Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules

Verified compilation of open modules (i.e., modules whose functionality depends on other modules) provides foundation for end-to-end verification of modular programs ubiquitous in contemporary software. However, despite intensive investigation in this topic for decades, the proposed approaches are still difficult to use in practice as they rely on closed-world assumptions about the internal working of compilers, which not only incurs high complexity in verification but also makes it difficult for external users to apply the verification results. In this paper, we propose an approach to verified compositional compilation with no closed-world assumptions in the setting of verifying compilation of heterogeneous modules written in first-order languages supporting global memory and pointers. Our approach is based on a new discovery that a Kripke relation with a notion of memory protection can serve as a uniform and composable semantic interface for the compiler passes. By absorbing the rely-guarantee conditions on memory evolution for all compiler passes into this Kripke Memory Relation and by piggybacking requirements on compiler optimizations onto it, we get compositional correctness theorems for realistic optimizing compilers as refinements that directly relate physical semantics of open modules and that are ignorant of internal compilation processes. Such direct refinements support all the compositionality and adequacy properties essential for verified compilation of open modules. We demonstrate that our compiler correctness theorems are open to composition, intuitive and easy to use with reduced verification complexity through end-to-end verification of non-trivial heterogeneous modules that may freely invoke each other (e.g., mutually recursively). Our development is fully formalized in Coq based CompCertO and supports the full compilation chain of CompCert with its Clight source language.

1 INTRODUCTION

Verified compilation ensures that behaviors of source programs can be faithfully transported to executable target code, a property indispensable for end-to-end formal verification of software. As contemporary software is usually composed of program modules independently developed, compiled and linked, researchers have developed a wide range of techniques for *verified compositional compilation* or VCC [Gu et al. 2015; Jiang et al. 2019; Koenig and Shao 2021; Song et al. 2020; Stewart et al. 2015; Wang et al. 2019] that support modules mutually invoking each other (i.e., open), being written in different languages (i.e., heterogeneous) and transformed by different compilers. However, as it stands now, the proposed approaches are inherently flawed at supporting open modules (e.g. libraries) as they either deviate from the physical semantics of open modules or expose internal working of compilers, resulting in correctness theorems with closed-world assumptions that are difficult to work with and incur high cost in verification. In this paper, we investigate an approach to eliminating these flaws while retaining the full benefit of VCC, i.e., obtaining correctness of compiling open modules that is *fully composable*, *adequate*, and *truly open*.

1.1 Full Compositionality and Adequacy in Verified Compilation

Correctness of compiling open modules is usually described as refinement between semantics of source and target modules. We shall write L (possibly with subscripts) to denote semantics of open modules and write $L_1 \preccurlyeq L_2$ to denote that L_1 is refined by L_2. Therefore, the compilation of any module M_2 into M_1 is correct iff $[[M_1]] \preccurlyeq [[M_2]]$ where $[[M_i]]$ denotes the semantics of M_i.

To support the most general form of VCC, it is critical that the established refinements are *fully composable*, i.e., both *horizontally and vertically composable*, and *adequate for physical semantics*:

$$\begin{aligned}
\textbf{Vertical Compositionality:}\quad & L_1 \preccurlyeq L_2 \Rightarrow L_2 \preccurlyeq L_3 \Rightarrow L_1 \preccurlyeq L_3\\
\textbf{Horizontal Compositionality:}\quad & L_1 \preccurlyeq L_1' \Rightarrow L_2 \preccurlyeq L_2' \Rightarrow L_1 \oplus L_1' \preccurlyeq L_2 \oplus L_2'\\
\textbf{Adequacy for Physical Semantics:}\quad & [[M_1 + M_2]] \preccurlyeq [[M_1]] \oplus [[M_2]]
\end{aligned}$$

The first property states that refinement is transitive. It is essential for composing proofs for multi-pass compilers. The second property guarantees that refinement is preserved by semantic linking (denoted by $\oplus$). It is essential for composing correctness of compiling open modules (possibly through different compilers). The last one ensures that, given any modules, their semantic linking coincides with their syntactic linking (denoted by $+$). It ensures that linked semantics do not deviate from physical semantics and is essential to propagate verified properties to final target programs.

We use the example in Fig. 1 to illustrate the importance of the above properties in VCC where heterogeneous modules are compiled through vastly different compilation chains. Here, a source C module $\mathtt{a.c}$ is compiled to assembly $\mathtt{a.s}$ in three passes through two intermediate representations $\mathtt{a.i}_1$ and $\mathtt{a.i}_2$, and then linked with a library module $\mathtt{b.s}$ which is not compiled at all (an extreme case where the compilation chain is empty). The goal is to prove that the semantics of linked target assembly $\mathtt{a.s} + \mathtt{b.s}$ refines the combined source semantics $\llbracket \mathtt{a.c} \rrbracket \oplus L_b$ where L_b is the semantic specification of $\mathtt{b.s}$, i.e., $\llbracket \mathtt{a.s} + \mathtt{b.s} \rrbracket \preccurlyeq \llbracket \mathtt{a.c} \rrbracket \oplus L_b$. The proof proceeds as follows:

$$\begin{tikzcd}
\llbracket \mathtt{a.c} \rrbracket & \oplus & L_b \\
\llbracket \mathtt{a.i}_1 \rrbracket \arrow[u, phantom, "\rotatebox{90}{$\preccurlyeq$}"] & & \\
\llbracket \mathtt{a.i}_2 \rrbracket \arrow[u, phantom, "\rotatebox{90}{$\preccurlyeq$}"] & & \\
\llbracket \mathtt{a.s} \rrbracket \arrow[u, phantom, "\rotatebox{90}{$\preccurlyeq$}"] & \oplus & \llbracket \mathtt{b.s} \rrbracket \arrow[uuu, phantom, "\rotatebox{90}{$\preccurlyeq$}"] \\
& \llbracket \mathtt{a.s} + \mathtt{b.s} \rrbracket \arrow[u, phantom, "\rotatebox{90}{$\preccurlyeq$}"] &
\end{tikzcd}$$

Fig. 1. Motivating Example

(1) Prove every pass respects refinement, from which we get $\llbracket \mathtt{a.i}_1 \rrbracket \preccurlyeq \llbracket \mathtt{a.c} \rrbracket$, $\llbracket \mathtt{a.i}_2 \rrbracket \preccurlyeq \llbracket \mathtt{a.i}_1 \rrbracket$ and $\llbracket \mathtt{a.s} \rrbracket \preccurlyeq \llbracket \mathtt{a.i}_2 \rrbracket$. Furthermore, show $\mathtt{b.s}$ meets its specification, i.e., $\llbracket \mathtt{b.s} \rrbracket \preccurlyeq L_b$;
(2) By vertical compositionality, compose the refinement relations for compiling $\mathtt{a.c}$ to get $\llbracket \mathtt{a.s} \rrbracket \preccurlyeq \llbracket \mathtt{a.c} \rrbracket$;
(3) By horizontal compositionality, compose the above refinement with that for $\mathtt{b.t}$ to get $\llbracket \mathtt{a.s} \rrbracket \oplus \llbracket \mathtt{b.s} \rrbracket \preccurlyeq \llbracket \mathtt{a.c} \rrbracket \oplus L_b$;
(4) By adequacy of linking at the target level and a final vertical composition, conclude $\llbracket \mathtt{a.s} + \mathtt{b.s} \rrbracket \preccurlyeq \llbracket \mathtt{a.c} \rrbracket \oplus L_b$.

1.2 Problems with Existing Approaches to Refinements

Despite the simplicity of VCC at an intuitive level, full compositionality and adequacy are surprisingly difficult to prove for any non-trivial multi-pass compiler. First and foremost, the formal definitions must take into account the facts that each intermediate representation has different semantics and each pass may imply a different refinement relation. To facilitate the discussion below, we classify different open semantics by *languages interfaces* (or simply interfaces) which formalize their interaction with environments. We write $L : \mathcal{I}$ to denote that L has a language interface $\mathcal{I}$. For instance, $\llbracket \mathtt{a.c} \rrbracket : C$ denotes semantics of $\mathtt{a.c}$ with interface C for interaction with environment through function calls and returns in C. Similarly, $\llbracket \mathtt{a.s} \rrbracket : \mathcal{A}$ denotes semantics of $\mathtt{a.s}$ where $\mathcal{A}$ allows for interaction at the assembly level. Note that semantics may not have their native interfaces. $\llbracket \mathtt{a.s} \rrbracket : C$ asserts that $\llbracket \mathtt{a.s} \rrbracket$ actually converts assembly level calls/returns to C function calls/returns for interacting with C modules. In this case, the "wrapped" semantics *deviate from their physical semantics*, hence is not adequate. When the interface of semantics $\llbracket M \rrbracket$ for module M is not explicitly given, it either can be inferred from the context or is simply the native interface for the language of M. Refinements may relate source and target semantics with different interfaces. We write $\preccurlyeq : \mathcal{I}_1 \Leftrightarrow \mathcal{I}_2$ to denote that $\preccurlyeq$ is a refinement between two semantics with interfaces $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. For instance, $\llbracket \mathtt{b.s} \rrbracket \preccurlyeq_{\mathsf{ac}} L_b$ asserts that $\llbracket \mathtt{b.s} \rrbracket : \mathcal{A}$ is refined by $L_b : C$ with $\preccurlyeq_{\mathsf{ac}} : \mathcal{A} \Leftrightarrow C$ that relates open semantics at the C and assembly levels.

For VCC, it is essential that variance of open semantics and refinements does not impede compositionality and adequacy. The existing approaches achieve this by forcing all refinements to have certain coherent shapes, i.e., imposing *algebraic structures* on refinements. We categorize

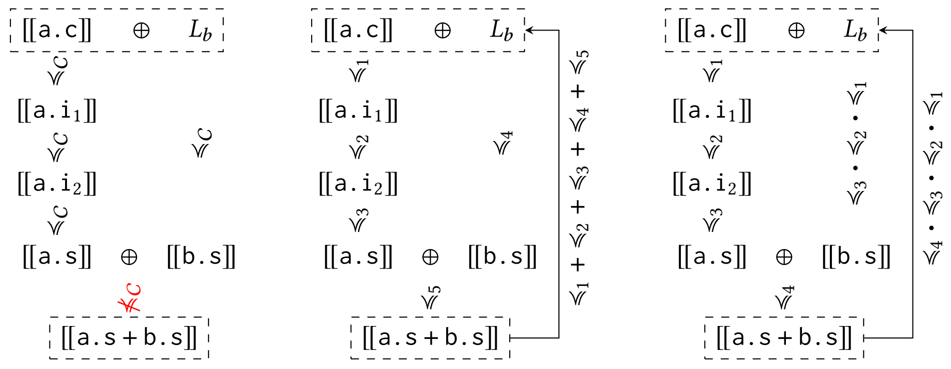

[[a.c]] ⊕ L_b
$\preccurlyeq_C$
[[a.i$_1$]]
[[a.i$_2$]]
[[a.s]] ⊕ [[b.s]]
$\not\preccurlyeq_C$
[[a.s + b.s]]
(a) CompComp
$\preccurlyeq_1$
$\preccurlyeq_2$
$\preccurlyeq_3$
$\preccurlyeq_4$
$\preccurlyeq_5$
$\preccurlyeq_1 + \preccurlyeq_2 + \preccurlyeq_3 + \preccurlyeq_4 + \preccurlyeq_5$
(b) CompCertM
$\preccurlyeq_1 \cdot \preccurlyeq_2 \cdot \preccurlyeq_3$
$\preccurlyeq_1 \cdot \preccurlyeq_2 \cdot \preccurlyeq_3 \cdot \preccurlyeq_4$
(c) CompCertO

Fig. 2. Refinements in Existing Approaches to VCC

these approaches by such algebraic structures below, and explain the problems facing them via three well-known extensions of CompCert [Leroy 2023] (the state-of-the-art verified compiler) to support VCC, i.e., Compositional CompCert (CompComp) [Stewart et al. 2015], CompCertM [Song et al. 2020] and CompCertO [Koenig and Shao 2021].

Constant Refinement. An obvious way to account for different semantics in VCC is to force every semantics to use the same language interface $\mathcal{I}$ and use a constant refinement relation $\preccurlyeq_{\mathcal{I}}: \mathcal{I} \Leftrightarrow \mathcal{I}$. CompComp adopts this "one-type-fits-all" approach by having every language of CompCert to use C function calls/returns for module-level interactions and using a uniform refinement relation $\preccurlyeq_C: C \Leftrightarrow C$ known as *structured simulation* [Stewart et al. 2015]. In this case, vertical and horizontal compositionality is established by proving transitivity of $\preccurlyeq_C$ and symmetry of rely-guarantee conditions of $\preccurlyeq_C$. However, the verification results severely deviate from physical semantics since not all languages conform to the C interface. In particular, the adequacy at the target level is lost, making end-to-end compiler correctness not provable as shown in Fig. 2a.

