defines program loading and thread primitives is generated. Threaded forward simulations can also be flipped into backward simulations and composed into closed backward simulation by exploiting the *determinacy* of target semantics and *receptiveness* of source semantics like in CompCert.

2.5 Verification of the Running Example

We illustrate how to verify our running example with the above key ideas by following the pattern in Fig. 3. First, we need a verified compiler for compiling `client.c` and `server.c` into assembly code. For this, we have developed CompCertOC which consists of 18 passes of CompCert's 20

verified passes. We prove each pass correct w.r.t a TKMR. By vertical compositionality, we get CompCertOC's compiler correctness as the threaded forward simulation $\leqslant_{\mathbb{C}}$. Second, we verify that the assembly module `encrypt.s` is related to a specification L_{E} through the same $\leqslant_{\mathbb{C}}$. Finally, we perform module linking to link the three modules into a complete program, and thread linking to give closed semantics to `pthread_create`, `pthread_join` and `yield` and to derive the closed forward simulation $\leqslant_{\texttt{closed}}$. A closed backward simulation $\geqslant_{\texttt{closed}}$ can also be obtained by first flipping $\leqslant_{\mathbb{C}}$ and then performing horizontal composition.

The most critical part of the verification is to prove that each open module running on a single thread satisfies the rely-guarantee conditions of TKMR (i.e., internal and external accessibilities). To see how it works, let us revisit our failed example of proving compilation of yield at the end of §2.2. A complete loop of context switches between the two threads is depicted in Fig. 9. The execution follows the solid arrows where the vertical ones stand for context switches and horizontal ones stand for internal execution. The private and public memory are determined by injection as depicted in Fig. 6b. Now, consider proving the correctness for compiling `server.c` which treats `encrypt` as an external function. We first need to prove $w_0^S \rightsquigarrow_i w_1^S$ holds for internal execution, i.e., `j++` does not modify any private variables `a`, `mask` and `i` in $b_{\texttt{main}}$ of the main thread. This is obviously true. Similarly, we can prove that `encrypt` also satisfies the internal accessibility $w_1^S \rightsquigarrow_i w_2^S$. By linking with the module `encrypt.s`, we expose this internal accessibility. Then, we derive the external accessibility $w_0^C \rightsquigarrow_e w_2^C$ for the entire yield sequence from $w_0^S \rightsquigarrow_i w_1^S$ and $w_1^S \rightsquigarrow_i w_2^S$ because they guarantee the internal steps of the server do not modify any private stack variable of the client. Note that $w_0^C \rightsquigarrow_e w_2^C$ holds because, unlike the original accessibility, $\rightsquigarrow_e$ does not require `j` to be unchanged. Symmetrically, $w_2^S \rightsquigarrow_e w_3^S$ is satisfied by the internal guarantees of the main thread.

The changes from KMR to TKMR may look simple at first. However, their formal development is quite challenging and requires substantial efforts, which we elaborate in the following sections.

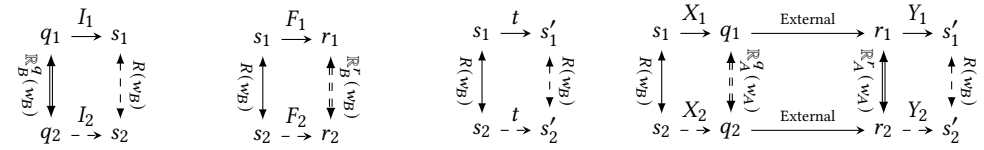
3 Background and Challenges

3.1 Background

3.1.1 Memory Models. The block-based memory model of CompCert [Leroy et al. 2012] defines a memory state m (of type `mem`) as a disjoint set of *memory blocks*. A pointer (b, o) points to the o-th byte from block b where b has type `block` and $o \in Z$ is an integer. Values of type `val` are either 32- or 64-bit integers or floats, pointers ($\texttt{Vptr}(b, o)$) or undefined values (`Vundef`). During the compilation, source and target memories are related via *injection functions* $j : \texttt{block} \to \lfloor \texttt{block} \times \texttt{Z} \rfloor$ where $\lfloor\ \rfloor$ is overloaded for the `Option` type and its `Some` constructor. $j(b) = \emptyset$ stands for b being removed by compilation and $j(b) = \lfloor (b', o) \rfloor$ if the source block b is embedded at (b', o) in the target memory. The type of j is given the name `meminj`. The values v_1 and v_2 are related by j (denoted as $v_1 \hookrightarrow_v^j v_2$) if either $v_1 = \texttt{Vundef}$, $v_1 = v_2$ for scalar values, or $v_1 = \texttt{Vptr}(b_1, o_1)$, $v_2 = \texttt{Vptr}(b_2, o_2)$ and $j(b_1) = \lfloor (b_2, o_2 - o_1) \rfloor$, i.e., pointers are related by injection. Two memory states m_1 and m_2 are related by j (denoted by $m_1 \hookrightarrow_m^j m_2$) if j is a homomorphism from values in m_1 to those in m_2.

Nominal Memory Model [Wang et al. 2022] is an extension of the block-based memory model using *nominal techniques* [Pitts 2016]. It introduces a nominal interface for the memory model such that the type of memory blocks `block` can be further instantiated by other types. This enables separation of different memory regions (e.g., global memory and stacks for individual threads) by instantiating `block` which we shall exploit later.

3.1.2 Open Simulation of Sequential Semantics. Our work is based on the framework for open simulation in CompCertO [Koenig and Shao 2021]. A *language interface* $A = \langle A^q, A^r \rangle$ is a pair of sets where A^q and A^r denote the type of queries and replies for an open module. The language interface for C semantics is defined as $C = \langle \texttt{val} \times \texttt{sig} \times \texttt{val}^* \times \texttt{mem}, \texttt{val} \times \texttt{mem} \rangle$. A query $v_f[sg](\vec{v})@m$ consists



(a) Initial states (b) Final states (c) Internal steps (d) External calls

Fig. 10. Simulation Diagrams for Open Forward Simulation

of function pointer, signature, arguments and memory. A reply $v'@m'$ contains a return value together with the updated memory. The interface for assembly is $\mathcal{A} = \langle \texttt{regset} \times \texttt{mem}, \texttt{regset} \times \texttt{mem} \rangle$ where its queries and replies have the same form $rs@m$ where rs maps registers to values.

Open labeled transition systems (LTS) represent semantics of modules that may accept queries and provide replies at the *incoming side* and provide queries and accept replies at the *outgoing side* (i.e., calling external functions). An open LTS $L : A \twoheadrightarrow B$ is a tuple $\langle D, S, I, \rightarrow, F, X, Y \rangle$ where A and B are the language interfaces for outgoing and incoming sides, $D \subseteq B^q$ a set of initial queries, S a set of internal states, $I \subseteq D \times S$ and $F \subseteq S \times B^r$ transition relations for incoming queries and replies, $X \subseteq S \times A^q$ and $Y \subseteq S \times A^r \times S$ transitions for outgoing queries and replies, and $\rightarrow \subseteq S \times \mathcal{E}^* \times S$ internal transitions emitting events of type $\mathcal{E}$. Note that $(s, q^O) \in X$ iff an outgoing query q^O happens at s; $(s, r^O, s') \in Y$ iff s receives the reply r^O and resumes with an updated state s'.

Kripke relations are used to describe evolution of program states in open simulations. A Kripke relation $R : W \rightarrow \{S \mid S \subseteq A \times B\}$ is a family of relations indexed by a *Kripke world* W; for simplicity, we define $\mathcal{K}_W(A, B) = W \rightarrow \{S \mid S \subseteq A \times B\}$. The *simulation convention* relating two language interfaces A_1 and A_2 is defined by $\mathbb{R} : A_1 \Leftrightarrow A_2 = \langle W, \mathbb{R}^q : \mathcal{K}_W(A_1^q, A_2^q), \mathbb{R}^r : \mathcal{K}_W(A_1^r, A_2^r) \rangle$

A open forward simulation (denoted by $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$) is defined as [Koenig and Shao 2021]:

Definition 3.1. Given $L_1 : A_1 \twoheadrightarrow B_1$, $L_2 : A_2 \twoheadrightarrow B_2$, $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$, $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ holds if there is some Kripke relation $R \in \mathcal{K}_{W_B}(S_1, S_2)$ that satisfies:

(1) $\forall\ w_B\ q_1\ q_2,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1 \in D_1 \Leftrightarrow q_2 \in D_2)$
(2) $\forall\ w_B\ q_1\ q_2\ s_1,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1, s_1) \in I_1 \Rightarrow \exists\ s_2, (s_1, s_2) \in R(w_B) \wedge (q_2, s_2) \in I_2.$
(3) $\forall\ w_B\ s_1\ s_2\ r_1,\ (s_1, s_2) \in R(w_B) \Rightarrow (s_1, r_1) \in F_1 \Rightarrow \exists\ r_2, (r_1, r_2) \in \mathbb{R}_B^r(w_B) \wedge (s_2, r_2) \in F_2.$
(4) $\forall\ w_B\ s_1\ s_2\ t\ s_1',\ (s_1, s_2) \in R(w_B) \Rightarrow s_1 \xrightarrow{t} s_1' \Rightarrow \exists\ s_2', (s_1', s_2') \in R(w_B) \wedge s_2 \xrightarrow{t}{}^{*} s_2'.$
(5) $\forall\ w_B\ s_1\ s_2\ q_1,\ (s_1, s_2) \in R(w_B) \Rightarrow (s_1, q_1) \in X_1 \Rightarrow$
$\exists w_A\ q_2,\ (q_1, q_2) \in \mathbb{R}_A^q(w_A) \wedge (s_2, q_2) \in X_2 \wedge$
$\forall\ r_1\ r_2\ s_1', (r_1, r_2) \in \mathbb{R}_A^r(w_A) \Rightarrow (s_1, r_1, s_1') \in Y_1 \Rightarrow \exists\ s_2', (s_1', s_2') \in R(w_B) \wedge (s_2, r_2, s_2') \in Y_2.$

Property (1) requires that L_1 and L_2 accept related queries. The remaining properties are illustrated in Fig. 10 where dashed arrows represent existentially quantified relations. Property (2) initializes the simulation invariant $R(w_B)$ from related incoming queries; (3) requires that the final replies are related by the initial world w_B; (4) requires internal execution to preserve the invariant $R(w_B)$; (5) states that if L_1 issues external call q_1, L_2 should issue q_2 related to q_1 by some world w_A derived from the current program states. When the external calls return, the invariant $R(w_B)$ should be established again. Note that the *rely-guarantee* conditions in Fig. 4 is implicitly defined in the Kripke relation of replies $\mathbb{R}^r(w)$ by recording initial memory states in w [Koenig and Shao 2021].