Sum of Refinements. A more relaxed approach allows users to choose language interfaces for different IRs from a finite collection $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ and to choose refinement relations for different passes from a finite set $\{\preccurlyeq_1, \ldots, \preccurlyeq_n\}$ relating these interfaces, in which $\preccurlyeq_i: \mathcal{I}_1 + \ldots + \mathcal{I}_m \Leftrightarrow \mathcal{I}_1 + \ldots + \mathcal{I}_m$. In essence, a constant refinement is split into a sum of refinements s.t. $L \preccurlyeq_1 + \ldots + \preccurlyeq_n L'$ holds if $L \preccurlyeq_i L'$ for some $1 \leq i \leq n$. Then, every compiler pass can use $\preccurlyeq_1 + \ldots + \preccurlyeq_n$ as the uniform refinement relation, which is proven both composable and adequate under certain well-formedness constraints. Fig. 2b depicts an example where semantics have both C and assembly interfaces, e.g., [[a.s]] : $\mathcal{A} + C$. This is the approach adopted by CompCertM to recover adequacy [Song et al. 2020]. However, this approach has several disadvantages. First, one must know beforehand that all the refinements in compilation are included in $\{\preccurlyeq_1, \ldots, \preccurlyeq_n\}$, which is essentially a *closed-world* assumption. Second, horizontal composition only works for modules *self-related* by all the refinements $\preccurlyeq_i$ $(1 \leq i \leq n)$, which is another closed-world restriction. Finally, since $\preccurlyeq_i$s are summarized from intermediate compiler passes, the top-level refinement inevitably depends on internals of compilers which hinders further compositional verification.

Product of Refinements. The previous approach effectively "flattens" the refinements for individual compiler passes into an end-to-end refinement. A different approach adopted by CompCertO [Koenig and Shao 2021] is to "concatenate" the refinements for individual passes into a chain of refinements by a product operation (_ • _). In particular, $L \preccurlyeq_1 \cdot \preccurlyeq_2 L''$ if there is some L' s.t. $L \preccurlyeq_1 L'$ and $L' \preccurlyeq_2 L''$. With the freedom to choose language interfaces, adequacy is still guaranteed.

With the bottom-up construction of refinements, no a priori knowledge about the whole compilation is needed for compositionality. Fig. 2c illustrates how it works. However, dependency on internals of compilation still exists. Note that vertical composition of refinements for compiling `a.c` results in a product refinement $[\![\texttt{a.s}]\!] \preccurlyeq_3 \cdot \preccurlyeq_2 \cdot \preccurlyeq_1 [\![\texttt{a.c}]\!]$. To horizontally compose with it, it is necessary to show L_b refines $[\![\texttt{b.s}]\!]$ via the same product, i.e., to construct intermediate semantics bridging $\preccurlyeq_1$, $\preccurlyeq_2$ and $\preccurlyeq_3$. Therefore, even though this approach allows more flexible vertical composition, it is still dependent on the details of compilation for horizontal composition.

In summary, the existing approaches for VCC either lack adequacy because of deviation from physical semantics or lack compositionality that is truly extensional because of their dependency on internals of compilation. Such dependency makes the obtained correctness theorems for compiling open modules (e.g., libraries) difficult to further compose with. It also incurs high cost in verification.

1.3 Challenges for Direct Refinement of Open Modules

The ideal approach to VCC should produce refinements between source and target open modules that *1)* relate their semantics at their native language interfaces for adequacy, *2)* support vertical and horizontal composition without assuming internal details of compilation for truly open compositionality. For example, the refinement between `a.c` and `a.s` could be $\preccurlyeq_{\mathrm{ac}}: \mathcal{A} \Leftrightarrow C$ s.t. $[\![\texttt{a.s}]\!] \preccurlyeq_{\mathrm{ac}} [\![\texttt{a.c}]\!]$. It directly relates assembly and C open semantics, and could be further horizontally composed with $[\![\texttt{b.s}]\!] \preccurlyeq_{\mathrm{ac}} L_b$ and vertically composed by adequacy to get $[\![\texttt{a.s} + \texttt{b.s}]\!] \preccurlyeq_{\mathrm{ac}} [\![\texttt{a.c}]\!] \oplus L_b$. Such refinements are easy to work with because of their extensionality, and can always be composed further with other refinements when necessary, thereby effectively support VCC for open programs and libraries (note that even the top-level refinement is still open to horizontal and vertical composition). We shall call them *direct refinements of open modules*.

The main challenge in getting direct refinements is tied to the well-known problem of proving "real" vertical composition for open refinements where the composed refinement must directly relates source and target semantics. That is, given any $\preccurlyeq_1 \cdot \preccurlyeq_2$, how to show it is equivalent to a single direct refinement $\preccurlyeq_3$. This is considered very technical and involved (see [Chung-Kil Hur and Vafeiadis 2012; Neis et al. 2015; Patterson and Ahmed 2019; Song et al. 2020]) because of the difficulty in constructing *interpolating* program states for transitively relating evolving source and target states across *external calls* of open modules. This problem also manifests in proving transitivity for *logical relations* where construction of interpolating terms of higher-order types is not in general possible [Ahmed 2006]. In the setting of compiling first-order languages with memory states, all previous work avoids proving real vertical compositionality directly. Some introduces intrusive changes to deal with complex rely-guarantee conditions on interleaving module states, which either destroy adequacy or weaken compositionality [Song et al. 2020; Stewart et al. 2015]. Indeed, CompComp instruments the semantics of languages with *effect annotations* to expose internal effects and to make the construction of interpolating states feasible; CompCertM partially avoids the problem when restricting vertical composition to *closed* programs; CompCertO provides ad-hoc merging of refinements that still exposes the intermediate states of compilation.

Even if the problem of real vertical composition was solved, it is not clear if the solution can scale to realistic compilers with complex optimizations which have additional assumptions on programs (e.g. control or data flow analysis). Furthermore, to show that VCC with direct refinements are indeed useful in practice, it needs to support heterogeneous modules with free interaction between them and, even more, end-to-end and compositional program verification.

1.4 Our Contributions

In this paper, we are concerned with VCC of imperative programs with first-order states and support of pointers, e.g., C and assembly programs. In this setting, we address all of the above challenges.

First, we propose an approach to proving real vertical composition and to direct refinements of open modules with full support of compositionality and adequacy. Second, we demonstrate its effectiveness in verification of the *full* compilation chain of CompCert starting from its Clight source language. Third, we show the same approach easily supports end-to-end refinement verification.

A critical observation we make is that interpolating states for proving vertical compositionality of refinements can be constructed by exploiting properties on memory injections in CompCert. With it we discover that a *Kripke relation with memory protection* can serve as a uniform and composable interface for characterizing evolution of memory states across external calls. By adopting this relation in the refinement of CompCert's passes, we successfully combined their refinement proofs into a direct refinement from C modules to assembly modules, which does not have the weaknesses of existing approaches described in Sec. 1.2. We summarize our technical contributions below:

- We prove that `injp`—a Kripke Memory Relation with a notion of protection on evolving memory states across external calls—is both uniform (i.e., memory transformation in every compiler pass respects this relation) and composable (i.e., transitive modulo an equivalence relation). The critical observation making this proof possible is that interpolating memory states can be constructed by exploiting memory protection *inherent* to memory injections and the *functional* nature of injections.
- Based on the above observation, we show that a direct refinement from C to assembly can be derived by composing open refinements for all of CompCert's passes starting from Clight. In particular, we show that compiler passes can use different Kripke relations sufficient for their proofs (which may be weaker than `injp`) and these relations will later be absorbed into `injp` via refinements of open semantics. Furthermore, we show that assumptions for compiler optimizations can be formalized as *semantic invariants* and, when piggybacked onto `injp`, can be transitively composed. Based on these techniques, we upgrade the proofs in CompCertO to get a direct refinement from C to assembly for the full CompCert, including all of its optimization passes. These experiments show that direct refinements can be achieved without fundamental changes to the verification framework of CompCert.
- We demonstrate the simplicity and usefulness of direct refinements by applying it to end-to-end verification of several non-trivial examples with heterogeneous modules that *mutually* invoke each other. In particular, we observe that C level refinements can be absorbed into the direct refinement of CompCert by compositionality of `injp`. Combining with full compositionality and adequacy, we derive end-to-end refinements from high-level source specifications to physical semantics of linked assembly modules in a straightforward manner.

The above developments are fully formalized in Coq based on the latest release of CompCertO [Koenig and Shao 2021] which is in turn based on CompCert v.3.10 (See the supplementary materials). We choose CompCert because it is the most realistic and sophisticated verified compiler with non-trivial memory model supporting pointers and complex optimizations relying on memory invariants. Our development demonstrates that realistic verified compilers supporting the complete features of VCC and direct refinements of open modules are achievable without fundamental changes to verification frameworks and with reasonable effort. Therefore, our approach provides a promising direction for further evolving the techniques for compositional compiler verification.

1.5 Structure of the Paper

In the rest of the paper, we first talk about the background and key ideas of our approach in Sec. 2. We then present our technical contributions in Sec. 3, Sec. 4 and Sec. 5. We discuss evaluation and related work in Sec. 6 and finally conclude in Sec. 7.

```c
/* client.c */
int result;

void encrypt(int i,
     void(*p)(int*));

void process(int *r) {
  result = *r;
}

int request(int i) {
  encrypt(i,process);
  return i;
}
```

(a) Client in C

```
/* server.s */
key:
  .long 42
encrypt:
  // allocate frame
  Pallocframe 24 16 0
  // RSP[8] = i XOR key
  Pmov key RAX
  Pxor RAX RDI
  Pmov RDI 8(RSP)
  // call p(RSP + 8)
  Plea 8(RSP) RDI
  Pcall RSI
  // free frame
  Pfreeframe 24 16 0
  Pret
```

(b) Server in Asm

```
/* server_opt.s
 * key is an constant
 * and inlined in code */
encrypt:
  // allocate frame
  Pallocframe 24 16 0
  // RSP[8] = i XOR 42
  Pxori 42 RDI

  Pmov RDI 8(RSP)
  // call p(RSP + 8)
  Plea 8(RSP) RDI
  Pcall RSI
  // free frame
  Pfreeframe 24 16 0
  Pret
```

(c) Optimized Server

Fig. 3. An Example of Encryption Client and Server

2 KEY IDEAS

We introduce a running example with heterogeneous modules and callback functions to illustrate the usefulness of direct refinements. This example is representative of mutual dependency between modules that often appears in practice and it shows how free-form invocation between modules can be supported by our approach. As we shall see later, our approach also handles more complicated programs with mutually *recursive* heterogeneity without any problem (discussed in Sec. 5).

The example is given in Fig. 3. It consists of a client written in C (Fig. 3a) and an encryption server hand-written in x86 assembly by using CompCert's assembly syntax where instruction names begin with `P` (Fig. 3b). For now, let us ignore Fig. 3c which illustrates how optimizations work in direct refinements. Users invoke `request` to initialize an encryption request. It is relayed to the function `encrypt` in the server with the prototype `void encrypt(int i, void (*p)(int*))` which respects a calling convention placing the first and second arguments in registers `RDI` and `RSI`, respectively. The main job of the server is to encrypt `i` (`RDI`) by XORing it with an encryption key (stored in the global variable `key`) and invoke the callback function `p` (`RSI`). Finally, the client takes over and stores the encrypted value in the global variable `result`. The pseudo instruction `Pallocframe m n o` allocates a stack frame of `m` bytes and stored its address in register `RSP`. In this frame, a pointer to the caller's stack frame is stored at the `o`-th byte and the return address is stored at the `n`-the byte. Note that `Pallocframe 24 16 0` in `encrypt` reserves 8 bytes on the stack from `RSP + 8` to `RSP + 16` for storing the encrypted value, whose address is passed as an argument to the callback function `p`. `Pfreeframe m n o` frees the frame and restores `RSP` and `RA`.