3.1.3 Kripke Memory Relations and Direct Refinements. For the subsequent formal development, we describe the most general definition of KMR `injp` [Zhang et al. 2024], which is for memory protection in verification of the composition of sequential heterogeneous modules.

Definition 3.2 (Kripke Memory Relation). A Kripke Memory Relation is a tuple $\langle W, f, \rightsquigarrow, R\rangle$ where W is a set of worlds, $f : W \to \mathtt{meminj}$ a function for extracting injections from worlds, $\rightsquigarrow \subseteq W \times W$ an accessibility relation between worlds, and $R : \mathcal{K}_W(\mathtt{mem}, \mathtt{mem})$ a Kripke relation over memory states that is compatible with the memory operations. We write $w \rightsquigarrow w'$ for $(w, w') \in \rightsquigarrow$.

Definition 3.3 (Kripke Relation with Memory Protection). $\mathtt{injp} = \langle W_{\mathtt{injp}}, f_{\mathtt{injp}}, \rightsquigarrow_{\mathtt{injp}}, R_{\mathtt{injp}}\rangle$ where $W_{\mathtt{injp}} = (\mathtt{meminj} \times \mathtt{mem} \times \mathtt{mem})$, $f_{\mathtt{injp}}(j, \_, \_) = j$, $(m_1, m_2) \in R_{\mathtt{injp}}(j, m_1, m_2) \Leftrightarrow m_1 \hookrightarrow^{j}_{m} m_2$ and

$$(j, m_1, m_2) \rightsquigarrow_{\mathtt{injp}} (j', m_1', m_2') \;\Leftrightarrow\; j \subseteq j' \wedge \mathsf{unmapped}(j) \subseteq \mathsf{unchanged\text{-}on}(m_1, m_1') \wedge \mathtt{out\text{-}of\text{-}reach}(j, m_1) \subseteq \mathsf{unchanged\text{-}on}(m_2, m_2').$$

$\mathsf{unchanged\text{-}on}(m, m')$ denotes the footprint of memory cells whose permissions and values are not changed from m to m'. $(b_1, o_1) \in \mathsf{unmapped}(j)$ iff $j(b_1) = \emptyset$ (i.e., b_1 is removed from memory by turning into temporary or register values). $(b_2, o_2) \in \mathtt{out\text{-}of\text{-}reach}(j, m_1)$ means if $j(b_1) = \lfloor (b_2, o_2 - o_1)\rfloor$ then (b_1, o_1) is not a valid memory location in m_1 (i.e., (b_2, o_2) cannot be reached from any value in m_1 via injection). $\mathsf{unmapped}$ and $\mathtt{out\text{-}of\text{-}reach}$ exactly define the *private* regions in source and target memories as depicted in Fig. 5. Therefore, the accessibility $\rightsquigarrow_{\mathtt{injp}}$ ensures that private regions are unchanged between function calls and returns (i.e., they are protected).

Definition 3.4 (Direct Refinement). A *direct refinement* is a forward simulation $L_{\mathtt{C}} \leqslant_{\mathtt{CA}_{\mathtt{injp}} \twoheadrightarrow \mathtt{CA}_{\mathtt{injp}}} L_{\mathtt{asm}}$ which directly relates C and assembly semantics. The simulation convention $\mathtt{CA}_{\mathtt{injp}}$ is implicitly defined using $\mathtt{injp}$ as: $\mathtt{CA}_{\mathtt{injp}} = \langle W_{\mathtt{CA}}, \mathbb{R}^{q}_{\mathtt{CA}}, \mathbb{R}^{r}_{\mathtt{CA}}\rangle$ where $W_{\mathtt{CA}} = (W_{\mathtt{injp}}, \mathtt{sig}, \mathtt{regset})$. A world $w = ((j, m_1, m_2), sg, rs)$ consists of related memories, the signature of incoming function and the initial register values. $(v_f[sg](\vec{v})@m_1, rs@m_2) \in \mathbb{R}^{q}_{\mathtt{CA}}(w)$ holds if *1)* w corresponds to the initial queries, i.e. $w = ((j, m_1, m_2), sg, rs)$, *2)* the memories and arguments are related by j, and *3)* the private stack data before the call is indeed in the private memory regions by $\mathtt{injp}$. $(res@m_1', rs'@m_2') \in \mathbb{R}^{r}_{\mathtt{CA}}(w)$ holds if the registers are protected according to the calling convention of CompCert and private memory are protected by $\mathtt{injp}$, i.e., $(j, m_1, m_2) \rightsquigarrow_{\mathtt{injp}} (j', m_1', m_2')$ and $(m_1', m_2') \in R_{\mathtt{injp}}(j', m_1', m_2')$.

3.2 Challenges

Challenges in our formal development of VCC with shared stacks can be summarized at three levels. First and foremost, we need to extend KMR and forward simulation to the threaded versions as described in §2.3 and §2.4. As TKMR requires distinguishing thread local stack blocks from other memory blocks, we need an enriched memory model for its formal definition. Second, with TKMR as the new rely-guarantee conditions, it is unclear whether horizontal and vertical compositionality of forward simulations still hold. In particular, context switches between threads are very different from function calls: the beginning of context switch for one thread is *not* the beginning of context switch for the target thread, but its *end*. Therefore, conflating yield with function calls requires *symmetric* simulation conventions which were not presented in CompCertO. This change combined with the complexity of TKMR incurs major technical challenges in proving the horizontal and vertical compositionalities. Finally, realizing our ideas in a realistic optimizing compiler such as CompCert is highly non-trivial. We need to adapt CompCertO with symmetric rely-guarantee interfaces and to adapt the proofs for CompCert's compiler passes and optimizations to fit into threaded open simulations. Below we discuss our formal development to address these challenges.

4 Threaded Kripke Memory Relations and Simulations

To address the first two challenges, we first introduce a multi-threaded memory model. We then extend $\mathtt{injp}$ into threaded Kripke memory relation $\mathtt{tinjp}$ with internal and external accessibilities. Next, we define threaded simulation conventions and forward simulations using TKMR. Their designs are non-trivial because the internal accessibilities do not fit into the pattern of function calls. Finally, we discuss how to prove the compositionality of threaded forward simulations.

4.1 A Multi-Threaded Memory Model

We implement multi-threaded memory model as an instance of the Nominal Memory Model. The block type `block` is defined as $\texttt{block} = \texttt{nat} \times \texttt{positive}$. A block name $b = (t, p)$ consists of its thread id t and a block identifier p. We write $\texttt{tid}(b)$ for the thread id of b. A memory state m contains a *support* for recording the history of memory blocks allocated in different regions of memory. Its type is $\texttt{sup} = \texttt{nat} \times \texttt{list}\ (\texttt{list}\ \texttt{positive})$.[3] A support $(t, stacks)$ consists of the current running thread t and lists of blocks $stacks$. A thread id t is always non-zero. $stacks[0]$ represents the list of global memory blocks allocated so far, and $stacks[t](t > 0)$ contains stack blocks allocated for thread t. A block $(t, p) \in (t', stacks)$ is valid iff $p \in stacks[t]$. When a new block is allocated in a memory state with support $(t, stacks)$, it is given the fresh name $\texttt{fresh\_block}((t, stacks)) = (t, max(stacks(t)) + 1)$. We shall write $\texttt{tid}(m)$ for the current thread id of the support in memory state m.

With the above memory model, it is possible to decide whether b is allocated by the current thread by comparing $\texttt{tid}(b)$ with $\texttt{tid}(m)$, which is essential for defining accessibilities in TKMR. The semantics of `thread_create` and `yield` affect only the support $(t, stacks)$: `yield` changes the current thread id t to some t' where $1 \le t' < \texttt{length}(stacks)$. `thread_create` appends an empty list after $stacks$ as the name space for the new thread. Notice that, because we have explicitly maintained stacks for individual threads, our memory model is more faithful to how OS views the memory state than previous work. See §7 for more details.

4.2 Threaded Kripke Memory Relations

We define threaded Kripke memory relation by extending Definition 3.2 with two accessibilities as a tuple $\langle W, f, \rightsquigarrow_i, \rightsquigarrow_e, R\rangle$. TKMR with memory protection is defined as follows.

Definition 4.1 (Threaded Kripke Memory Relation with Memory Protection). The threaded version of `injp` is defined as $\texttt{tinjp} = \langle W_{\texttt{tinjp}}, f_{\texttt{tinjp}}, \rightsquigarrow_i^{\texttt{tinjp}}, \rightsquigarrow_e^{\texttt{tinjp}}, R_{\texttt{tinjp}}\rangle$ where the related memories have the same thread id, i.e., $R_{\texttt{tinjp}}(j, m_1, m_2) \Leftrightarrow m_1 \hookrightarrow_m^j m_2$ and $\texttt{tid}(m_1) = \texttt{tid}(m_2)$. $W_{\texttt{tinjp}}$ and $f_{\texttt{tinjp}}$ are the same as in `injp` (Definition 3.3). The internal and external accessibilities are defined as follows:

$$
\begin{aligned}
(j, m_1, m_2) \rightsquigarrow_i^{\texttt{tinjp}} (j', m_1', m_2') \Leftrightarrow\ & j \subseteq j' \wedge \texttt{tid}(m_1) = \texttt{tid}(m_1') \wedge \\
& \texttt{unmapped}(j) \cap \texttt{thr-ext}(m_1) \subseteq \texttt{unchanged-on}(m_1, m_1') \wedge \\
& \texttt{out-of-reach}(j, m_1) \cap \texttt{thr-ext}(m_2) \subseteq \texttt{unchanged-on}(m_2, m_2').
\end{aligned}
$$

$$
\begin{aligned}
(j, m_1, m_2) \rightsquigarrow_e^{\texttt{tinjp}} (j', m_1', m_2') \Leftrightarrow\ & j \subseteq j' \wedge \texttt{tid}(m_1) = \texttt{tid}(m_1') \wedge \\
& \texttt{unmapped}(j) \cap \texttt{thr-int}(m_1) \subseteq \texttt{unchanged-on}(m_1, m_1') \wedge \\
& \texttt{out-of-reach}(j, m_1) \cap \texttt{thr-int}(m_2) \subseteq \texttt{unchanged-on}(m_2, m_2').
\end{aligned}
$$

This is the formal definition of $\rightsquigarrow_i$ and $\rightsquigarrow_e$ we presented in §2.3. The *private* regions in (j, m_1, m_2) are already defined by $\texttt{unmapped}(j) \subseteq m_1$ and $\texttt{out-of-reach}(j, m_1) \subseteq m_2$ via the injection function j in Definition 3.3. $\texttt{thr-ext}(m) \subseteq m$ stands for the *thread-external* blocks, i.e., $b \in \texttt{thr-ext}(m) \Leftrightarrow \texttt{tid}(b) \neq \texttt{tid}(m)$. Similarly, *thread-internal* blocks are defined as $\texttt{thr-int}(m) \subseteq m$ where $b \in \texttt{thr-int}(m) \Leftrightarrow \texttt{tid}(b) = \texttt{tid}(m)$. Note that both $\rightsquigarrow_i$ and $\rightsquigarrow_e$ require the thread id unchanged. The current thread id `tid` is changed by context switches. As a result, although we have a static depiction of private memory (e.g., in Fig. 6b, `j` is always private as it is in $\texttt{unmapped}(j)$), the accessibilities ensure that private stack memory is allowed to be modified by its owner and protected from the other threads.