With the running example, our goal is to verify its end-to-end correctness by exploiting the direct refinement $\preccurlyeq_{\mathrm{ac}}: \mathcal{A} \Leftrightarrow C$ derived from CompCert's compilation chain as shown in Fig. 4. The verification proceeds as follows. First, we establish $[\![\texttt{client.s}]\!] \preccurlyeq_{\mathrm{ac}} [\![\texttt{client.c}]\!]$ by the correctness of compilation. Then, we prove $[\![\texttt{server.s}]\!] \preccurlyeq_{\mathrm{ac}} L_{\mathrm{S}}$ manually by providing a specification L_{S} for the server that respects the direct refinement. At the source level, the combined semantics is further refined to a single top-level specification L_{CS}. Finally, the source and target level refinements are absorbed into the direct refinement by vertical composition and adequacy, resulting in a *single direct refinement* between the top-level specification and the target program:

$$[\![\texttt{client.s} + \texttt{server.s}]\!] \preccurlyeq_{\mathrm{ac}} L_{\mathrm{CS}}$$

The above verification is feasible thanks to the following benefits of direct refinements. First and foremost, direct refinements are truly extensional as they are ignorant of how modules are complied (if they are compiled at all), thereby can be easily adopted to handle free-form heterogeneous modules such as hand-written assembly with callback. Second, they respects adequacy, thereby can relate physical semantics of linked modules. Third, they always relate open modules and support full compositionality, thereby can be further composed with other refinement relations when necessary (e.g. the C level refinement to L_{CS}). This is fundamentally different from the closed nature of existing approaches described in Sec. 1.2.

$$\begin{tikzcd}
 & \boxed{L_{CS}} \arrow[d, phantom, "\curlyvee"] & & \\
{[\![\texttt{client.c}]\!]} \arrow[d, dashed, "\text{CompCert}"', "\curlyvee_{ac}"] & \oplus & L_S \arrow[d, phantom, "\curlyvee_{ac}"] & \\
{[\![\texttt{client.s}]\!]} & \oplus \arrow[d, phantom, "\curlyvee"] & {[\![\texttt{server.s}]\!]} & \\
 & \boxed{[\![\texttt{client.s} + \texttt{server.s}]\!]} \arrow[uuu, bend right, "\curlyvee_{ac}"'] & &
\end{tikzcd}$$

Fig. 4. Verifying the Running Example

In the subsections below, we give an in-depth exposure of the advantages of direct refinements by elaborating on how they directly relate source and target semantics while ignoring internals of compilation, and our approach to obtaining direct refinements.

2.1 Background

We begin by introducing necessary background, including the memory model, the framework for simulation-based refinement, and `injp` which is critical for direct refinements.

2.1.1 Block-based Memory Model. In CompCert's memory model [Leroy et al. 2012], a memory state m (of type `mem`) consists of a disjoint set of *memory blocks* with unique identifiers and linear address space. A memory address or pointer (b, o) points to the o-th byte in the block b where b has type `block` and o has type `Z` (integers). The value of memory cell (one byte) at (b, o) is denoted by $m[b, o]$. Values are either undefined, 32- or 64-bit integers or floats, or pointers and defined by $\texttt{val} := \texttt{Vundef} \mid i_{32} \mid i_{64} \mid f_{32} \mid f_{64} \mid \texttt{Vptr}(b, o)$. For simplicity, we often write b for $\texttt{Vptr}(b, 0)$. The memory operations including allocation, free, read and write are provided and governed by permissions of cells. A memory cell have the following permissions ordered from high to low: $\texttt{Freeable} \geqslant \texttt{Writable} \geqslant \texttt{Readable} \geqslant \texttt{Nonempty}$ where `Freeable` enables all operations, `Writable` enables all but free, `Readable` enables only read, and `Nonempty` enables none. If $p_1 \geqslant p_2$ then any cell with permission p_1 also implicitly has permission p_2. $\texttt{perm}(m, P)$ denotes the set of memory cells with at least permission P. For example, $(b, o) \in \texttt{perm}(m, \texttt{Readable})$ iff the cell at (b, o) in m is readable. An address with no permission at all is not in the footprint of memory.

Transformations of memory states are captured via partial functions $j : \texttt{block} \to \lfloor \texttt{block} \times \texttt{Z} \rfloor$ called *injection functions*, s.t. $j(b) = \emptyset$ if b is removed from memory and $j(b) = \lfloor (b', o) \rfloor$ if b is shifted (injected) to (b', o) in the target memory. We define $\texttt{meminj} = \texttt{block} \to \lfloor \texttt{block} \times \texttt{Z} \rfloor$. v_1 and v_2 are related under j (denoted by $v_1 \hookrightarrow_v^j v_2$) if either v_1 is `Vundef`, or they are both equal scalar values, or pointer shifted according to j, i.e., $v_1 = \texttt{Vptr}(b, o)$, $j(b) = \lfloor (b', o') \rfloor$ and $v_2 = \texttt{Vptr}(b', o + o')$.

Given this relation, a *memory injection* between the source memory state m_1 and the target state m_2 under j (denoted by $m_1 \hookrightarrow_m^j m_2$) if the following properties are satisfied which ensure preservation of permissions and values under injection:

$$\forall\, b_1\ b_2\ o\ o'\ p,\ \ j(b_1) = \lfloor (b_2, o') \rfloor \Rightarrow (b_1, o) \in \texttt{perm}(m_1, p) \Rightarrow (b_2, o + o') \in \texttt{perm}(m_2, p)$$
$$\forall\, b_1\ b_2\ o\ o',\ \ j(b_1) = \lfloor (b_2, o') \rfloor \Rightarrow (b_1, o) \in \texttt{perm}(m_1, \texttt{Readable}) \Rightarrow m_1[b_1, o] \hookrightarrow_v^j m_2[b_2, o + o']$$

Memory injections are necessary for verifying compiler transformations of memory structures (e.g., merging local variables into stack-allocated data and generating a concrete stack frame). For the remaining passes, a simpler relation called *memory extension* is used instead, which employs

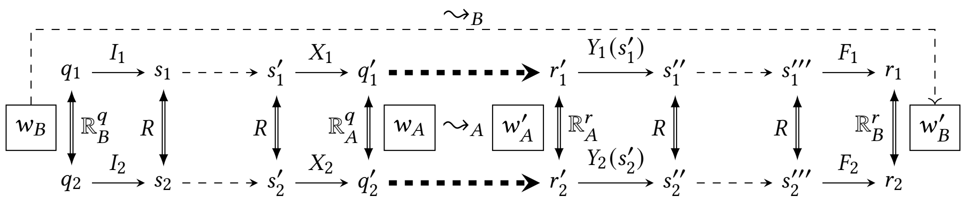


Fig. 5. Open Simulation between LTS

an identity injection function. As we shall see, reasoning about permissions and states under refinements is a major source of complexity in our work.

2.1.2 A Framework for Open Simulations. We adopt a language-independent framework for open semantics and simulations with customizable language interfaces for adequately describing language semantics at different levels, as introduced in CompCertO [Koenig and Shao 2021].

A *language interface* $A = \langle A^q, A^r \rangle$ is a pair of sets A^q and A^r denoting acceptable queries and replies for open modules, respectively. Different language interfaces may be used at different stages of compilation. The relevant ones for our discussion are shown as follows (we omit language interfaces for intermediate languages in CompCert):

Languages	**Interfaces**	**Queries**	**Replies**
C/Clight	$C = \langle \mathtt{val} \times \mathtt{sig} \times \mathtt{val}^* \times \mathtt{mem}, \mathtt{val} \times \mathtt{mem} \rangle$	$v_f[sg](\vec{v})@m$	$v'@m'$
Asm	$\mathcal{A} = \langle \mathtt{regset} \times \mathtt{mem}, \mathtt{regset} \times \mathtt{mem} \rangle$	$rs@m$	$rs'@m'$

Here, the C interface is used for front-end languages where $v_f[sg](\vec{v})@m$ is a function call to v_f with signature sg and with a list of arguments $\vec{v}$ and a memory state m and $v'@m'$ carries a return value v' and an updated memory state m'. The assembly interface supports queries and replies carrying pairs of register sets (denoted by rs) and memory as program states.

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 (B) is the language interface for outgoing (incoming) queries and replies, $D \subseteq B^q$ a set of initial queries, S a set of internal states, $I \subseteq D \times S$ ($F \subseteq S \times B^r$) transition relations for incoming queries (replies), $X \subseteq S \times A^q$ ($Y \subseteq S \times A^r \times S$) transitions for outgoing queries (replies), and $\rightarrow \subseteq S \times E^* \times S$ internal transitions emitting events of type E. Note that $(s, q^O) \in X$ iff an outgoing query q^O happens at s; $(s, r^O, s') \in Y$ iff after q^O returns with r^O the execution continues with an updated state s'.

Kripke relations are used to describe evolution of program states in open simulations between LTS. 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 $\mathcal{K}_W(A, B) = W \rightarrow \{S \mid S \subseteq A \times B\}$. A *simulation convention* relating two language interfaces A_1 and A_2 is a tuple $\mathbb{R} = \langle W, \mathbb{R}^q : \mathcal{K}_W(A_1^q, A_2^q), \mathbb{R}^r : \mathcal{K}_W(A_1^r, A_2^r) \rangle$ with type $\mathbb{R} : A_1 \Leftrightarrow A_2$. Simulation conventions serve as interfaces of open simulations by relating source and target language interfaces. For example, the basic C-level convention $\mathtt{c} : C \Leftrightarrow C = \langle \mathtt{meminj}, \mathbb{R}_\mathtt{c}^q, \mathbb{R}_\mathtt{c}^r \rangle$ relates C queries and replies as follows, where the Kripke world consists of injections and, in a given world j, the values and memory in queries and replies are related by j.

$$
\begin{aligned}
(v_f[sg](\vec{v})@m, v'_f[sg](\vec{v'})@m') \in \mathbb{R}_\mathtt{c}^q(j) \quad &\Leftrightarrow \quad v_f \hookrightarrow_v^j v'_f \land \vec{v} \hookrightarrow_v^j \vec{v'} \land m \hookrightarrow_m^j m' \\
(v@m, v'@m') \in \mathbb{R}_\mathtt{c}^r(j) \quad &\Leftrightarrow \quad v \hookrightarrow_v^j v' \land m \hookrightarrow_m^j m'
\end{aligned}
$$

Open simulations describe refinement between LTS. To establish an open (forward) simulation between $L_1 : A_1 \twoheadrightarrow B_1$ and $L_2 : A_2 \twoheadrightarrow B_2$, one needs to find two simulation conventions $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$ that connect queries and replies at the outgoing and incoming sides, and show the internal execution steps and external interactions of open modules are related by an

Kripke invariant R, as shown in Fig. 5. We shall write such a simulation as $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ and, for simplicity, write $L_1 \leqslant_{\mathbb{R}} L_2$ to denote $L_1 \leqslant_{\mathbb{R} \twoheadrightarrow \mathbb{R}} L_2$.

The Kripke worlds (e.g., memory injections) may evolve as the execution goes on. *Rely-guarantee* reasoning about such evolution is essential for horizontal composition of simulations in presence of mutual calls between open modules. For illustration, the Kripke worlds at the boundary of modules is displayed in Fig. 5. The evolution of worlds across external calls is governed by an *accessibility relation* $\leadsto_A$ used to describe the *rely-condition* $w_A \leadsto_A w'_A$. By assuming external calls meet the rely-condition, one needs to prove the *guarantee condition* $w_B \leadsto_B w'_B$, i.e., the whole execution results in evolution of worlds respecting $\leadsto_B$. As a result, simulations with symmetric rely-guarantee conditions can be horizontally composed, even with mutual calls between modules.

Note that the accessibility relation and evolution of worlds between queries and replies is not encoded explicitly in the definition of simulation conventions. Instead, they are implicit by assuming a modality operator $\Diamond$ is always applied to $\mathbb{R}^r$ s.t. $r \in \Diamond\mathbb{R}^r(w) \Leftrightarrow \exists\, w', w \leadsto w' \wedge r \in \mathbb{R}^r(w')$. For simplicity, we often ignore accessibility and modality when talking *purely* about properties of simulations conventions in the remaining discussion.

Accessibility relations are mainly for describing evolution of memory states across external calls. For this, simulation conventions are parameterized by *Kripke Memory Relations* or KMR. We often write $\mathbb{R}_K$ to emphasize a convention $\mathbb{R}$ is parameterized by the KMR K.

Definition 2.1. A Kripke Memory Relation is a tuple $\langle W, f, \leadsto, R\rangle$ where W is a set of worlds, $f : W \rightarrow \texttt{meminj}$ a function for extracting injections from worlds, $\leadsto \subseteq W \times W$ an accessibility relation between worlds and $R : \mathcal{K}_W(\texttt{mem}, \texttt{mem})$ a Kripke relation over memory states that is compatible with the memory operations. We write $w \leadsto w'$ for $(w, w') \in \leadsto$.

The most interesting KMR is `injp` as it provides protection on memory w.r.t. injections.