[3]The terminology *support* comes from nominal techniques [Pitts 2016].

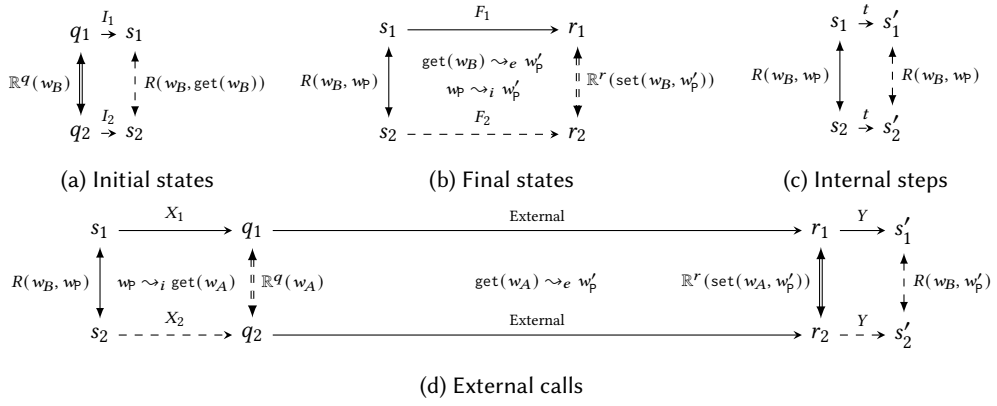


Fig. 11. Simulation Diagrams for Threaded Forward Simulation

4.3 Threaded Forward Simulations

Definition 4.2 (Threaded Simulation Conventions). A *threaded simulation convention* $\mathbb{R}_\mathsf{P} : A_1 \Leftrightarrow A_2 = \langle W, \mathbb{R}^q : \mathcal{K}_W(A_1^q, A_2^q), \mathbb{R}^r : \mathcal{K}_W(A_1^r, A_2^r), \mathsf{P}\rangle$ is defined using a TKMR $\mathsf{P} = \langle W_\mathsf{P}, f_\mathsf{P}, \rightsquigarrow_i^\mathsf{P}, \rightsquigarrow_e^\mathsf{P}, R_\mathsf{P}\rangle$. where W_P is a *sub-world* of W equipped with operations $\mathtt{get} : W \to W_\mathsf{P}$ and $\mathtt{set} : W \to W_\mathsf{P} \to W$.

Compared with the original simulation conventions, there are two differences in the threaded version. First, the accessibility $\rightsquigarrow$ is implicitly encoded in $\mathbb{R}^r$ with a $\Diamond$ modality in the former, while the threaded simulation conventions explicitly contain a TKMR P with internal and external accessibilities. This exposure is necessary for defining the threaded forward simulations as we shall discuss shortly. For simplicity, we often drop the superscript P in $\rightsquigarrow_i^\mathsf{P}$ and $\rightsquigarrow_e^\mathsf{P}$ in our discussions. Second, threaded simulation conventions provide explicit `get` and `set` operators for extracting Kripke worlds W_P for memory states from Kripke worlds W for program states which may contain additional information such as registers and function signatures. This is necessary because internal execution between context switches may relate functions with entirely different arguments and signatures (e.g., `encrypt` and `yield`). To describe these internal guarantees, we need to extract *pure* memory states from program states through `get` and `set`.

Definition 4.3 (Threaded Forward Simulations). A threaded forward simulation $L_1 \leqslant_{\mathbb{R}_\mathsf{P}} L_2$ is defined between open semantics with the same language interfaces on both sides, i.e., for $L_1 : A_1 \twoheadrightarrow A_1$ and $L_2 : A_2 \twoheadrightarrow A_2$. The internal invariant is defined using a Kripke Relation with two worlds $\mathcal{K}_{W_1,W_2}(A, B) : W_1 \to W_2 \to \{S \mid S \subseteq A \times B\}$. Given the threaded simulation convention $\mathbb{R}_\mathsf{P} : A_1 \Leftrightarrow A_2$, $L_1 \leqslant_{\mathbb{R}_\mathsf{P}} L_2$ holds if there exists some Kripke relation $R \in \mathcal{K}_{W,W_\mathsf{P}}(S_1, S_2)$ that satisfies:

(1) $\forall\ w_B\ q_1\ q_2,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1 \in D_1 \Leftrightarrow q_2 \in D_2)$
(2) $\forall\ w_B\ q_1\ q_2\ s_1,\ (q_1, q_2) \in \mathbb{R}_B^q(w_B) \Rightarrow (q_1, s_1) \in I_1 \Rightarrow$
$\quad \exists\ s_2, (s_1, s_2) \in R(w_B, \mathtt{get}(w_B)) \wedge (q_2, s_2) \in I_2.$
(3) $\forall\ w_B\ w_\mathsf{P}\ s_1\ s_2\ r_1,\ (s_1, s_2) \in R(w_B, w_\mathsf{P}) \Rightarrow (s_1, r_1) \in F_1 \Rightarrow$
$\quad \exists\ w'_\mathsf{P}\ r_2, (r_1, r_2) \in \mathbb{R}_B^r(\mathtt{set}(w_B, w'_\mathsf{P})) \wedge (s_2, r_2) \in F_2 \wedge \mathtt{get}(w_B) \rightsquigarrow_e w'_\mathsf{P} \wedge w_\mathsf{P} \rightsquigarrow_i w'_\mathsf{P}.$
(4) $\forall\ w_B\ w_\mathsf{P}\ s_1\ s_2\ t\ s'_1,\ (s_1, s_2) \in R(w_B, w_\mathsf{P}) \Rightarrow s_1 \xrightarrow{t} s'_1 \Rightarrow \exists\ s'_2, (s'_1, s'_2) \in R(w_B, w_\mathsf{P}) \wedge s_2 \xrightarrow{t}{}^{*} s'_2.$
(5) $\forall\ w_B\ w_\mathsf{P}\ s_1\ s_2\ q_1,\ (s_1, s_2) \in R(w_B, w_\mathsf{P}) \Rightarrow (s_1, q_1) \in X_1 \Rightarrow$
$\quad \exists w_A\ q_2,\ (q_1, q_2) \in \mathbb{R}_A^q(w_A) \wedge (s_2, q_2) \in X_2\ \wedge w_\mathsf{P} \rightsquigarrow_i \mathtt{get}(w_A) \wedge$
$\quad\quad \forall\ w'_\mathsf{P}\ r_1\ r_2\ s'_1, \mathtt{get}(w_A) \rightsquigarrow_e w'_\mathsf{P} \Rightarrow (r_1, r_2) \in \mathbb{R}_A^r(\mathtt{set}(w_A, w'_\mathsf{P})) \Rightarrow (s_1, r_1, s'_1) \in Y_1 \Rightarrow$
$\quad\quad\quad \exists\ s'_2, (s'_1, s'_2) \in R(w_B, w'_\mathsf{P}) \wedge (s_2, r_2, s'_2) \in Y_2.$

This definition is an enhancement to Definition 3.1. To deal with symmetric behaviors of context switches as indicated in §3.2, the same simulation convention $\mathbb{R}_\mathsf{P}$ is used for both incoming and outgoing sides (Symmetric interfaces are not a limitation as any LTS $L : A \twoheadrightarrow B$ can be embedded into the interface $A \cup B \twoheadrightarrow A \cup B$). The differences from Definition 3.1 are highlighted in red. The key to proving threaded forward simulation is to verify that internal and external accessibilities hold. For this, the simulation invariant $R(w_B, w_\mathsf{P})$ should remember the relation between source and target initial queries (when entering an open module for the first time) in w_B and the relation between source and target memories *when the last time the module is entered* in w_P (either by an initial function call or a return from an external function call). When $(s_1, s_2) \in R(w_B, w_\mathsf{P})$ holds, it implies that the *current* Kripke world $w_s = (j, m_1, m_2)$ (where m_1 and m_2 are the memory states in s_1 and s_2, respectively, and j is the current memory injection) satisfies both the external guarantee from w_B and the internal guarantee from w_P, i.e., $\mathtt{get}(w_B) \rightsquigarrow_e w_s$ and $w_\mathsf{P} \rightsquigarrow_i w_s$. By maintaining both guarantees as the execution goes, the threaded forward simulation is proved.

Based on the above observations, we elaborate on the ideas behind properties (2-5) which are illustrated in Fig. 11 (property (1) is the same as before). Property (2) initializes the simulation invariant R using the memories $\mathtt{get}(w_B)$ in the initial queries (Fig. 11a). Dually, property (3) describes returning replies r_1 and r_2 related by the updated world $\mathtt{set}(w_B, w'_\mathsf{P})$ from the final invariant R (Fig. 11b). The final memory world w'_P should satisfy both the *external guarantee* $\mathtt{get}(w_B) \rightsquigarrow_e w'_P$ for the whole function call and the *internal guarantee* $w_\mathsf{P} \rightsquigarrow_i w'_\mathsf{P}$ for the internal execution since last time the module was entered. Using the definition of R mentioned above, they can be satisfied by picking the current memory world w_s for w'_P.

Property (4) states that, although the current memories are changed by internal steps, the accessibilities $\mathtt{get}(w_B) \rightsquigarrow_e w_s$ and $w_\mathsf{P} \rightsquigarrow_i w_s$ still hold for the same w_B and w_P (Fig. 11d). For this, one needs to prove that an internal step cannot change the thread local private memory in w_B, and it cannot change the thread external private memory in w_P. In other words, a function call can only change the private stack regions both owned by the current thread and allocated after the initial query. In our running example, the `main` function can modify the loop invariant `i` allocated by itself but can not change `j` because it is owned by another thread. The `encrypt` function also cannot modify `j` because it is already allocated by the server on its private stack before calling `encrypt`.