Definition 2.2 (Kripke Relation with Memory Protection). $\texttt{injp} = \langle W_{\texttt{injp}}, f_{\texttt{injp}}, \leadsto_{\texttt{injp}}, R_{\texttt{injp}}\rangle$ where $W_{\texttt{injp}} = (\texttt{meminj} \times \texttt{mem} \times \texttt{mem})$, $f_{\texttt{injp}}(j, \_, \_) = j$, $(m_1, m_2) \in R_{\texttt{injp}}(j, m_1, m_2) \Leftrightarrow m_1 \hookrightarrow^j_m m_2$ and

$$\begin{aligned}(j, m_1, m_2) \leadsto_{\texttt{injp}} (j', m'_1, m'_2) \;\Leftrightarrow\; & j \subseteq j' \wedge \texttt{unmapped}(j) \subseteq \texttt{unchanged-on}(m_1, m'_1)\\ & \wedge \texttt{out-of-reach}(j, m_1) \subseteq \texttt{unchanged-on}(m_2, m'_2).\\ & \wedge \texttt{mem-acc}(m_1, m'_1) \wedge \texttt{mem-acc}(m_2, m'_2)\end{aligned}$$

Here, $\texttt{mem-acc}(m, m')$ denotes monotonicity of memory states such as valid blocks can only increase and read-only data does not change in value. $\texttt{unchanged-on}(m, m')$ denotes memory cells whose permissions and values are not changed from m to m' and

$$\begin{aligned}(b_1, o_1) \in \texttt{unmapped}(j) \;&\Leftrightarrow\; j(b_1) = \emptyset\\ (b_2, o_2) \in \texttt{out-of-reach}(j, m_1) \;&\Leftrightarrow\; \forall\, b_1\, o'_2,\; j(b_1) = \lfloor(b_2, o'_2)\rfloor \Rightarrow (b_1, o_2 - o'_2) \notin \texttt{perm}(m_1, \texttt{Nonempty}).\end{aligned}$$

Intuitively, a world (j, m_1, m_2) evolves to (j', m'_1, m'_2) under `injp` only if j' is strictly larger than j and any memory cells in m_1 and m_2 not in the domain (i.e., unmapped by j), or image of j (i.e., out of reach by j from m_1) will be protected, meaning their values and permissions are unchanged from m_1 (m_2) to m'_1 (m'_2). An example is shown in Fig. 6 where the shaded regions in m_1 are unmapped by j and unchanged while those in m_2 are out-of-reach from j and unchanged. m'_1 and m'_2 may contain newly allocated blocks; those blocks are never protected by `injp`. When `injp` is used at the outgoing side, it denotes that the simulation relies on knowing that the unmapped and out-of-reach regions at the call side are not modified across external calls. When `injp` is used at incoming side, it denotes that the simulation guarantees the unmapped and out-of-reach regions at initial queries are not modified by the simulation itself.

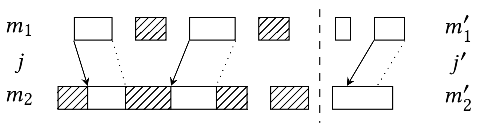


Fig. 6. Kripke Worlds Related by injp

$$\begin{tikzcd}
L_S : & \mathtt{encrypt}(i,p)@m_1 \arrow[r, "I_1"] \arrow[dd, leftrightarrow, "\mathbb{C}^q"] & ([i,p], m_1) \arrow[r, "K_1"] \arrow[dd, leftrightarrow, "R"'] & ([i,p], m_1') \arrow[r, "X_1"] \arrow[dd, leftrightarrow, "R"'] & p(b_0)@m_1' \arrow[rr, dashed, Rightarrow] \arrow[dd, leftrightarrow, "\mathbb{C}^q"'] & & ()@m_1'' \arrow[dd, leftrightarrow, "\mathbb{C}^r"] & \cdots \\
\rotatebox{90}{$\leqslant_{\mathbb{C}}$} & & & & & \boxed{w} \arrow[r, squiggly, "\mathtt{injp}"] & \boxed{w'} & \\
{[[\mathtt{server.s}]]}: & rs@m_2 \arrow[r, "I_2"] & rs@m_2 \arrow[r, dashed, "K_2"] & rs'@m_2' \arrow[r, "X_2"] & rs'@m_2' \arrow[rr, dashed, Rightarrow] & & rs''@m_2'' & \cdots
\end{tikzcd}$$

Fig. 7. Direct Refinement of the Hand-written Server

2.2 Direct Refinement of Open Modules for CompCert

We now define $L_1 \leqslant_{\mathsf{ac}} L_2$ as $L_2 \leqslant_{\mathbb{C}} L_1$ where the simulation convention $\mathbb{C} : C \Leftrightarrow \mathcal{A}$ is parameterized by `injp` and relates open semantics of C and assembly.[1] Given $q_C = v_f[sg](\vec{v})@m_1$, $q_{\mathcal{A}} = rs@m_2$, $r_C = res@m_1'$, and $r_{\mathcal{A}} = rs'@m_2'$. $\mathbb{C}^q(q_C, q_{\mathcal{A}})$ requires that

(1) The memory states are related by an injection function j, i.e., $m_1 \hookrightarrow_m^j m_2$;
(2) The function pointer in C query is related to the PC register, i.e., $v_f \hookrightarrow_v^j rs(\mathsf{PC})$;
(3) The arguments in C query $\vec{v}$ are projected either to registers or to outgoing argument slots in the stack frame RSP according to the C calling convention;
(4) The outgoing arguments region in target stack frame is *freeable* and not in the image of j.

The first three requirements ensures that C arguments and memory are related to assembly registers and memory according to CompCert's C calling convention. The last one ensures outgoing arguments are protected, thereby preserving the invariant of open simulation across external calls.

For replies, $\mathbb{C}^r(r_C, r_{\mathcal{A}})$ requires that

(1) The updated memory states are related by $m_1' \hookrightarrow_m^{j'} m_2'$ where $j \subseteq j'$ and protected by `injp`, i.e., $(j, m_1, m_2) \leadsto_{\mathtt{injp}} (j', m_1', m_2')$;
(2) The C-level return value *res* is related to the value stored in the register for return value;
(3) For any callee-saved register r, $rs'(r) = rs(r)$;
(4) $rs'(\mathsf{RSP}) = rs(\mathsf{RSP})$, $rs'(\mathsf{PC}) = rs(\mathsf{RA})$.

The first two requirements ensure that return values and memories are related according to the calling convention and the private regions are protected. The last two ensure that registers are correctly restored before returning. Finally, the direct refinement contains additional *semantic invariants* at the C level to ensure that read-only global variables are never modified and well-typedness of function calls and returns. They will be further discussed in Sec. 4.

2.3 Effectiveness of the Direct Refinement

We now illustrate how the above simple definition works for VCC of heterogeneous modules, including how it directly relates source and target semantics, how it ensures simulation by exploiting memory protection provided by `injp`, and how it handles compiler optimizations. The discussion is informal here; a formal development will be presented in Sec. 5.

The first thing to notice is that the definition of direct refinements does not mention anything about compilation: it is basically a formalized C calling convention with direct relations between source and target program states described via injections, together with invariants for protecting register values and memory states across external calls. Therefore, the definition is extensional and works for arbitrary code that meets its requirements. Take the refinement of hand-written assembly in Fig. 4 as an example, we need to prove $L_S \leqslant_{\mathbb{C}}$ [[`server.s`]] by hand. The first few steps of the proof are depicted in Fig. 7, where L_S is an LTS hand-written by users and [[`server.s`]] is derived from CompCert assembly semantics. Because we are free to design L_S as long as it respects

[1]Note that the direction of refinement is reverted because of forward simulation, which is common in work based on CompCert as forward simulations are easier to prove and can be flipped into backward simulations [Sevcík et al. 2013].

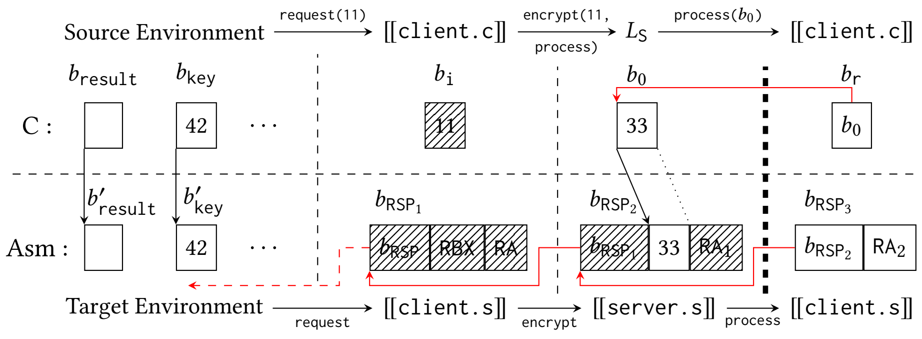


Fig. 8. Snapshot of the Memory State after Call Back

the direct refinement, we choose a form easy to comprehend where the internal executions are all in big steps. Now, suppose the environment calls `encrypt`, the source and target queries are initially related by $\mathbb{C}^q$ s.t. $rs(\texttt{RDI}) = i$ and $rs(\texttt{RSI}) = p$ by the calling convention. Then, according to I_1 and I_2, execution enters internal states related by an invariant R. Now, target execution takes internal steps until it hits an external call. It corresponds to executing lines 5-13 in Fig. 3b, which allocates the stack frame RSP, performs encryption by storing `i XOR key` at the address RSP+8, and invokes the callback function p with RSP+8. This corresponds to one big-step execution K_1 at the source level which allocates a memory block b_0, stored `i XOR key` at b_0, and preparing to call p with b_0. Therefore, the injection in the invariant R maps b_0 to RSP+8. The execution continues with external call to p, returns from p and goes on until `encrypt` returns.

Another point to note is that the requirement $(j, m_1, m_2) \leadsto_{\texttt{injp}} (j', m'_1, m'_2)$ between calls and returns is essential for protecting private values in stack memory so that simulation is preserved across external calls. This is true for any semantics related by the direct refinement, even for hand-written assembly. We illustrate how it works with an example. The environment calls `request` in the client with 11 which in turn calls `encrypt` in the server to generate an encrypted value 11 XOR 42 = 33 , whose address is passed back to the client by calling `process`. Fig. 8 depicts the snapshot of the memory states and their injection relation right after the call back function is invoked (i.e., at line 8 in Fig. 3a), where black arrows between blocks denote injections; red arrows denote pointers. The source semantics allocates one block for each local variable (b_i for `i`, b_0 for the encrypted value and b_r for `r`) while the target semantics stores their values in registers or stacks (11 is stored in RDI while the encrypted value on the stack because its address is taken and may be modified by the callee). One stack frame is allocated for each function call which also stores private values including pointers to previous frames ($b_{\texttt{RSP}}$), return addresses (RA), and callee-saved registers (e.g., RBX). We use j, m_1 and m_2 to denote the injection, source and target memory states right at the call to `process` (they are to the left of the thick dashed line in Fig. 8), and j', m'_1, m'_2 to denote the injection and memory states after `process` returns. Then, $(j, m_1, m_2) \leadsto_{\texttt{injp}} (j', m'_1, m'_2)$ ensures that all the shaded memory areas are never modified between the call and return of `process` because they are out-of-reach from j. Only the stack-allocated encrypted value can be modified because the source block is also modifiable. Therefore, the return address or stack pointer of `encrypt` is never modified by `process`, thereby guaranteeing the simulation invariant to continue to hold after `process` returns. A similar situation can be observed at the call site to `encrypt` in the client, where even the source block b_i is protected because it is unmapped by injection. Note that this intuitive and direct protection of memory through `injp` is not present in previous work. CompCertO does not have a direct relation between source and target memory [Koenig and Shao 2021]. Therefore, the memory protection must be broken down into a sequence of relations between intermediate states, with which memory protection is hard to enforce. In CompCertM, memory injections are enriched with explicit private and public memory to identify which memory regions need protection [Song et al. 2020], which introduces extra mechanisms and proofs. By contrast, our memory protection is naturally derived from the rely conditions in `injp` that already appeared in the vanilla CompCert.

Lastly, the semantic invariants implied by the direct refinement not only enable compiler optimizations based on static value analysis but also similar optimizations for hand-written assembly. For example, since the key value is never modified in the running example, we can inline 42 in the code, resulting in an optimized server as shown in Fig. 3c. Then, we reuse L_S and designate the global variable `key` as a constant. By exploiting the semantic invariants, we can prove $L_S \leqslant_{\mathbb{C}} [[\texttt{server\_opt.s}]]$ and derive a similar end-to-end correctness theorem as shown in Fig. 4.