Finally, for property (5) of external calls (Fig. 11d), when the semantics issue queries q_1 and q_2 related by w_A, the *internal guarantee* of recent internal execution is enforced by $w_\mathsf{P} \rightsquigarrow_i \mathtt{get}(w_A)$. After the external call, we require that the replies are related by $\mathtt{set}(w_A, w'_\mathsf{P})$ where w'_P stands for the updated memory states. Given the rely condition $\mathtt{get}(w_A) \rightsquigarrow_e w'_\mathsf{P}$, we reestablish the internal invariant $(s'_1, s'_2) \in R(w_B, w'_\mathsf{P})$ using the memory world w'_P from replies. Note that one needs to ensure thread local private memory is not changed by the environment (i.e., $w_B \rightsquigarrow_e w'_\mathsf{P}$ holds). For the function `server` in running example, the rely conditions on `encrypt` and `yield` ensure that *any* thread local blocks are unchanged. Therefore, the private memory from its caller is properly protected during the whole execution of `server`. Note also the necessity of `get` and `set` here: we need to relate the memory state w_P initially extracted from w_B to a different world w_A.

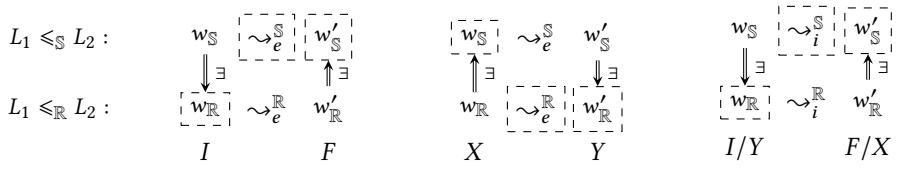
4.4 Compositionality of Threaded Forward Simulations

4.4.1 Vertical Composition. A naive composition theorem for threaded forward simulations holds:

Theorem 4.4. *If* $L_1 : A_1 \twoheadrightarrow A_1, L_2 : A_2 \twoheadrightarrow A_2, L_3 : A_3 \twoheadrightarrow A_3$ *and* $\mathbb{R} : A_1 \Leftrightarrow A_2, \mathbb{S} : A_2 \Leftrightarrow A_3$, *then*

$$L_1 \leqslant_\mathbb{R} L_2 \Rightarrow L_2 \leqslant_\mathbb{S} L_3 \Rightarrow L_1 \leqslant_{\mathbb{R}\cdot\mathbb{S}} L_3.$$

Here, $(\_ \cdot \_)$ is the concatenation of threaded simulation convention s.t. $\mathbb{R} \cdot \mathbb{S} = \langle W_\mathbb{R} \times W_\mathbb{S}, \mathbb{R}^q \cdot \mathbb{S}^q, \mathbb{R}^r \cdot \mathbb{S}^r, \mathsf{P}_\mathbb{R} \cdot \mathsf{P}_\mathbb{S} \rangle$ where $(q_1, q_3) \in \mathbb{R}^q \cdot \mathbb{S}^q \Leftrightarrow \exists\, q_2, (q_1, q_2) \in \mathbb{R}^q(q_1, q_2)$ and $(q_2, q_3) \in \mathbb{S}^q(q_2, q_3)$.



(a) External Guarantee (b) Rely (c) Internal Guarantee

Fig. 12. A Refinement of Threaded Simulation Convention $R \sqsubseteq S$

(same for $\mathbb{R}^r \cdot \mathbb{S}^r$). The TKMRs are also composed where $(w_1, w_2) \rightsquigarrow_i (w'_1, w'_2)$ is defined as $w_1 \rightsquigarrow_i w_2 \wedge w'_1 \rightsquigarrow_i w'_2$ (same for $\rightsquigarrow_e$). Since the concatenated simulation convention is defined as the composition of two relations with an existential quantifier in the middle, one can simply instantiate the existential quantifier using the states of L_2 to complete the proof.

However, this composed interface exposes the internal compilation and weakens the compiler correctness. To get a direct refinement between C and assembly semantics, we need to define the refinement between simulation conventions. Given $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$, we write $\mathbb{R} \sqsubseteq \mathbb{S}$ for that $\mathbb{R}$ is refined by $\mathbb{S}$. The following refinement of threaded simulations can be proved:

Theorem 4.5. *Given* $L_1 : A_1 \twoheadrightarrow A_1$, $L_2 : A_2 \twoheadrightarrow A_2$ *and* $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$,

$$\mathbb{R} \sqsubseteq \mathbb{S} \Rightarrow L_1 \leqslant_{\mathbb{R}} L_2 \Rightarrow L_1 \leqslant_{\mathbb{S}} L_2$$

The key ideas behind its proof are depicted in Fig. 12. To refine the interface $\mathbb{R}$ of the threaded simulation to $\mathbb{S}$, we need to construct multiple worlds with proper accessibilities (denoted using dashed boxes in Fig. 12) as rely-guarantee conditions. In Fig. 12a, given the world $w_{\mathbb{S}}$ of initial queries (I), we first construct $w_{\mathbb{R}}$ to relate the queries such that the property (2) of $L_1 \leqslant_{\mathbb{R}} L_2$ is satisfied. When the semantics return with replies related by $w'_{\mathbb{R}}$ (F), we need to construct an updated $w'_{\mathbb{S}}$ for these replies such that the external guarantee $\rightsquigarrow_e^{\mathbb{S}}$ is satisfied. Symmetrically, in Fig. 12b, when the semantics issue external calls related by $w_{\mathbb{R}}$ (X), we construct $w_{\mathbb{S}}$ for external calls using interface $\mathbb{S}$. After external calls (Y), we construct $w'_{\mathbb{R}}$ and prove $\rightsquigarrow_e^{\mathbb{R}}$ as the rely condition for $L_1 \leqslant_{\mathbb{R}} L_2$. For threaded simulations, we also need to prove the refinement of internal guarantees, as depicted in Fig. 12c. Note that the internal guarantee has four variants: an open module can begin its partial execution by I or Y and end by X or F. Notice that since internal and external guarantees may share the same starting world, we need to prove Fig. 12c together with Fig. 12a and Fig. 12b in one pass.

With the above refinement, vertical composition of threaded forward simulations becomes possible. For example, a common convention is $\mathtt{c_{tinjp}} : C \Leftrightarrow C$ for simulations between semantics with C interfaces using `tinjp`. We can prove its transitivity as the following refinement theorem:

Theorem 4.6. $\mathtt{c_{tinjp}} \cdot \mathtt{c_{tinjp}} \sqsubseteq \mathtt{c_{tinjp}}$

Then, given two threaded forward simulations $L_1 \leqslant_{\mathtt{c_{tinjp}}} L_2$ and $L_2 \leqslant_{\mathtt{c_{tinjp}}} L_3$, we first compose them into $L_1 \leqslant_{\mathtt{c_{tinjp}} \cdot \mathtt{c_{tinjp}}} L_3$ by Theorem 4.4, and then derive $L_1 \leqslant_{\mathtt{c_{tinjp}}} L_3$ by applying Theorem 4.5 to Theorem 4.6. As we shall see in §5, other simulation conventions can support vertical composition in a way similar to $\mathtt{c_{tinjp}}$.

4.4.2 Horizontal Composition. The horizontal composition of threaded forward simulations can be divided into two steps. We first compose open modules assuming that they run on the same thread. Then, we compose open semantics into closed semantics running on a multi-threaded abstract machine. Their closed simulation can be derived from the threaded forward simulations. We call the first step *module linking* and the second step *thread linking*. We discuss their correctness below.

Theorem 4.7 (Correctness of Module Linking). *Let $L_1 \oplus L_2$ denote the semantic linking of L_1 and L_2. Given $L_1, L_1' : A_1 \twoheadrightarrow A_1$, $L_2, L_2' : A_2 \twoheadrightarrow A_2$ and threaded simulation convention $\mathbb{R} : A_1 \Leftrightarrow A_2$, the simulations can be horizontally composed as:*

$$L_1 \leqslant_{\mathbb{R}} L_2 \Rightarrow L_1' \leqslant_{\mathbb{R}} L_2' \Rightarrow L_1 \oplus L_1' \leqslant_{\mathbb{R}} L_2 \oplus L_2'$$

We omit the concrete definition of the semantic linking $L_1 \oplus L_2$. Its proof is similar to the horizontal composition in CompCertO [Koenig and Shao 2021]. The main difference is that we also need to compose *internal guarantee* conditions, which is achieved with the transitivity of $\rightsquigarrow_i$.

The multi-threaded semantics is defined as a closed LTS with a non-deterministic thread-scheduling strategy. A closed LTS is a tuple $\langle S, I, F, \rightarrow \rangle$ containing program state S, predicates for initial and final state I, F and the internal transition $\rightarrow$.

We define *thread linking function* $\mathtt{MT}_C$ and $\mathtt{MT}_{\mathcal{A}}$ for transferring C and assembly open semantics into multi-threaded semantics. We omit the concrete definition for simplicity and present the key ideas using C semantics; the construction for assembly is similar. Given $L : C \twoheadrightarrow C$, $\mathtt{MT}_C(L) = \langle S_M, I_M, F_M, \rightarrow_M \rangle$. The program state S_M includes the current thread id t and a list of *thread states* which are defined using the program state of L. The current global memory can be found in the current thread state. The initial state I_M and final state F_M are defined using the *query* and *reply* of `main` function on the main thread. The main thread applies I (from L) to take a query to `main` as initialization and finally issues a reply using F to produce the return value. The transition step $\rightarrow_M$ has three cases. The internal step is defined as a step of L on the current running thread. The thread creation step happens if the current thread calls `pthread_create` (as an external call). Its arguments and the current memory state are used to create a new thread state. Finally, for context switches, the current thread can switch out by calling `yield` or `thread_join`, or ending its execution. Similarly, the target thread is waiting for a reply of a previous `yield`, a `thread_join`, or thread initialization. The execution simply changes the current thread id, pass it (by constructing a query or reply) to wake up the target thread, and put the current thread into a waiting state.

The correctness of thread linking is stated as follows.

Theorem 4.8 (Correctness of Thread Linking).

$$\forall\, L_1 : C \twoheadrightarrow C, L_2 : \mathcal{A} \twoheadrightarrow \mathcal{A},\ L_1 \leqslant_{\mathbb{C}} L_2 \Rightarrow \mathtt{MT}_C(L_1) \leqslant_{\mathtt{closed}} \mathtt{MT}_{\mathcal{A}}(L_2)$$

Here, $\mathbb{C}$ is the simulation convention derived from CompCertOC which we define in §5.3. The closed simulation $\leqslant_{\mathtt{closed}}$ is exactly the final forward simulation in the original CompCert. This theorem is proved by exploiting the rely-guarantee conditions of threaded forward simulations, especially the internal guarantees, as depicted in Fig. 7b.