2.4 Challenges for Establishing Direct Refinements

Having demonstrated the usefulness of using open simulation as direct refinements, we now face the problem of how to construct such a direct refinement for multi-pass optimizing compilers like CompCert. The obvious approach is to vertically compose the refinements proved for individual passes together. The most basic vertical composition for open simulations is stated as follows and can be easily proved by parallel pairing of individual simulations [Koenig and Shao 2021].

Theorem 2.3 (V. Comp). *Given $L_1 : A_1 \twoheadrightarrow B_1$, $L_2 : A_2 \twoheadrightarrow B_2$ and $L_3 : A_3 \twoheadrightarrow B_3$, and given $\mathbb{R}_{12} : A_1 \Leftrightarrow A_2$, $\mathbb{S}_{12} : B_1 \Leftrightarrow B_2$, $\mathbb{R}_{23} : A_2 \Leftrightarrow A_3$ and $\mathbb{S}_{23} : B_2 \Leftrightarrow B_3$,*

$$L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{S}_{12}} L_2 \Rightarrow L_2 \leqslant_{\mathbb{R}_{23} \twoheadrightarrow \mathbb{S}_{23}} L_3 \Rightarrow L_1 \leqslant_{\mathbb{R}_{12} \cdot \mathbb{R}_{23} \twoheadrightarrow \mathbb{S}_{12} \cdot \mathbb{S}_{23}} L_3.$$

Here, $(\_ \cdot \_)$ is a composed simulation convention s.t. $\mathbb{R} \cdot \mathbb{S} = \langle W_{\mathbb{R}} \times W_{\mathbb{S}}, \mathbb{R}^q \cdot \mathbb{S}^q, \mathbb{R}^r \cdot \mathbb{S}^r \rangle$ where for any q_1 and q_3, $(q_1, q_3) \in \mathbb{R}^q \cdot \mathbb{S}^q(w_{\mathbb{R}}, w_{\mathbb{S}}) \Leftrightarrow \exists q_2, (q_1, q_2) \in \mathbb{R}^q(w_{\mathbb{R}}) \land (q_2, q_3) \in \mathbb{S}^q(w_{\mathbb{S}})$ (similarly for $\mathbb{R}^r \cdot \mathbb{S}^r$). That is, the composed simulation assumes queries and replies are "concatenation" of individual queries and replies, and fed into and obtained from paralleling simulations, respectively. Then, given any compiler with N passes and their refinement relations $L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{S}_{12}} L_2, \ldots, L_N \leqslant_{\mathbb{R}_{N,N+1} \twoheadrightarrow \mathbb{S}_{N,N+1}} L_{N+1}$, we get their concatenation $L_1 \leqslant_{\mathbb{R}_{12} \cdot \ldots \cdot \mathbb{R}_{N,N+1} \twoheadrightarrow \mathbb{S}_{12} \cdot \ldots \cdot \mathbb{S}_{N,N+1}} L_{N+1}$, which exposes internal compilation and weakens compositionality as we have discussed in Sec. 1.2.

The above problem may be solved if the composed simulation convention can be *refined* into a single convention directly relating source and target queries and replies. Given two simulation conventions $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$, $\mathbb{R}$ is *refined* by $\mathbb{S}$ if

$$\forall\, w_{\mathbb{S}}\ q_1\ q_2,\ (q_1, q_2) \in \mathbb{S}^q(w_{\mathbb{S}}) \Rightarrow \exists\, w_{\mathbb{R}},\ (q_1, q_2) \in \mathbb{R}^q(w_{\mathbb{R}}) \land \\ \forall\, r_1\ r_2,\ (r_1, r_2) \in \mathbb{R}^r(w_{\mathbb{R}}) \Rightarrow (r_1, r_2) \in \mathbb{S}^r(w_{\mathbb{S}})$$

which we write as $\mathbb{R} \sqsubseteq \mathbb{S}$. If both $\mathbb{R} \sqsubseteq \mathbb{S}$ and $\mathbb{S} \sqsubseteq \mathbb{R}$, then $\mathbb{R}$ and $\mathbb{S}$ are equivalent (written as $\mathbb{R} \equiv \mathbb{S}$). By definition, $\mathbb{R} \sqsubseteq \mathbb{S}$ indicates any query for $\mathbb{S}$ can be converted into a query for $\mathbb{R}$ and any reply resulting from the converted query can be eventually converted back to a rely for $\mathbb{S}$. Therefore, $\mathbb{S}$ is a refinement more *general* than $\mathbb{R}$ and $\mathbb{R}$ is more *specialized* than $\mathbb{S}$. By wrapping the incoming side of an open simulation with a more general convention and its outgoing side with a more specialized convention, one gets another valid open simulation, described as follows [Koenig and Shao 2021]:

Theorem 2.4. *Given $L_1 : A_1 \twoheadrightarrow B_1$ and $L_2 : A_2 \twoheadrightarrow B_2$, if $\mathbb{R}'_A \sqsubseteq \mathbb{R}_A : A_1 \Leftrightarrow A_2$, $\mathbb{R}_B \sqsubseteq \mathbb{R}'_B : B_1 \Leftrightarrow B_2$ and $L_1 \leqslant_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$, then $L_1 \leqslant_{\mathbb{R}'_A \twoheadrightarrow \mathbb{R}'_B} L_2$.*

Now, we would like to prove the "real" vertical composition generating direct refinements. Given any $L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{R}_{12}} L_2$ and $L_2 \leqslant_{\mathbb{R}_{23} \twoheadrightarrow \mathbb{R}_{23}} L_3$, if we can show the existence of simulation conventions $\mathbb{R}_{13}$ directly relating source and target semantics s.t. $\mathbb{R}_{13} \equiv \mathbb{R}_{12} \cdot \mathbb{R}_{23}$, then $L_1 \leqslant_{\mathbb{R}_{13} \twoheadrightarrow \mathbb{R}_{13}} L_3$ holds by Theorem 2.3 and Theorem 2.4, which is the desired direct refinement. This composition is illustrated in Fig. 9 where the parts enclosed by dashed boxes represent the concatenation of

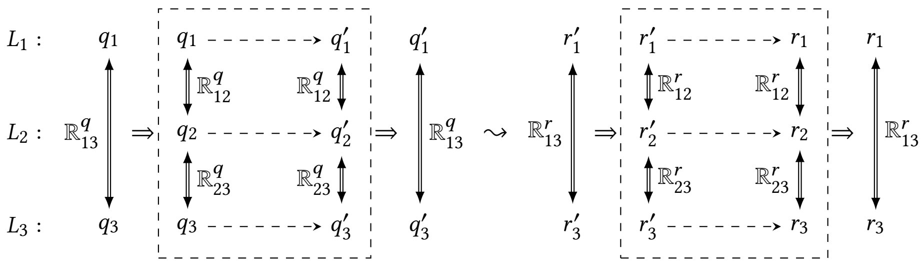

L_1 :
L_2 :
L_3 :
q_1 q_2 q_3 q_1' q_2' q_3'
$\mathbb{R}^q_{13}$ $\mathbb{R}^q_{12}$ $\mathbb{R}^q_{23}$
$\Rightarrow$ $\rightsquigarrow$
r_1 r_2 r_3 r_1' r_2' r_3'
$\mathbb{R}^r_{13}$ $\mathbb{R}^r_{12}$ $\mathbb{R}^r_{23}$

Fig. 9. Vertical Composition of Open Simulations by Refinement of Simulation Conventions

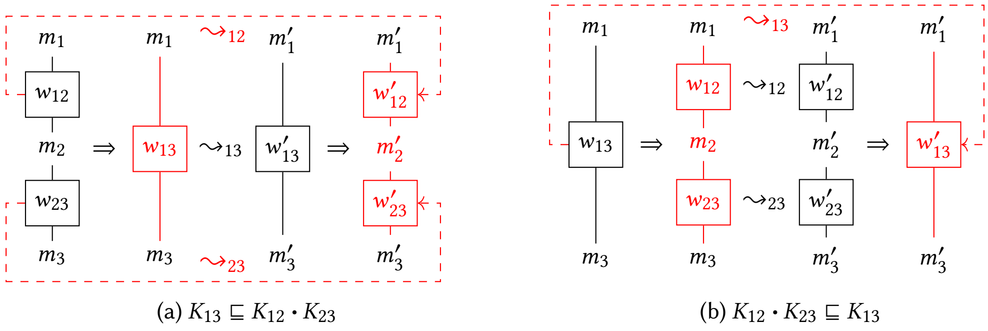

m_1 m_2 m_3 m_1' m_2' m_3'
w_{12} w_{23} w_{13} w_{12}' w_{23}' w_{13}'
$\rightsquigarrow_{12}$ $\rightsquigarrow_{13}$ $\rightsquigarrow_{23}$
$\Rightarrow$
(a) $K_{13} \sqsubseteq K_{12} \cdot K_{23}$
(b) $K_{12} \cdot K_{23} \sqsubseteq K_{13}$

Fig. 10. Composition of KMRs

$L_1 \leqslant_{\mathbb{R}_{12} \twoheadrightarrow \mathbb{R}_{12}} L_2$ and $L_2 \leqslant_{\mathbb{R}_{23} \twoheadrightarrow \mathbb{R}_{23}} L_3$. The direct queries and replies are split and merged for interaction with parallely running simulations underlying the direct refinement.

However, there exist significant conceptual and technical challenges for obtaining such direct refinements and for successfully applying them to realistic optimizing compilers, which are also the root causes for existing approaches to opt for weaker refinements as discussed in Sec. 1.2.

Challenge 1: Composition of KMRs. A major obstacle is to show that KMRs for individual simulations can be composed into a single KMR. For this, one needs to define refinements between KMRs. Given any KMRs K and L, $K \sqsubseteq L$ (i.e., K is refined by L) holds if the following is true:

$$\begin{aligned}
&\forall\, w_L,\ (m_1, m_2) \in R_L(w_L) \Rightarrow \exists\, w_K,\ (m_1, m_2) \in R_K(w_K) \land f_L(w_L) \subseteq f_K(w_K) \land \\
&\qquad\qquad \forall\, w_K'\ m_1'\ m_2',\ w_K \rightsquigarrow_K w_K' \Rightarrow (m_1', m_2') \in R_K(w_K') \Rightarrow \\
&\qquad\qquad\quad \exists\, w_L',\ w_L \rightsquigarrow_L w_L' \land (m_1', m_2') \in R_L(w_L') \land f_K(w_K') \subseteq f_L(w_L').
\end{aligned}$$

K and L are equivalent, written as $K \equiv L$ iff $K \sqsubseteq L$ and $L \sqsubseteq K$.

Continue with the proof of real vertical composition. Assume $\mathbb{R}_i$ is parameterized by KMR K_i, showing the existence of $\mathbb{R}_{13}$ s.t. $\mathbb{R}_{13} \sqsubseteq \mathbb{R}_{12} \cdot \mathbb{R}_{23}$ amounts to proving a parallel refinement over the parameterizing KMRs, i.e., there exists K_{13} s.t. $K_{13} \sqsubseteq K_{12} \cdot K_{23}$. A more intuitive interpretation is depicted in Fig. 10a where black symbols are $\forall$-quantified (assumptions we know) and red ones are $\exists$-quantified (conclusions we need to construct); note that Fig. 10a exactly mirrors the refinement on the outgoing side in Fig. 9. For simplicity, we not only use w_i to represent worlds, but also to denote $R_i(w_i)$ (where R_i is the Kripke relation given by KMR K_i) when it connects memory states through vertical lines. A dual property we need to prove for the incoming side is shown in Fig. 10b.

In both cases in Fig. 10, we need to construct interpolating states for relating source and target memory (i.e., m_2' in Fig. 10a and m_2 in Fig. 10b). This is in general considered very difficult. The construction of m_2' is especially challenging, for which we need to decompose the evolved world w_{13}' into w_{12}' and w_{23}' s.t. they are accessible from the original worlds w_{12} and w_{23}. It is not clear at all how this construction is possible because *1)* m_2' may have many forms since Kripke *relations* are in general non-deterministic (unlike functions), and *2)* KMR (e.g., injp) may introduce memory protections across external calls which may not hold after the composition.

Because of the above difficulties, existing approaches either make substantial changes to semantics for constructing interpolating states, thereby destroy adequacy [Stewart et al. 2015], or do not even try to merge Kripke memory relations, but instead leave them as separate entities [Koenig and Shao 2021; Song et al. 2020]. As a result, direct refinements cannot be achieved.

Challenge 2: End-to-end Direct Refinement for Optimizing Compilers. Even if a solution to composing KMRs is found, there are still problems to be solved for applying it to realistic optimizing compilers. First, we need to control the complexity of proofs when facing realistic compilers. Ideally, we would like to reuse the existing proofs as much as possible. However, it is not clear if the solution to Challenge 1 can indeed enable such reuse. Second, optimization passes generate additional rely-guarantee conditions on open semantics that cannot be composed into KMRs and that are embedded in the intermediate languages and interfere with direct refinements.

2.5 Our Approach

We discuss our approach to addressing the above challenges by using CompCert and CompCertO as the concrete platforms. To overcome the first challenge, we exploit the observation that `injp` in fact captures the rely-guarantee conditions for memory protection needed by any compiler pass in CompCert. Therefore, `injp` can be viewed as the most general KMR and adopted uniformly by all the compiler passes. Then, compositionality of KMRs depicted in Fig. 10 is reduced to transitivity of `injp`, i.e., `injp` $\equiv$ `injp` $\cdot$ `injp`. The proof is based on the discovery that constructing interpolating states for `injp` is possible because *1)* it encodes a *partial functional transformation* on memory states and *2)* any memory states not in the domain or range of the partial function is protected and unchanged throughout external calls. Although the proof is quite involved, the result can be reused for all compiler passes thanks to `injp`'s uniformity. The above solutions are discussed in Sec. 3.

To solve the second challenge, we notice that, although conceptually every compiler pass can use `injp`, it requires extensive rewriting of proofs. Instead, we start from the refinement proofs with least restrictive KMRs for individual passes in CompCertO, and exploit the properties that these KMRs can eventually be "absorbed" into `injp` in vertical composition to generate the direct refinement parameterized only by `injp`. CompCert also supports optimizations based on static value analysis, which rely on a *semantic invariant* that cannot be absorbed into `injp`. We discover that when piggybacked onto `injp` this semantic invariant can be transitively composed along with `injp`. By permutation of refinements, it is pushed to C level and becomes a condition for enabling optimization at the source level. Moreover, the pass for eliminating unused global variables (`Unusedglob`) can also be handled by absorbing its refinement into `injp`. In the end, we get an version of CompCert with end-to-end direct refinement $\leqslant_{\mathbb{C}}$. These solutions are discussed in Sec. 4.

Finally, we observe that source level refinements can be viewed as compilation using `injp`. Therefore, direct refinements can be easily extended to be end-to-end as we shall see in Sec. 5.

3 A UNIFORM AND COMPOSABLE KRIPKE MEMORY RELATION

The critical observation we make—also the cornerstone of this work—is that `injp` can serve as a uniform and composable KMR. We discuss its uniformity, i.e., `injp` directly captures the rely-guarantee conditions for memory protection for all compiler passes, and prove its transitivity.

3.1 Uniformity of injp

We show that `injp` is both a reasonable guarantee condition and a reasonable rely condition for memory protection for all the compiler passes in CompCert. The critical observation is that a notion of private and public memory can naturally be derived from `injp` where the private memory corresponds to regions unmapped by and out-of-reach from injections and the public memory corresponds to memory in the domain and image of injections. Then, any access to the public memory related by injection will never go out of the public memory. Therefore, the memory access is protected by `injp`. Details can be found in the technical report in the supplementary materials.

3.2 Transitivity of `injp`

The goal is to show the two refinements in Fig. 10 holds when K_{ij} = `injp`, i.e., `injp` ≡ `injp` · `injp`. As discussed in Sec. 2.4 the critical step is to construct interpolating memory states and Kripke worlds that transitively relate interpolating states to source and target states. Conceptually, this is possible because given an injection j and the source memory m_1, to build an interpolating memory m_2, we can project the values in m_1 through j to m_2. Furthermore, any state of m_2 not in the image of j would be determined by memory protection provided by `injp`. Below we elaborate on these key ideas and proof steps. A complete proof of transitivity of `injp` is given in the technical report.

$$\begin{tikzcd}
m_1 \arrow[rr] \arrow[d, "j_{12}"'] & & m_1' \arrow[dd, "j_{13}'"'] & & m_1 \arrow[rr] \arrow[d, "j_{12}"'] & \arrow[d, phantom, "\rightsquigarrow_{\texttt{injp}}"] & m_1' \arrow[d, "j_{12}'"] \arrow[dd, bend left=60, "j_{13}'"] \\
m_2 \arrow[d, "j_{23}"'] & \rightsquigarrow_{\texttt{injp}} & & \Rightarrow & m_2 \arrow[rr] \arrow[d, "j_{23}"'] & \arrow[d, phantom, "\rightsquigarrow_{\texttt{injp}}"] & m_2' \arrow[d, "j_{23}'"] \\
m_3 \arrow[rr] & & m_3' & & m_3 \arrow[rr] & & m_3'
\end{tikzcd}$$

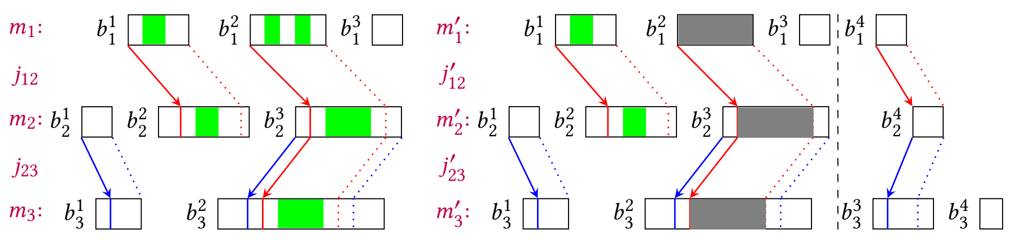
Fig. 11. Construction of Interpolating States

3.2.1 `injp` ⊑ `injp` · `injp`. By definition, we need to prove the following lemma:

Lemma 3.1. `injp` ⊑ `injp` · `injp` *holds. That is,*

$$\forall j_{12}\ j_{23}\ m_1\ m_2\ m_3,\ m_1 \hookrightarrow_m^{j_{12}} m_2 \Rightarrow m_2 \hookrightarrow_m^{j_{23}} m_3 \Rightarrow \exists j_{13},\ m_1 \hookrightarrow_m^{j_{13}} m_3 \wedge$$
$$\forall m_1'\ m_3'\ j_{13}',\ (j_{13}, m_1, m_3) \rightsquigarrow_{\texttt{injp}} (j_{13}', m_1', m_3') \Rightarrow m_1' \hookrightarrow_m^{j_{13}'} m_3' \Rightarrow$$
$$\exists m_2'\ j_{12}'\ j_{23}', (j_{12}, m_1, m_2) \rightsquigarrow_{\texttt{injp}} (j_{12}', m_1', m_2') \wedge m_1' \hookrightarrow_m^{j_{12}'} m_2'$$
$$\wedge (j_{23}, m_2, m_3) \rightsquigarrow_{\texttt{injp}} (j_{23}', m_2', m_3') \wedge m_2' \hookrightarrow_m^{j_{23}'} m_3'.$$

This lemma conforms to the graphic representation in Fig. 10a. To prove it, an obvious choice is to pick $j_{13} = j_{23} \cdot j_{12}$ (i.e., function composition of j_{12} and j_{23}). Then we are left to prove the existence of interpolating state m_2' and the memory and accessibility relations as shown in Fig. 11.



(a) Before (b) After

Fig. 12. Constructing of an Interpolating Memory State

We use the concrete example in Fig. 12 to motivate the construction of m_2'. Here, the white and green areas correspond to locations in `perm(_, Nonempty)` (i.e., having at least some permission) and in `perm(_, Readable)` (i.e., having at least readable permission). Given $m_1 \hookrightarrow_m^{j_{12}} m_2$, $m_2 \hookrightarrow_m^{j_{23}} m_3$ and $(j_{23} \cdot j_{12}, m_1, m_3) \rightsquigarrow_{\texttt{injp}} (j_{13}', m_1', m_3')$, we need to define j_{12}' and j_{23}' and then build m_2' satisfying $m_1' \hookrightarrow_m^{j_{12}'} m_2'$, $m_2' \hookrightarrow_m^{j_{23}'} m_3'$, $(j_{12}, m_1, m_2) \rightsquigarrow_{\texttt{injp}} (j_{12}', m_1', m_2')$, $(j_{23}, m_2, m_3) \rightsquigarrow_{\texttt{injp}} (j_{23}', m_2', m_3')$. m_1'

and m'_3 are expansions of m_1 and m_3 with new blocks and possible modification to the public regions of m_1 and m_3. Here, m'_1 has a new block b_1^4 and m'_3 has two new block b_3^3 and b_3^4.

We first fix j'_{12}, j'_{23} and the shape of blocks in m'_2. We begin with m_2 and introduce a newly allocated block b_2^4 whose shape matches with b_1^4 in m'_1. Then, j'_{12} is obtained by expanding j_{12} with identity mapping from b_1^4 to b_2^4. Furthermore, j'_{23} is also expanded with a mapping from b_2^4 to a block in m'_3; this mapping is determined by j'_{13}.

We then set the values and permissions for memory cells in m'_2 so that it satisfies injection and the `unchanged-on` properties for readable memory regions implied by $(j_{12}, m_1, m_2) \leadsto_{\texttt{injp}} (j'_{12}, m'_1, m'_2)$ and $(j_{23}, m_2, m_3) \leadsto_{\texttt{injp}} (j'_{23}, m'_2, m'_3)$. The values and permissions for newly allocated blocks are obviously mapped from m'_1 by j'_{12}. Those for old blocks are fixed as follows. By memory protection provided in $(j_{23} \cdot j_{12}, m_1, m_3) \leadsto_{\texttt{injp}} (j'_{13}, m'_1, m'_3)$, the only memory cells in m_1 that may have been modified in m'_1 are those mapped all the way to m_3 by $j_{23} \cdot j_{12}$, while the cells in m_3 that may be modified in m'_3 must be in the image of $j_{23} \cdot j_{12}$. To match this fact, the only old memory regions in m'_2 whose values and permissions may be modified are those both in the image of j_{12} and the domain of j_{23}. In our example, these regions are the gray areas in Fig. 12b. The *permissions* in those regions are then projected from m'_1 by applying the injection function j_{12}. Note that *values* in those regions are projected only if they are *not read-only* in m_2. The remaining old memory regions should have the same values and permissions as in m_2.

In summary, the values in m'_2 can be derived from m'_1 because of the following two reasons. First, as memory injections get composed, unmapped and out-of-reach regions get *bigger*, meaning more memory regions gets protected. For example, in Fig. 12, b_1^1 is mapped by j_{12} but becomes unmapped by $j_{23} \cdot j_{12}$; the image of b_2^1 in b_3^1 is in-reach by j_{23} but becomes out-of-reach by $j_{23} \cdot j_{12}$. Protected regions must have unchanged values and permissions. The only unprotected regions are those in the domain and image of the composed injections (i.e., gray areas in Fig. 12), whose values can be easily fixed because injections are partial functions: we can deterministically project the values and permission from a source state into the interpolating state.

3.2.2 $\texttt{injp} \cdot \texttt{injp} \sqsubseteq \texttt{injp}$. By definition, we need to prove:

Lemma 3.2. $\texttt{injp} \cdot \texttt{injp} \sqsubseteq \texttt{injp}$ *holds. That is,*

$$\forall j_{13}\ m_1\ m_3,\ m_1 \hookrightarrow_m^{j_{13}} m_3 \Rightarrow \exists j_{12}\ j_{23}\ m_2,\ m_1 \hookrightarrow_m^{j_{12}} m_2 \land m_2 \hookrightarrow_m^{j_{23}} m_3 \land$$
$$\forall m'_1\ m'_2\ m'_3\ j'_{12}\ j'_{23},\ (j_{12}, m_1, m_2) \leadsto_{\texttt{injp}} (j'_{12}, m'_1, m'_2) \Rightarrow (j_{23}, m_2, m_3) \leadsto_{\texttt{injp}} (j'_{23}, m'_2, m'_3) \Rightarrow$$
$$m'_1 \hookrightarrow_m^{j'_{12}} m'_2 \Rightarrow m'_2 \hookrightarrow_m^{j'_{23}} m'_3 \Rightarrow \exists j'_{13},\ (j_{13}, m_1, m_3) \leadsto_{\texttt{injp}} (j'_{13}, m'_1, m'_3) \land m'_1 \hookrightarrow_m^{j'_{13}} m'_3.$$

This lemma conforms to the graphic representation in Fig. 10b. To prove it, we pick j_{12} to be an identity injection, $j_{23} = j_{13}$ and $m_2 = m_1$. Then the lemma is reduced to proving the existence of j'_{13} that satisfies $(j_{13}, m_1, m_3) \leadsto_{\texttt{injp}} (j'_{13}, m'_1, m'_3)$ and $m'_1 \hookrightarrow_m^{j'_{13}} m'_3$. By picking $j'_{13} = j'_{12} \cdot j'_{23}$, we can easily prove these properties are satisfied by exploiting the guarantees provided by `injp`.