5 CompCert for Open and Concurrent Modules

We now discuss how to verify the compiler passes of CompCert and compose them into the threaded forward simulation $\leqslant_{\mathbb{C}}$ as the correctness of CompCertOC. CompCert compiles C programs into `Asm` programs through 20 passes, including optimization passes on the `RTL` intermediate language. Most of the extensions to CompCert start from the `Clight` language which is compiled from C source through the first pass named `SimplExpr` by extracting side effects from expressions. Our development is based on the latest CompCertO [Zhang et al. 2024] which also starts from `Clight` and hence contains 19 passes of CompCert. However, it is recently discovered that the optimization pass `Unusedglob` in CompCertO for removing unused static definitions relies on an incorrect axiom about the transformation of global symbols made by CompCert. As a result, the axioms for CompCertO lead to inconsistency. Fortunately, because this axiom is only used for proving `Unusedglob`, consistency can be recovered by simply removing `Unusedglob` from CompCertO. Therefore, CompCertOC contains 18 passes of CompCert. We believe this omission is orthogonal

Table 1. Significant Passes of CompCertOC

Languages/Passes	Interfaces/Conventions	Languages/Passes	Interfaces/Conventions
Clight	$C \twoheadrightarrow C$	Inlining	$\mathtt{c}_{\mathtt{tinjp}}$
Self-Sim	ro	Constprop	$\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}}$
SimplLocals	$\mathtt{c}_{\mathtt{tinjp}}$	CSE	$\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}}$
Csharpminor	$C \twoheadrightarrow C$	Deadcode	$\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}}$
Cminorgen	$\mathtt{c}_{\mathtt{tinjp}}$	Allocation	$\mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL}$
Cminor	$C \twoheadrightarrow C$	**LTL**	$\mathcal{L} \twoheadrightarrow \mathcal{L}$
Selection	$\mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}}$	Tunneling	$\mathtt{ltl}_{\mathtt{ext}}$
CminorSel	$C \twoheadrightarrow C$	**Linear**	$\mathcal{L} \twoheadrightarrow \mathcal{L}$
RTLgen	$\mathtt{c}_{\mathtt{ext}}$	Stacking	$\mathtt{LM}_{\mathtt{tinjp}}$
RTL	$C \twoheadrightarrow C$	**Mach**	$\mathcal{M} \twoheadrightarrow \mathcal{M}$
Self-Sim	$\mathtt{c}_{\mathtt{tinjp}}$	Asmgen	$\mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
Tailcall	$\mathtt{c}_{\mathtt{ext}}$	**Asm**	$\mathcal{A} \twoheadrightarrow \mathcal{A}$

to our work as Unusedglob only deals with *global memory* while CompCertOC deals with shared *stack memory*. A solution to add back Unusedglob is left for future work.

In the subsections below, we first discuss the verification of individual passes of CompCertOC in §5.1. We then discuss properties for refining simulation conventions used in CompCertOC in §5.2. Finally, we discuss the composition of the correctness of individual passes into a threaded forward simulation with the simulation convention $\mathbb{C}$ that directly relates C and assembly semantics in §5.3.

5.1 Threaded Forward Simulations for Individual Passes

The important compiler passes and their simulation types are listed in Table 1 together with the intermediate languages and their interfaces (which are in bold fonts). The passes on the right follow those on the left in the compilation chain. The passes marked in red are optimizations for RTL. We insert two self-simulations (in blue) for refining the composed simulation convention into a direct refinement. Passes using the identity simulation do not affect the composition and hence are omitted in Table 1 (including Cshmgen, Renumber, Linearize, CleanupLabels and Debugvar).

5.1.1 Simulation Conventions and Semantic Invariants. We first discuss the simulation conventions presented in Table 1. The simulation conventions in the form of $\mathtt{I}_{\mathtt{P}} : \mathcal{X} \Leftrightarrow \mathcal{X}$ use TKMR P to relate the source and target semantics with the language interface $\mathcal{X}$ for language I. $\mathcal{L}$ is the interface for the LTL intermediate language. ext is a simplified version of tinjp which assumes source and target memories have the same structure. Therefore no private memory needs to be protected and the only rely-guarantee condition is the invariance of thread id.

$\mathtt{CL} : C \Leftrightarrow \mathcal{L}$ and $\mathtt{MA} : \mathcal{M} \Leftrightarrow \mathcal{A}$ are *structural conventions* for relating queries and replies in different intermediate languages. They assume that the source and target memory states are the same. Another structural convention $\mathtt{LM}_{\mathtt{tinjp}} : \mathcal{L} \Leftrightarrow \mathcal{M}$ for the Stacking pass is parameterized by tinjp. This is because the source and target memories are related by an injection as the stack pointer, return address and outgoing arguments are allocated on the stack after this pass. Therefore, the private stack in the target memory needs to be protected by tinjp.

ro and wt are *semantic invariants* relating queries and replies in the same language. They describe invariants of an open semantics. wt ensures that the arguments and return values are well-typed. ro ensures that the values of read-only *constants* are never modified by execution.

5.1.2 Verification of Individual Passes. We revise the correctness proofs for the compiler passes in CompCertO so that they fit into the framework of threaded forward simulations. The difficulties

come from two limitations in the original proofs. First, many passes in CompCertO are proved using *different* simulation conventions for the incoming and outgoing sides. Second, in the original proofs, memory protection is only provided at the outgoing side, not at the incoming side. Both limitations need to be lifted because the symmetry of context switches demands the *same* simulation convention and memory protection for *both* the incoming and outgoing sides. Moreover, according to the discussion in §4.3 (after Definition 4.3), to establish a threaded forward simulation for a compiler pass, we need to find a simulation invariant R s.t. $(s_1, s_2) \in R(w_B, w_\mathtt{P})$ iff the external and internal accessibilities $\mathtt{get}(w_B) \rightsquigarrow_e w_s$ and $w_\mathtt{P} \rightsquigarrow_i w_s$ hold for the current Kripke world w_s extracted from s_1 and s_2. The most difficult passes to prove are optimizations involving static analysis (`ConstProp`, `CSE` and `DeadCode`) and `Stacking` for concretely laying out the stack frames. For the former, we need to prove that the blocks whose addresses are never taken (hence may be optimized away) are always labeled as private in the memory world $w_\mathtt{P}$ in $R(w_B, w_\mathtt{P})$. For the latter, because the original `Stacking` pass uses different simulation conventions for the incoming and outgoing sides, we have to do a substantial rewrite of the proof with the new symmetric structural convention $\mathtt{LM}_\mathtt{tinjp}$ and prove that the newly added private stack is protected by an invariant R.

5.2 Properties for Refining Thread Simulation Conventions

We adopt the techniques discussed in §4.4.1 to compose and refine the correctness of individual passes into that of CompCertOC. For this, we discuss the necessary refinement lemmas for threaded simulation conventions. First, the following lemmas are for refining C-level simulation conventions:

Lemma 5.1. (1) $\mathtt{c}_\mathtt{tinjp} \cdot \mathtt{c}_\mathtt{tinjp} \sqsubseteq \mathtt{c}_\mathtt{tinjp}$ (2) $\mathtt{c}_\mathtt{ext} \cdot \mathtt{c}_\mathtt{ext} \sqsubseteq \mathtt{c}_\mathtt{ext}$
(3) $\mathtt{c}_\mathtt{tinjp} \cdot \mathtt{c}_\mathtt{ext} \cdot \mathtt{c}_\mathtt{tinjp} \sqsubseteq \mathtt{c}_\mathtt{tinjp}$ (4) $\mathtt{ro} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{ro} \cdot \mathtt{c}_\mathtt{tinjp} \sqsubseteq \mathtt{ro} \cdot \mathtt{c}_\mathtt{tinjp}$.

Property (1) is exactly Theorem 4.6. (2) and (3) are trivial because `ext` does not protect any private memory. (4) is an extension of (1) for transitively composing the `ro` invariant with `tinjp`.

Lemma 5.2. *For* $K \in \{\mathtt{ext}, \mathtt{tinjp}\}$, (1) $\mathtt{c}_K \cdot \mathtt{CL} \equiv \mathtt{CL} \cdot \mathtt{ltl}_K$ (2) $\mathtt{mach}_K \cdot \mathtt{MA} \equiv \mathtt{MA} \cdot \mathtt{asm}_K$.

Here, $\mathbb{R} \equiv \mathbb{S} \Leftrightarrow \mathbb{R} \sqsubseteq \mathbb{S} \wedge \mathbb{S} \sqsubseteq \mathbb{R}$. Since the structural conventions `CL` and `MA` assume the same memory for source and target, we can freely move conventions with TKMR through them. The refinement lemmas for $\mathtt{LM}_\mathtt{tinjp}$ are as follows:

Lemma 5.3. (1) $\mathtt{ltl}_\mathtt{tinjp} \cdot \mathtt{ltl}_\mathtt{ext} \cdot \mathtt{LM}_\mathtt{tinjp} \sqsubseteq \mathtt{LM}_\mathtt{tinjp}$ (2) $\mathtt{CL} \cdot \mathtt{LM}_\mathtt{tinjp} \cdot \mathtt{MA} \sqsubseteq \mathtt{CA}_\mathtt{tinjp}$

Property (1) is based on the same compositionality of TKMR as property (3) in Lemma 5.1. Property (2) composes the structural relations to hide the intermediate languages.

Finally, the permutation and elimination lemmas for `wt` can be proved as follows:

Lemma 5.4. (1) $\mathtt{ro} \cdot \mathtt{wt} \equiv \mathtt{wt} \cdot \mathtt{ro}$ (2) *For* $K \in \{\mathtt{ext}, \mathtt{tinjp}\}$, $\mathtt{c}_K \cdot \mathtt{wt} \sqsubseteq \mathtt{wt} \cdot \mathtt{c}_K$
(3) $\mathtt{wt} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{wt} \sqsubseteq \mathtt{wt} \cdot \mathtt{c}_\mathtt{tinjp}$

5.3 Composing the Threaded Forward Simulations

For composing individual correctness proofs, we first insert the self-simulations as presented in Table 1. The self-simulation for each intermediate language can be easily proved. The trivial vertical composition of simulation conventions by Theorem 4.4 for all passes results in:

$$\mathbb{R} = \mathtt{ro} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{wt} \cdot \mathtt{c}_\mathtt{ext} \cdot \mathtt{c}_\mathtt{ext} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{c}_\mathtt{ext} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{ro} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{ro} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{ro} \cdot \mathtt{c}_\mathtt{tinjp} \cdot \mathtt{wt} \cdot \mathtt{c}_\mathtt{ext} \cdot \mathtt{CL} \cdot \mathtt{ltl}_\mathtt{ext} \cdot \mathtt{LM}_\mathtt{tinjp} \cdot \mathtt{mach}_\mathtt{ext} \cdot \mathtt{MA}$$

Our goal is to refine $\mathbb{R}$ into the simulation convention $\mathbb{C}$ for the whole compiler:

$$\mathbb{C} = \mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CA}_\mathtt{tinjp} \cdot \mathtt{asm}_\mathtt{ext}$$

Note that ro and wt apply to source C programs and $\mathtt{asm}_{\mathtt{ext}}$ applies to the target assembly program. They are all irrelevant because ro and wt are always satisfied by any well-behaved C semantics, and any well-behaved assembly semantics is self-simulating. Therefore, the only important component is $\mathtt{CA}_{\mathtt{tinjp}}$ which *directly* relates C and assembly semantics through tinjp. In particular, $\mathtt{CA}_{\mathtt{tinjp}} : \mathcal{C} \Leftrightarrow \mathcal{A} = \langle W_{\mathtt{CA}}, \mathbb{R}^q_{\mathtt{CA}}, \mathbb{R}^r_{\mathtt{CA}}, \mathtt{tinjp}\rangle$ where $W_{\mathtt{CA}} = (W_{\mathtt{tinjp}}, \mathtt{sig}, \mathtt{regset})$. The get and set operations for the sub world $W_{\mathtt{tinjp}}$ are simply defined as getting and setting the first component of $w \in W_{\mathtt{CA}}$. $\mathbb{R}^q_{\mathtt{CA}}$ and $\mathbb{R}^r_{\mathtt{CA}}$ are almost the same as in Definition 3.4. The only difference is that the accessibilities are explicitly exposed in tinjp.

We prove $\mathbb{R} \sqsubseteq \mathbb{C}$ by a sequence of refined simulation conventions $\mathbb{R} \sqsubseteq \mathbb{R}_1 \sqsubseteq \ldots \sqsubseteq \mathbb{R}_n \sqsubseteq \mathbb{C}$:

(1) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(2) $\mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(3) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(4) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(5) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c}_{\mathtt{tinjp}} \cdot \mathtt{c}_{\mathtt{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{mach}_{\mathtt{ext}} \cdot \mathtt{MA}$
(6) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CL} \cdot \mathtt{ltl}_{\mathtt{tinjp}} \cdot \mathtt{ltl}_{\mathtt{ext}} \cdot \mathtt{LM}_{\mathtt{tinjp}} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{ext}}$
(7) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CA}_{\mathtt{tinjp}} \cdot \mathtt{asm}_{\mathtt{ext}}$

The red letters mark the simulation conventions refined in each step. In step (1), we use all properties in Lemma 5.1 to compose conventions involving $\mathtt{c}_{\mathtt{tinjp}}$. In step (2-3), we use the properties in Lemma 5.4 to move wt upwards, compose them using $\mathtt{c}_{\mathtt{tinjp}}$, and commute the first ro with wt. In step (4), we do the composition of TKMRs again using Lemma 5.1. In step (5), we transfer the c simulation conventions to ltl and the $\mathtt{mach}_{\mathtt{ext}}$ to asm using Lemma 5.2. The invariants for the source program are swapped again using property (1) in Lemma 5.4. Finally, we compose the conventions on ltl and structural conventions CL and MA with $\mathtt{LM}_{\mathtt{tinjp}}$ using Lemma 5.3 to get $\mathbb{C}$.

The final correctness of CompCertOC is presented below:

Theorem 5.5 (Compiler Correctness of CompCertOC).

$$\forall\ (M : \mathtt{Clight})(M' : \mathtt{Asm}),\ \mathtt{CompCertOC}(M) = M' \Rightarrow [\![M]\!] \leqslant_{\mathbb{C}} [\![M']\!] \wedge [\![M]\!] \geqslant_{\mathbb{C}} [\![M']\!].$$

The threaded forward simulation $[\![M]\!] \leqslant_{\mathbb{C}} [\![M']\!]$ is easily proved by applying Theorem 4.5 to $\mathbb{R} \sqsubseteq \mathbb{C}$. We also get a threaded *backward* simulation $\geqslant_{\mathbb{C}}$ between the source and target modules. It is defined as a generalization of backward simulations in CompCert. We are able to flip threaded forward simulation into backward simulation by showing the source semantics (i.e., Clight) is *receptive* and the target (i.e., Asm) is *determinate*. The proof is almost the same as in CompCert.

6 Application and Evaluation

In this section, we formally verify the running example as depicted in Fig. 3. We also evaluate our Coq development of CompCertOC comparing to CompCertO [Zhang et al. 2024].

6.1 Verification of Heterogeneous Multi-threaded Modules

We first define the C-level specification $L_{\mathtt{E}}$ for encrypt.s. Here $X_{\mathtt{E}}$ and $Y_{\mathtt{E}}$ are empty sets and omitted because encrypt does not contain external calls.

Definition 6.1. LTS of $L_{\mathtt{E}}$ is defined as

$S_{\mathtt{E}} := \{\mathtt{Initial}\ m\ i\ r, \mathtt{Final}\ m\};$
$I_{\mathtt{E}} := \{(\mathtt{Vptr}(b_e, 0)[\mathtt{int} \to \mathtt{ptr} \to \mathtt{void}]([i, r])@m, \mathtt{Initial}\ m\ i\ r)\};$
$\to_{\mathtt{E}} := \{\mathtt{Initial}\ m\ i\ r, \mathtt{Final}\ m') \mid m' = m[*r \leftarrow (i\ \mathtt{XOR}\ 42)]\}$
$F_{\mathtt{E}} := \{(\mathtt{Final}\ m, \mathtt{Vundef}@m)\}.$

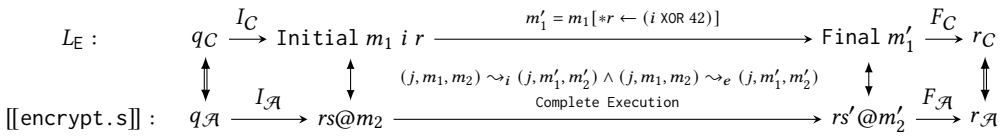


Fig. 13. Simulation Diagram for $L_{\mathsf{E}} \leqslant_{\mathtt{CA}_{\mathtt{tinjp}}} [\![\mathtt{encrypt.s}]\!]$

We then prove the threaded forward simulation of `encrypt` as:

Lemma 6.2. $L_{\mathsf{E}} \leqslant_{\mathbb{C}} [\![\mathtt{encrypt.s}]\!]$.

Proof Sketch. Observe $\mathbb{C} = \mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CA}_{\mathtt{tinjp}} \cdot \mathtt{asm}_{\mathtt{ext}}$. The source invariants on L_{E} hold according to its definition. The self-simulation using $\mathtt{asm}_{\mathtt{ext}}$ holds for any CompCert assembly. The proof of $L_{\mathsf{E}} \leqslant_{\mathtt{CA}_{\mathtt{tinjp}}} [\![\mathtt{encrypt}]\!]$ is depicted in Fig. 13. We need to verify the internal and external guarantee conditions of the whole execution which is straightforward because `encrypt` only changes the public memory pointed by r in both source and target semantics. Note that even if `encrypt` can yield back to other threads or allocate stack memory, this proof will not be more complicated by much. This is because `yield` is treated as a regular call. With public stack memory, we only need to modify L_E to allow it be shared and the remaining stack be protected. □

To propagate the simulation to linked assembly code, we also prove the equivalence of syntactic and semantics linking of assembly modules:

Theorem 6.3. $\forall (M\ M' : \mathtt{Asm}), [\![M]\!] \oplus [\![M']\!] \leqslant_{\mathtt{id}} [\![M + M']\!]$.

To get the final correctness result, we first perform module linking by applying Theorem 4.7 to compose Lemma 6.2 and the compiler correctness of `client.c` and `server.c` obtained by applying Theorem 5.5. These results are then vertically extended to the linked target assembly by using Theorem 6.3 (`id` is absorbed using $\mathbb{C} \cdot \mathtt{id} \sqsubseteq \mathbb{C}$). It gets us the following result:

Lemma 6.4 (Threaded Forward Simulation for Composed Modules).

$$[\![\mathtt{client.c}]\!] \oplus [\![\mathtt{server.c}]\!] \oplus L_{\mathsf{E}} \leqslant_{\mathbb{C}} [\![\mathtt{client.s} + \mathtt{server.s} + \mathtt{encrypt.s}]\!]$$

As mentioned in §5.3, we are able to flip threaded forward simulations into backward simulations by showing the composed C-level semantics is receptive. Moreover, our correctness of thread linking (Theorem 4.8) remains valid for backward simulation with additional determinacy and receptiveness requirements on the source semantics, which can be satisfied by the composed C-level semantics. Therefore, we are able to prove the closed backward simulation for our running example, which has the same definition as in the original CompCert.

Theorem 6.5 (Backward Simulation for the Complete Program).

$$\mathtt{MT}_C([\![\mathtt{client.c}]\!] \oplus [\![\mathtt{server.c}]\!] \oplus L_{\mathsf{E}}) \geqslant_{\mathtt{closed}} \mathtt{MT}_{\mathcal{A}}([\![\mathtt{client.s} + \mathtt{server.s} + \mathtt{encrypt.s}]\!])$$

6.2 Evaluation

Our Coq development took about 10 person-months and contains 21.4k lines of code (LOC) on top of the latest CompCertO [Zhang et al. 2024]. We added 1.0k LOC to implement the multi-stack memory model based on Nominal Memory Model, 2.6k LOC to define the framework of threaded simulations with module linking. We modified the proof of 13 compilation passes (listed in Table 1 using TKMR) by *adding* 2.8k LOC on top of 19.3k LOC in CompCertO, which is about 14% increase. We added 7.6k LOC to verify the refinements of TKMRs and thread simulation conventions. We defined the multi-threaded semantics based on open semantics and verified the thread linking by 6.3k LOC. Finally, we verified the running example with 1.1k LOC.

Table 2. Comparison of Work on Verified Compositional Compilation

	CCC	CCM	CCO	CCTSO	CCX	CASCC	**CCOC**
Cooperative Concurrency	No	No	No	No	Yes	Yes	**Yes**
Preemptive Concurrency	No	No	No	Yes	No	Yes	**No**
Stack Sharing	No	No	No	Yes	No	No	**Yes**
Direct Refinement	No	No	Yes	Closed	No	No	**Yes**
Vertical Composition	Yes	RUSC	Yes	No	CCAL	Yes	**Yes**
Horizontal Composition	Yes	RUSC	Yes	No	CCAL	Yes	**Yes**
Heterogeneous Modules	Yes	Yes	Yes	No	Yes	Yes	**Yes**

7 Related Work

We are concerned with VCC of imperative programs with pointers. Compared to languages without pointers, this significantly complicates memory protection on shared stacks. We compare CompCertOC (CCOC) with other work on VCC not supporting concurrency, including Compositional CompCert (CCC) [Stewart et al. 2015], CompCertM (CCM) [Song et al. 2020], CompCertO (CCO) [Koenig and Shao 2021; Zhang et al. 2024], and those with concurrency, including Thread-safe CompCertX (CCX) [Gu et al. 2018], CASCompCert (CASCC) [Jiang et al. 2019a] and CompCertTSO (CCTSO) [Sevcík et al. 2013]. Table 2 summarizes this comparison where each row displays one feature the verified compilers support or not (texts in brown indicate that a weaker variant of the feature is supported). *Direct refinement* means that the source and target semantics are related at their *native* language interfaces and without exposing any intermediate semantics; it is important for compositionality and usability by third-parties. We elaborate on the comparison below.