4 DERIVATION OF THE DIRECT REFINEMENT FOR COMPCERT

CompCert compiles `Clight` programs into `Asm` programs through 19 passes [Leroy 2023], including several optimization passes working on the `RTL` intermediate language. In this section, we discuss the vertical composition of open simulations for all these passes into the direct refinement $\leqslant_{\mathbb{C}}$ following the approach discussed in Sec. 2.5. It consists of the following three steps. First, we prove the open simulation for all these passes with appropriate simulation conventions. In particular, we directly reuse the proofs of non-optimizing passes in CompCertO. For the optimization passes, we adapt the original proofs in CompCert to match the semantic invariants for handling optimizations. Second, we prove a collection of properties for refining simulation conventions in preparation for vertical composition. Those properties enables absorption of KMRs into `injp` and composition of

Table 1. Significant Passes of CompCert

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

semantic invariants. They rely critically on the transitivity of `injp` in Sec. 3. Finally, we vertically compose the simulation proofs and refine the incoming and outgoing simulation conventions into the single convention $\mathbb{C}$. We elaborate on the above steps in the subsequent subsections.

4.1 Open Simulation of Individual Passes

We list the compiler passes and their simulation types in Table 1 (passes on the right follows the last pass on the left), together with their source and target languages and interfaces (in bold fonts). The passes in black are reused from CompCertO, while those in red are reproved optimizing passes. The passes in blue are *self-simulating* passes which will be used in Sec. 4.3. Note that we have omitted passes with the identity simulation convention (i.e., with simulations $L_1 \leqslant_{\mathtt{id}} L_2$) in Table 1 as they do not affect the proofs. [2] Since the non-optimizing passes can be directly reused, the only non-trivial work is to prove open simulations for optimizations.

4.1.1 *Simulation Conventions and Semantic Invariants.*

We first introduce relevant simulation conventions and semantic invariants for simulation proofs. The simulation conventions $\mathtt{c}_K : \mathcal{C} \Leftrightarrow \mathcal{C}$, $\mathtt{ltl}_K : \mathcal{L} \Leftrightarrow \mathcal{L}$, $\mathtt{mach}_K : \mathcal{M} \Leftrightarrow \mathcal{M}$, and $\mathtt{asm}_K : \mathcal{A} \Leftrightarrow \mathcal{A}$ relate the same language interfaces with queries and replies native to the associated intermediate languages, and are most commonly used throughout the verification. They are parameterized by KMR K to allow different compiler passes to have different assumptions on memory evolution. Conceptually, this parameterization is unnecessary as we can simply use `injp` for every pass due to its uniformity (as discussed in Sec. 3.1). Nevertheless, it is useful because the compiler proofs become simpler and more nature when using least restrictive KMRs which may be weaker than `injp`. CompCertO defines several KMRs weaker than `injp`. `id` is used when memory is unchanged. `ext` is used when the source and target memory shares the same structure. `inj` is a simplified version of `injp` without its memory protection. The simulation conventions $\mathtt{CL} : \mathcal{C} \Leftrightarrow \mathcal{L}$, $\mathtt{LM} : \mathcal{L} \Leftrightarrow \mathcal{M}$ and $\mathtt{MA} : \mathcal{M} \Leftrightarrow \mathcal{A}$ capture the calling convention of CompCert: `CL` relates C-level queries and replies with those in the `LTL` language where the arguments are distributed to abstract stack slots; `LM` further relates abstract stack slots with states on an architecture independent machine; `MA` relates this state to registers and memory in the assembly language (X86 assembly in this case). As discussed before, some refinements rely on invariants on the source semantics. The semantic invariant `wt` enforces that arguments and return values of function calls respect function signatures. `ro` is critical for ensuring the correctness of optimizations, which will be discussed next.

[2]The omitted passes are `Cshmgen`, `Renumber`, `Linearize`, `CleanupLabels` and `Debugvar`

4.1.2 *Open Simulation of Optimization Passes.*

The optimizing passes `Constprop`, `CSE` and `Deadcode` perform constant propagation, common subexpression elimination and dead code elimination, respectively. They make use of a static analysis algorithm for collecting information of variables during the execution (e.g. a constant global variable always has its initial value). More specifically, for each function, the algorithm starts with the known initial values of read-only (constant) global variables. It simulates the program execution to analyze the values of global or local variables after executing each instruction. In particular, for global constant variables, their references at any point should have the initial values of constants. For local variables stored on the stack, their references may have initial values or may not if interfered by other function calls. When the analysis encounters a call to another function, it checks whether the address of current stack frame is leaked to the callee directly through arguments or indirectly through pointers in memory. If not, then the stack frame is considered *unreachable* from its callee. Consequently, the references to unreachable local variables after function calls remain to be their initial values. Based on this analysis, the three passes then identify and perform optimizations.

Most of the proofs of closed simulations for those passes can be adapted to open simulation straightforwardly. The only and main difficulty is to prove that information derived from static analysis is consistent with the dynamic memory states in incoming queries and after external calls return. We introduce the semantic invariant `ro` and combine it with `injp` to ensure this consistency. Therefore, the above optimization passes all use $\texttt{ro} \cdot \texttt{c}_{\texttt{injp}}$ as their simulation conventions (because RTL conforms to the C interface). The adaptation of optimization proofs for those pass is similar. As an example, we only discuss constant propagation whose correctness theorem is stated as follows:

```
1 const int key = 42;
2 void foo(int*);
3 int double_key() {
4   int a = key;
5   foo(&key);
6   return a + key;
7 }
```

(a) Source Program

```
1 const int key = 42;
2 void foo(int*);
3 int double_key() {
4   int a = 42;
5   foo(&key);
6   return 84;
7 }
```

(b) Target Program

Fig. 13. An Example of Constant Propagation

Lemma 4.1. $\forall (M\ M' : RTL), \texttt{Constprop}(M) = M' \Rightarrow [\![M]\!] \leqslant_{\texttt{ro}\cdot\texttt{c}_{\texttt{injp}}} [\![M']\!]$.

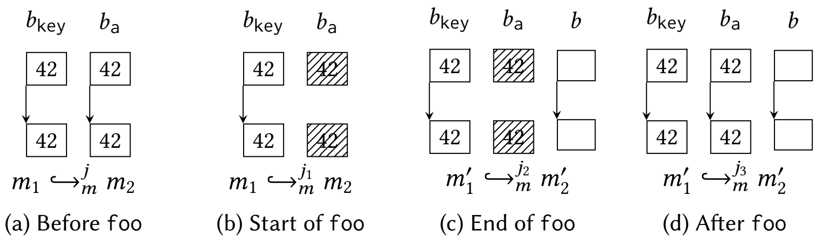
We discuss the proof of this lemma by illustrating how `ro` and `injp` help establish the open simulation for `Constprop` through a concrete example as depicted in Fig. 13. This example covers optimization for both global constants (`key`) and local variables (`a`) which are the focus of static analysis. By static analysis of Fig. 13a, *1)* `key` contains 42 at line 4 because `key` is a constant global variable and, *2)* both `key` and `a` contain 42 after the external call to `foo` returns to line 6. Here, the analysis confirms `key` has value 42 because `foo` (if well-behaved) will not modify a constant global variable. Furthermore, `a` has value 42 because it resides on the stack frame of `double_key` which is unreachable from `foo`. As a result, the source program is optimized to that in Fig. 13b.

We first show that `ro` guarantees the dynamic values of global constants are consistent with static analysis. That is, global variables are correct in incoming memory and are protected during external calls. `ro` is defined as follows:

Definition 4.2. $\texttt{ro} : C \Leftrightarrow C = \langle W_{\texttt{ro}}, \mathbb{R}^q_{\texttt{ro}}, \mathbb{R}^r_{\texttt{ro}} \rangle$ where $W_{\texttt{ro}} = (\texttt{symtbl} \times \texttt{mem})$ and

$$\mathbb{R}^q_{\texttt{ro}}(se, m) = \{(v_f[sg](\vec{v})@m, v_f[sg](\vec{v})@m) \mid \texttt{ro-valid}(se, m)\}$$
$$\mathbb{R}^r_{\texttt{ro}}(se, m) = \{(res@m', res@m') \mid \texttt{mem-acc}(m, m')\}$$

Note that although `ro` takes the form of a simulation convention, its relations only relate the same queries and replies, i.e., enforcing invariants only on one side. This kind of simulation conventions are what we called *semantic invariants*. The symbol tables (of type `symtbl` and a mapping from global symbols to memory addresses) are provided together with memory, so that the semantics



b_{key}
b_{a}
b
42
$m_1 \hookrightarrow^{j}_{m} m_2$
$m_1 \hookrightarrow^{j_1}_{m} m_2$
$m_1' \hookrightarrow^{j_2}_{m} m_2'$
$m_1' \hookrightarrow^{j_3}_{m} m_2'$
(a) Before foo
(b) Start of foo
(c) End of foo
(d) After foo

Fig. 14. Memory Injections from Call to Return of foo

can locate memory blocks of global definitions. $\texttt{ro-valid}(se, m)$ states that the values of global constant variables in the incoming memory m are the same as their initial values. Therefore, the optimization of key into 42 at line 4 of Fig. 13a is correct. For the external call to foo, monotonicity $\texttt{mem-acc}(m, m')$ ensures that read-only values in memory are unchanged, therefore the above property is preserved from external queries to replies (i.e., $\texttt{ro-valid}(se, m) \Rightarrow \texttt{mem-acc}(m, m') \Rightarrow \texttt{ro-valid}(se, m')$). Therefore, replacing key with 42 at line 6 makes sense.

We then show that injp guarantees the dynamic values of unreachable local variables are consistent with static analysis. That is, unreachable stack values are unchanged by external calls. This protection is realized by injp with *shrinking* memory injections. Fig. 14 shows the protection of a when calling foo. Before the external call to foo, the source block b_{a} and b_{key} are *mapped* to target blocks by the current injection j. If the analysis determines that the value of a is unchanged during foo, it indicates that *1)* the arguments to foo do not point to b_{a} and *2)* other pointers in m do not point to b_{a}. Therefore, we can simply remove b_{a} from the domain of j to get a shrunk yet valid memory injection j_1. Then, b_{a} is *unmapped* from the injection and protected during the call to foo. The injection of b_{a} is added back after the call returns and the simulation continues.

Finally, we talk a bit about the Unusedglob pass which removes unused static global variables from source modules. This pass was excluded from CompCertO because the original framework does not support the source and target semantics having different set of global symbols. We solve this problem by maintaining a skeleton of global definitions which stays the same throughout the compilation and maintaining local symbol tables that are subset of such skeletons. An interesting observation is that unlike other passes, injp is not needed for memory protection at the outgoing side (only inj is used) as this pass only changes global symbols, not external execution.

4.2 Properties for Refining Simulation Conventions

We present properties necessary for compose the simulation conventions in Table 1.

4.2.1 *Commutativity of KMRs and Structural Conventions.*

Lemma 4.3. *For* $\texttt{XY} \in \{\texttt{CL}, \texttt{LM}, \texttt{MA}\}$ *and* $K \in \{\texttt{ext}, \texttt{inj}, \texttt{injp}\}$ *we have* $\texttt{X}_K \cdot \texttt{XY} \sqsubseteq \texttt{XY} \cdot \texttt{Y}_K$.

This lemma is provided by CompCertO [Koenig and Shao 2021]. X and Y denote the simulation conventions for the source and target languages, respectively. For instance, $\texttt{X} = \texttt{c}$ and $\texttt{Y} = \texttt{ltl}$ when $\texttt{XY} = \texttt{CL}$. This lemma indicates at the outgoing (incoming) side a convention lower (higher) than CL, LM, MA may be lifted over them to a higher position (pushed down to a lower position).

4.2.2 Absorption of KMRs into injp*.* After we move the simulation conventions parameterized over different KMRs to the same level, we try to absorb all KMRs into injp. The following properties are needed for this purpose.