Verified Compositional Compilation without Concurrency. Compositional CompCert (CCC) is the pioneering work on VCC of imperative programs [Stewart et al. 2015]. Its *interaction semantics* provide the simulation-based foundation for VCC which is adopted and enhanced by many projects (including this one). Its main limitation is that every language semantics and simulation must conform to the C interface, with which direct refinements cannot be supported. CompCertM (CCM) fixes this problem by introducing *Refinement Under Self-related Contexts* or RUSC [Song et al. 2020] which is the collection of all intermediate semantics and simulation relations in compilation. However, a direct refinement is still not possible because the fixed collection exposes intermediate semantics. Instead, CompCertO (CCO) defines the transitive composition of all intermediate simulations as compiler correctness [Koenig and Shao 2021]. It is later shown that this transitive composition is equivalent to a direct refinement by exploiting the memory protection of Kripke Memory Relation [Zhang et al. 2024]. Therefore, CompCertO for the first time supports all the important properties for verified compilation of heterogeneous modules (i.e., the last four rows in Table 2) and provides the basis for developing CompCertOC.

Verified Compositional Compilation with Concurrency. An early work on verified compilation of concurrent programs is CompCertTSO (CCTSO) [Sevcík et al. 2013] which is a whole-program compiler for multi-threaded programs. Although CCTSO supports stack sharing, it does not support open modules or separate compilation due to the closed nature of its compiler correctness.

Thread-safe CompCertX (CCX) [Gu et al. 2018] extends CompCertX [Gu et al. 2015]—an extension of CompCert supporting mixed C and assembly programs—with the support of open threads. Its compositionality is related to Concurrent Certified Abstraction Layers (CCAL) [Gu et al. 2016] which disallows mutual calls between modules (i.e., no callback function). Moreover, because CCX puts all memory blocks into a single space like CompCert, it cannot support thread-local stacks. To avoid the complexity brought by this limitation, CCX forbids sharing of stack data.

CASCompCert (CASCC) is the first extension of CompCert that supports general recursion and concurrency [Jiang et al. 2019a]. Being built on Compositional CompCert, it inherits its limitations including forcing languages into a C-like interface, resulting in loss of direct refinements. CASCC supports thread-local stacks by dividing CompCert's memory state into stack and global memory space. However, it is unclear how to enable stack sharing while maintaining compositionality in CASCC. An appendix of 22 pages in the technical report of CASCC describes a solution on paper [Jiang et al. 2019b]. However, due to its complexity, this solution has not been formalized.

Therefore, CompCertOC is the first extension of CompCert that supports concurrency with stack sharing. Its correctness is described as *forward simulation* between *cooperative semantics*. The original CompCert has shown that forward simulations can be flipped into backward simulations if the source semantics is *receptive* and the target one is *determinate*. The source C semantics and the target assembly semantics in CompCertOC, CASCC and CCX all satisfy those properties. Therefore, their forward simulations can all be flipped into backward ones.

In both CASCC and CCX, a compiler correctness theorem is first stated against cooperative semantics where context switches are treated as external function calls. Then, a separate layer of verification framework on top of this compiler correctness proves that cooperative semantics is equivalent to preemptive ones. This separation makes the complexity of verified compilers much more manageable. We follow the same approach in this work. To extend our results to support preemptive semantics, we could follow CASCC to prove the equivalence between cooperative and preemptive semantics by exploiting a DRF-SC theorem to shuffle context switch points to thread primitives. We could also follow CCX to prove this equivalence in a program verification framework like CCAL. Those extra proofs are about program verification and left for future work.

Like CompCert, all stack blocks in CCX are in the same memory space [Gu et al. 2018]. Therefore, the stack blocks for different threads interleave with each other, making it difficult to link with a realistic machine model. CASCC alleviates this problem by creating separate memory space for stacks in an ad-hoc manner. In CompCertOC, we directly extend CompCert's memory model with a multi-stack name space. Because this extension is based on a nominal interface, it does not change the memory operations or break any existing properties of CompCert's memory model. Moreover, this treatment of stacks has already enabled elimination of pseudo instructions in CompCert's assembly code and further compilation to real machine code as demonstrated by Wang et al. [2022]. Therefore, our multi-stack memory model is closer to how an OS views the stack (each thread has its own stack) [Gu et al. 2018] and may enable further compilation to realistic machine models.

Relationship with other Concurrency Models and Separation Logics. Although we only consider sequential consistency in this work, we believe our ideas also apply to stronger concurrency models such as linearizability. Linearizable objects can be described as transition systems where methods take effect non-deterministically and atomically [Herlihy and Wing 1990]. Therefore, our module and thread linking theorems for closed programs may be extended to support well-encapsulated objects. A more difficult problem is to support relaxed memory models. Based on existing work [Cho et al. 2022; Kang et al. 2017; Lee et al. 2020; Zha et al. 2022], we need to develop new rely-guarantee simulations that allow both stack sharing and out-of-order execution, which is left for future work.

One may wonder if the rely-guarantee reasoning can be avoided by using separation logics. From our experience, the rely-guarantee reasoning in compiler verification needs to be ignorant of program structures and contexts, so that the compiler correctness can remain fully composable. By contrast, the algebras in separation logic are more geared towards reasoning about *specific programs* (i.e., program verification). It is not clear how to directly apply them to substitute rely-guarantee reasoning. Recently, Conditional Contextual Refinement [Song et al. 2023] has combined separation logics with refinements. Its application to compiler verification needs further investigation.

Data-Availability Statement

The Coq artifact containing the formal developments in this paper is available on Zenodo [Zhang et al. 2025].

Acknowledgments

We would like to thank the anonymous referees for their helpful feedback which improved this paper significantly. This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62372290 and 62002217. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

References

Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential reasoning for optimizing compilers under weak memory concurrency. In *Proc. 2022 ACM Conference on Programming Language Design and Implementation (PLDI'22)*. ACM, New York, NY, USA, 213–228. doi:10.1145/3519939.3523718

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, Sriram K. Rajamani and David Walker (Eds.). ACM, New York, NY, USA, 595–608. doi:10.1145/2775051.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, GA, 653–669.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahnia Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proc. 2018 ACM Conference on Programming Language Design and Implementation (PLDI'18)*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, New York, NY, USA, 646–661. doi:10.1145/3192366.3192381

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL'12)*, John Field and Michael Hicks (Eds.). ACM, New York, NY, USA, 59–72. doi:10.1145/2103656.2103666

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019a. Towards Certified Separate Compilation for Concurrent Programs. In *Proc. 2019 ACM Conference on Programming Language Design and Implementation (PLDI'19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, New York, NY, USA, 111–125. doi:10.1145/3314221.3314595

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019b. Towards Certified Separate Compilation for Concurrent Programs. https://plax-lab.github.io/publications/ccc/ccc-tr.pdf

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *Proc. 44th ACM Symposium on Principles of Programming Languages (POPL'17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, New York, NY, USA, 175–189. doi:10.1145/3009837.3009850

Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proc. 2021 ACM Conference on Programming Language Design and Implementation (PLDI'21)*. ACM, New York, NY, USA, 1095–1109. doi:10.1145/3453483.3454097

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proc. 2020 ACM Conference on Programming Language Design and Implementation (PLDI'20)*. ACM, New York, NY, USA, 362–376. doi:10.1145/3385412.3386010

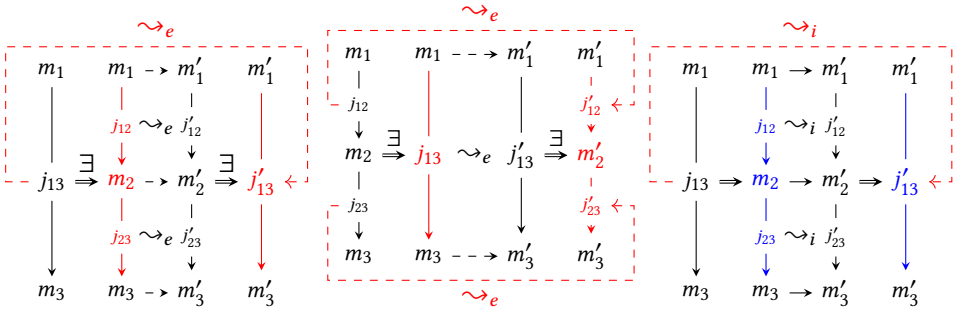
Xavier Leroy. 2005–2023. The CompCert Verified Compiler. https://compcert.org/.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL'12)*. ACM, New York, NY, USA, 455–468. doi:10.1145/2103621.2103711

Rust Standard Library. 2024. std::thread::scope. https://doc.rust-lang.org/std/thread/fn.scope.html

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a Compositionally Verified Compiler for a Higher-Order Imperative Language. In *Proc. 2015 ACM SIGPLAN International*



(a) External Guarantee (b) Rely (c) Internal Guarantee

Fig. 14. Composition of Accessibility Relation for `tinjp` · `tinjp` ⊑ `tinjp`

Conference on Functional Programming (ICFP'15), Kathleen Fisher and John H. Reppy (Eds.). ACM, New York, NY, USA, 166–178. doi:10.1145/2784731.2784764

Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (August 2019), 29 pages. doi:10.1145/3341689

Andrew M. Pitts. 2016. Nominal Techniques. *ACM SIGLOG News* 3, 1 (2016), 57–72. doi:10.1145/2893582.2893594

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. doi:10.1145/2487241.2487248

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (January 2020), 31 pages. doi:10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (January 2023), 31 pages. doi:10.1145/3571232

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, NY, USA, 275–287. doi:10.1145/2676726.2676985

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (January 2019), 30 pages. doi:10.1145/3290375

Yuting Wang, Ling Zhang, Zhong Shao, and Jérémie Koenig. 2022. Verified Compilation of C Programs with a Nominal Memory Model. *Proc. ACM Program. Lang.* 6, POPL, Article 25 (January 2022), 31 pages. doi:10.1145/3498686

Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. In *Proc. 2022 ACM Conference on Programming Language Design and Implementation (PLDI'22)*. ACM, New York, NY, USA, 903–917. doi:10.1145/3519939.3523734

Ling Zhang, Yuting Wang, Yalun Liang, and Zhong Shao. 2025. CompCertOC: Verified Compositional Compilation of Multi-Threaded Programs with Shared Stacks (Artifact). https://doi.org/10.5281/zenodo.15012729

Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2024. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules. *Proc. ACM Program. Lang.* 8, POPL, Article 72 (January 2024), 31 pages. doi:10.1145/3632914

A Transitivity of `tinjp`

Theorem A.1. $\mathtt{C}_{\mathtt{tinjp}} \cdot \mathtt{C}_{\mathtt{tinjp}} \sqsubseteq \mathtt{C}_{\mathtt{tinjp}}$

In order to prove the composition of `tinjp`, we are essentially proving the a composed interface for two simulations. According to Definition 4.3. The point to preserve the three accessibilities. Firstly, the *external guarantee* can be preserved by the construction depicted in Fig. 14a where black symbols are the assumptions (∀-quantified) and red ones are conclusions we need to construct (∃-quantified).

For external guarantee, from the memories m_1 and m_3 related by j_{13} from initial queries, we construct the mid-level memory m_2 as m_1 by picking $j_{12}(b) = \lfloor (b, 0) \rfloor$ if $j_{13} \neq \emptyset$ and $j_{23} = j_{13}$ such that $m_1 \hookrightarrow_m^{j_{12}} m_2$ and $m_2 \hookrightarrow_m^{j_{23}} m_3$ hold. After the *complete execution* (denoted by the dashed arrow) of the modules, we need to construct $j'_{23} = j'_{12} \cdot j'_{23}$ for the replies and prove $(j_{13}, m_1, m_3) \rightsquigarrow_e (j'_{13}, m'_1, m'_3)$ by composing $(j_{12}, m_1, m_2) \rightsquigarrow_e (j'_{12}, m'_1, m'_2)$ and $(j_{23}, m_2, m_3) \rightsquigarrow_e (j'_{23}, m'_2, m'_3)$. This can be trivially derived because the construction of m_2 and j_{12} ensure that all external private memory in m_1 and m_3 are protected by the executions.

For rely, the construction is in the reversed direction as depicted in Fig. 14b. We first compose the memories for external queries $m_1 \hookrightarrow_m^{j_{12}} m_2$ and $m_2 \hookrightarrow_m^{j_{23}} m_3$ into $m_1 \hookrightarrow_m^{j_{13}} m_3$. After the external evolution $(j_{13}, m_1, m_3) \rightsquigarrow_e (j'_{13}, m'_1, m'_3)$, we need to construct an updated mid-level memory state m'_2 such that $(j_{12}, m_1, m_2) \rightsquigarrow_e (j'_{12}, m'_1, m'_2)$ and $(j_{23}, m_2, m_3) \rightsquigarrow_e (j'_{23}, m'_2, m'_3)$ hold. The construction of m'_2 is similar as the composition of the sequential version of `injp` introduced in [Zhang et al. 2024] by copying the values of *public* regions (in both newly allocated blocks and old blocks) and leaving the *private* regions in m_2 unchanged.

For threaded `tinjp`, we also need to compose the internal guarantee of internal executions as depicted in Fig. 14c. This figure has the same shape with Fig. 14a while changing $\rightsquigarrow_e$ into $\rightsquigarrow_i$ for the thread internal execution from m_1 to m_2 (denoted by solid arrows). One may note that we represent the constructed parts by blue symbols because the constructions of memory states here have different situations and are *already defined*. For a open simulation, we have two ways of entering and exiting a module, respectively. The construction of m_2 for entering could from initialization (i.e. the left part of Fig. 14a) or returning from external calls (i.e. the right part of Fig. 14b). Similar the derivation of $(j_{13}, m_1, m_3) \rightsquigarrow_i (j'_{13}, m'_1, m'_3)$ for exiting comes from either final returns (i.e. the right part of Fig. 14a) or external calls (i.e. the left part of Fig. 14b). In order to unify different constructions of m_2, we defined the invariant $R_W \subseteq W_{\texttt{tinjp}\cdot\texttt{tinjp}} \times W_{\texttt{tinjp}}$ for the constructions at I and Y as $((j_{12}, m_1, m_2), (j_{23}, m_2, m_3)), (j_{13}, m_1, m_3)) \in R_W$ such that $j_{13} = j_{23} \cdot j_{12}$ and m_2 does not contain any *thread external* memory region that is *private in exactly one of* (j_{12}, m_1, m_2) *and* (j_{13}, m_2, m_3). For example, `client.c` is compiled through multiple compilation passes while the variable `mask` is removed from the memory and *becomes* private since one specific pass. Therefore, the block of `mask` in the memory before this pass is considered private regarding to the target semantics but public to the source semantics. However, we want to make sure that this is not the case for `size` from `server.c` because this will allow the internal execution of `client.c` to changed it as public region. It is possible because the memories for intermediate semantics are constructed (at I and Y) while we only add public threaded external blocks which exist in both source and target semantics to them. Using this invariant as precondition, we are able to prove the composed internal guarantee in Fig. 12c.

B Verification of Individual Passes

B.0.1 *Verification of* `Stacking` *Pass.*

Lemma B.1 (Correctness of `Stacking` pass). $\forall (M : \texttt{Linear})(M' : \texttt{Mach}), \texttt{Stacking}(M) = M' \Rightarrow [[M]] \leqslant_{\texttt{LM}_{\texttt{tinjp}}} [[M']]$.

$\texttt{LM}_{\texttt{tinjp}}$ is similar to $\texttt{CA}_{\texttt{tinjp}}$ where its world type $W_{\texttt{LM}}$ also contains $W_{\texttt{tinjp}}$ for memories, the function signature *sg* and the register set *rs*. $\texttt{LM}_{\texttt{tinjp}}$ also requires that the callee-saved registers are the same from the target query to reply.

In order to prove this simulation, we need to find an invariant $R : \mathcal{K}_{W_{\texttt{LM}}, W_{\texttt{tinjp}}}$ for source and target program states. For presentation, we simplify these states into just memories by ignoring other parts like control stack or registers. As mentioned in §4.3, $(m_1, m_2) \in R(w_B, w_P)$ is defined as $\exists j, w_B \rightsquigarrow_e (j, m_1, m_2) \wedge w_P \rightsquigarrow_i (j, m_1, m_2)$ where the internal memory world (j, m_1, m_2) is written

as w_s. Using this invariant, the proofs for initial states (Fig. 11a), final states (Fig. 11b) and external calls (Fig. 11d) are straightforward. We illustrate how to prove the internal step (Fig. 11c) by a snapshot of stack frames in Fig. 15. The shaded regions are *private* regions at the target level. The proof goal is to show that the internal execution preserves $R(w_B, w_P)$, i.e. it can only modify the private regions *allocated by itself*. Here M and M' are called by f to run function g on thread A. h is another function runs on thread B. The $w_B \rightsquigarrow_e w_s$ and $w_P \rightsquigarrow_i w_s$ from internal invariant protect the private regions in b'_f and b'_h, respectively. Although g can visit the stack frames of f and h, the source program M can only access the public regions in b_f and b_h. Therefore, forward simulation ensures that M' does not change *local private data* in b'_f and *external private data* in b'_h.

Assuming that M and M' are called by f and run function g on thread A. The blocks b_f and b'_f are included in initial world w_B while b_g and b'_g are newly allocated and not in w_B. During the execution of M (and M'), stack frames of another function h is allocated on thread B. When thread B switches back to A, b_h and b'_h are included in w_P. In order to preserve the accessibilities from w_B and w_P to current memory world w_s ($w_B \rightsquigarrow_e w_s$ and $w_P \rightsquigarrow_i w_s$), we need to show that any execution of M' can only access the private region in b'_g because it is the *module-local* private block allocated by M'. Although g can visit the stack-allocated data from f and h, the source program M can only access the public regions in b_f and b_h. Therefore, forward simulation ensures that M' does not change *local private data* in b'_f (which preserves $\rightsquigarrow_e$ from w_B) and *external private data* in b'_h (which preserves $\rightsquigarrow_i$ from w_P).

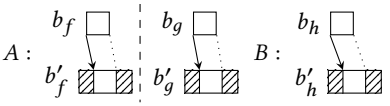


Fig. 15. Snapshot for `Stacking` pass

B.0.2 Verification of Optimization Passes using Static Analysis. The optimization passes `Constprop`, `CSE` and `Deadcode` (for constant propagation, common subexpression elimination and dead code elimination) share a static value analysis algorithm for collecting information of variables at runtime by abstract interpretation.

A simple example of constant propagation is depicted in Fig. 16. For the source program, the algorithm finds out that a has value 42 at line 3, the external function g does not change a at line 4 because a's address is never taken. Therefore it is reasonable to optimize the return value to 42 in the target program. From the perspective of verifier, we need to define a rely condition stating that a is not changed during g in order to establish the simulation. However, since the local variable a is not optimized into a register here, it is a *public* variable between source and target programs and *not protected* by `tinjp`.

```
1 void g();
2 int f() {
3   int a = 42;
4   g();
5   return a;
6 }
```

(a) Source Program

```
1 void g();
2 int f() {
3   int a = 42;
4   g();
5   return 42;
6 }
```

(b) Target Program

Fig. 16. An Example of Constant Propagation

The solution is to remove the block of a out of the domain of the injection when f calls g, therefore a could be protected by the guarantee condition of g. The invariant $(j, m_1, m_2) \in R(w_B, w_P)$ is defined such that the blocks for a is *public* in the internal program states (i.e. $b_a \in m_1, b'_a \in m_2$ and $j(b_a) = \lfloor (b'_a, 0) \rfloor$). But when f calls g, this mapping is removed from j, resulting to the memory world $w_A = (j', m_1, m_2)$ for queries. However, such construction is valid only if the mapping of a is not included in w_P to guarantee the increment of injection defined in $w_P \rightsquigarrow_i w_A$. We extended the invariant relating abstract interpretation and the dynamic semantics to ensure that if the current stack block is not leaked (e.g. b_a at line 4) then it is always *private* in the incoming memories in w_P.

It is worth noting that the reason we are able to remove the mapping for a without breaking the memory injection (i.e. $m_1 \hookrightarrow_m^j m_2 \Rightarrow m_1 \hookrightarrow_m^{j'} m_2$) is exactly the information *a is not leaked* provided by the static analysis. In other words, the definition of private and public regions via injection functions can *protect* not only the regions introduced by the change of memory structure, but also the regions introduced by static analysis.

Received 2024-11-15; accepted 2025-03-06