Lemma 4.4. *For any* $\mathbb{R}$, (1)$\mathbb{R}_{\texttt{injp}} \cdot \mathbb{R}_{\texttt{injp}} \equiv \mathbb{R}_{\texttt{injp}}$ (2)$\mathbb{R}_{\texttt{injp}} \sqsubseteq \mathbb{R}_{\texttt{inj}}$ (3)$\mathbb{R}_{\texttt{injp}} \cdot \mathbb{R}_{\texttt{inj}} \cdot \mathbb{R}_{\texttt{injp}} \sqsubseteq \mathbb{R}_{\texttt{injp}}$ (4)$\mathbb{R}_{\texttt{inj}} \cdot \mathbb{R}_{\texttt{inj}} \sqsubseteq \mathbb{R}_{\texttt{inj}}$ (5)$\mathbb{R}_{\texttt{ext}} \cdot \mathbb{R}_{\texttt{inj}} \equiv \mathbb{R}_{\texttt{inj}}$ (6)$\mathbb{R}_{\texttt{inj}} \cdot \mathbb{R}_{\texttt{ext}} \equiv \mathbb{R}_{\texttt{inj}}$ (7)$\mathbb{R}_{\texttt{ext}} \cdot \mathbb{R}_{\texttt{ext}} \equiv \mathbb{R}_{\texttt{ext}}$.

This lemma is a collection of refinement properties for simulation conventions lifted from their parameterizing KMRs. Property (1) is a direct consequence of $\texttt{injp} \cdot \texttt{injp} \equiv \texttt{injp}$ (Sec. 3), which is critical for merging to simulations using $\texttt{injp}$. Property (2) indicates we can replace $\mathbb{R}_{\texttt{inj}}$ with $\mathbb{R}_{\texttt{injp}}$ at the outgoing side of a simulation. Property (3) is for absorbing $\mathbb{R}_{\texttt{inj}}$ into surrounding $\mathbb{R}_{\texttt{injp}}$ at the incoming side. These three properties in essence depend on the transitivity and uniformity of $\texttt{injp}$ discussed in Sec. 3. The remaining properties are provided by CompCertO and their proofs are easy. Property (4) states $\mathbb{R}_{\texttt{inj}} \cdot \mathbb{R}_{\texttt{inj}}$ is refined by $\mathbb{R}_{\texttt{inj}}$. Note that the refinement from another direction is not provable because of lack of memory protection for constructing interpolating states (therefore $\texttt{injp}$ is needed in this case). Properties (5-6) indicate $\mathbb{R}_{\texttt{ext}}$ may be absorbed into $\mathbb{R}_{\texttt{inj}}$ and property (7) indicates $\mathbb{R}_{\texttt{ext}}$ is transitive.

4.2.3 Composition of Semantic Invariants. Lastly, we also need to handle the two semantic invariants $\texttt{ro}$ and $\texttt{wt}$. They cannot be absorbed into $\texttt{injp}$ because their assumptions are fundamentally different. Therefore, our goal is to permute them to the top-level and merge any duplicated copies.

LEMMA 4.5. *For any* $\mathbb{R}_K : C \Leftrightarrow C$, *we have* $(1)\mathbb{R}_K \cdot \texttt{wt} \equiv \texttt{wt} \cdot \mathbb{R}_K \cdot \texttt{wt}$ *and* $(2)\mathbb{R}_K \cdot \texttt{wt} \equiv \texttt{wt} \cdot \mathbb{R}_K$.

$\texttt{wt}$ is easy to handle with the above properties. They enable elimination and permutation of $\texttt{wt}$. $\texttt{ro}$ is more difficult to handle as it does not commute with simulation conventions. To eliminate redundant $\texttt{ro}$, we piggyback $\texttt{ro}$ onto $\texttt{injp}$ and prove the following transitivity property:

LEMMA 4.6. $\texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \equiv \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{ro} \cdot \texttt{c}_{\texttt{injp}}$

The proof is similar to $\texttt{c}_{\texttt{injp}} \equiv \texttt{c}_{\texttt{injp}} \cdot \texttt{c}_{\texttt{injp}}$. The key is to make sure properties of $\texttt{ro}$ are propagated to intermediate states when injections are splitting and merging (see the technical report).

Finally, we need that $\texttt{ro}$ and $\texttt{wt}$ are commutative at the top level, which is straightforward to prove because they are independent from each other.

LEMMA 4.7. $\texttt{ro} \cdot \texttt{wt} \equiv \texttt{wt} \cdot \texttt{ro}$

4.3 Proving Direct Open Simulation for CompCert

We first insert self-simulations into the compiler passes, as shown in Table 1. This is to supply extra $\mathbb{R}_{\texttt{inj}}$, $\mathbb{R}_{\texttt{injp}}$, and $\texttt{ro}$ for absorbing $\mathbb{R}_{\texttt{ext}}$ ($\mathbb{R}_{\texttt{inj}}$) into $\mathbb{R}_{\texttt{inj}}$ ($\mathbb{R}_{\texttt{injp}}$) by properties in Lemma 4.4, and transitive composition of $\texttt{ro}$. Self-simulations are obtained by the following lemma.

THEOREM 4.8. *If p is a program written in* $\texttt{Clight}$ *or* $\texttt{RTL}$ *and* $\mathbb{R} \in \{\texttt{c}_{\texttt{ext}}, \texttt{c}_{\texttt{inj}}, \texttt{c}_{\texttt{injp}}\}$, *or p is written in* $\texttt{Asm}$ *and* $\mathbb{R} \in \{\texttt{asm}_{\texttt{ext}}, \texttt{asm}_{\texttt{inj}}, \texttt{asm}_{\texttt{injp}}\}$, *then* $[[p]] \leqslant_{\mathbb{R} \twoheadrightarrow \mathbb{R}} [[p]]$ *holds.*

We then unify the conventions at the incoming and outgoing sides. We start with the simulation $L_1 \leqslant_{\mathbb{R} \twoheadrightarrow \mathbb{S}} L_2$ which is the transitive composition of compiler passes in Table 1 where

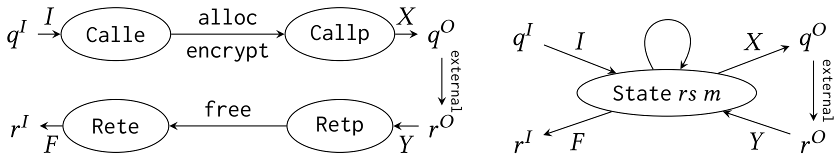
$$\begin{aligned}\mathbb{R} = {} & \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{wt} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{c}_{\texttt{inj}} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \\ & \cdot \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{c}_{\texttt{inj}} \cdot \texttt{wt} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{CL} \cdot \texttt{ltl}_{\texttt{ext}} \cdot \texttt{ltl}_{\texttt{injp}} \cdot \texttt{LM} \cdot \texttt{mach}_{\texttt{ext}} \cdot \texttt{MA} \cdot \texttt{asm}_{\texttt{inj}} \cdot \texttt{asm}_{\texttt{injp}} \\ \mathbb{S} = {} & \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{c}_{\texttt{inj}} \cdot \texttt{c}_{\texttt{inj}} \cdot \texttt{wt} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{c}_{\texttt{inj}} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{c}_{\texttt{inj}} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \\ & \cdot \texttt{ro} \cdot \texttt{c}_{\texttt{injp}} \cdot \texttt{c}_{\texttt{inj}} \cdot \texttt{wt} \cdot \texttt{c}_{\texttt{ext}} \cdot \texttt{CL} \cdot \texttt{ltl}_{\texttt{ext}} \cdot \texttt{LM} \cdot \texttt{mach}_{\texttt{inj}} \cdot \texttt{mach}_{\texttt{ext}} \cdot \texttt{MA} \cdot \texttt{asm}_{\texttt{inj}} \cdot \texttt{asm}_{\texttt{injp}}.\end{aligned}$$

We then find two sequences of refinements $\mathbb{C} \sqsubseteq \mathbb{R}_n \sqsubseteq \ldots \sqsubseteq \mathbb{R}_1 \sqsubseteq \mathbb{R}$ and $\mathbb{S} \sqsubseteq \mathbb{S}_1 \sqsubseteq \ldots \sqsubseteq \mathbb{S}_m \sqsubseteq \mathbb{C}$, by which and Theorem 2.4 to get the simulation $L_1 \leqslant_{\mathbb{C} \twoheadrightarrow \mathbb{C}} L_2$. The direct simulation convention is $\mathbb{C} = \texttt{ro} \cdot \texttt{wt} \cdot \texttt{CAinjp} \cdot \texttt{asm}_{\texttt{injp}}$. $\texttt{ro}$ enables optimizations at C level while $\texttt{wt}$ ensures well-typedness. $\texttt{CAinjp}$ is the key component discussed informally in Sec. 2.2; its formal definition is given in the technical report. The tailing $\texttt{asm}_{\texttt{injp}}$ is irrelevant as assembly code is self-simulating by Theorem 4.8.

The final correctness theorem is shown below:

THEOREM 4.9. *Compilation in CompCert is correct in terms of open simulations,*

$$\forall\, (M : \texttt{Clight})\ (M' : \texttt{Asm}),\ \texttt{CompCert}(M) = M' \Rightarrow [[M]] \leqslant_{\mathbb{C}} [[M']].$$



(a) L_S (b) [[server_opt.s]]

Fig. 15. The Specification and Open Semantics of server_opt.s

Below we explain how the refinements are carried out at the outgoing side. Refinements at the incoming side is similar and discussed in the technical report. The following is the sequence of refined simulation conventions $\mathbb{C} \sqsubseteq \mathbb{R}_n \sqsubseteq \ldots \sqsubseteq \mathbb{R}_1 \sqsubseteq \mathbb{R}$. It begins with the composition of all initial outgoing conventions (including those for self-simulations) and ends with the unified convention.

(1) $\mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{injp}} \cdot \mathtt{wt} \cdot \mathtt{c_{ext}} \cdot \mathtt{c_{ext}} \cdot \mathtt{c_{inj}} \cdot \mathtt{c_{ext}} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}}$
$\cdot \mathtt{c_{inj}} \cdot \mathtt{wt} \cdot \mathtt{c_{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl_{ext}} \cdot \mathtt{ltl_{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach_{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm_{inj}} \cdot \mathtt{asm_{injp}}$
(2) $\mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{wt} \cdot \mathtt{c_{inj}} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}}$
$\cdot \mathtt{c_{inj}} \cdot \mathtt{wt} \cdot \mathtt{c_{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl_{ext}} \cdot \mathtt{ltl_{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach_{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm_{inj}} \cdot \mathtt{asm_{injp}}$
(3) $\mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{wt} \cdot \mathtt{c_{inj}} \cdot \mathtt{wt} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}}$
$\cdot \mathtt{c_{inj}} \cdot \mathtt{c_{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl_{ext}} \cdot \mathtt{ltl_{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach_{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm_{inj}} \cdot \mathtt{asm_{injp}}$
(4) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{inj}} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}}$
$\cdot \mathtt{c_{inj}} \cdot \mathtt{c_{ext}} \cdot \mathtt{CL} \cdot \mathtt{ltl_{ext}} \cdot \mathtt{ltl_{injp}} \cdot \mathtt{LM} \cdot \mathtt{mach_{ext}} \cdot \mathtt{MA} \cdot \mathtt{asm_{inj}} \cdot \mathtt{asm_{injp}}$
(5) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{inj}} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{inj}} \cdot \mathtt{c_{ext}} \cdot \mathtt{c_{ext}} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{ext}} \cdot \mathtt{c_{inj}} \cdot \mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA} \cdot \mathtt{asm_{injp}}$
(6) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{injp}} \cdot \mathtt{c_{injp}} \cdot \mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA} \cdot \mathtt{asm_{injp}}$
(7) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA} \cdot \mathtt{asm_{injp}}$
(8) $\mathtt{wt} \cdot \mathtt{ro} \cdot \mathtt{c_{injp}} \cdot \mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA} \cdot \mathtt{asm_{injp}}$
(9) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{c_{injp}} \cdot \mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA} \cdot \mathtt{asm_{injp}}$
(10) $\mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{CAinjp} \cdot \mathtt{asm_{injp}}$

In each line, the letters in red are simulation conventions transformed by the refinement operation at that step. In step (1), we merge consecutive simulation conventions by applying property (1) in Lemma 4.4 for $\mathtt{c_{injp}}$ and properties (5-7) to compose $\mathtt{c_{ext}}$ and absorb it into $\mathtt{c_{inj}}$. We also apply Lemma 4.6 to merge consecutive $\mathtt{ro} \cdot \mathtt{c_{injp}}$. In step (2), we move wt to higher positions by property (2) in Lemma 4.5. In step (3), we eliminate the first wt by property (1) in Lemma 4.5 and move the remaining wt higher by property (2) in Lemma 4.5 and Lemma 4.7. In step (4), we lift conventions over CL, LM and MA to higher positions by Lemma 4.3. In step (5), we absorb $\mathtt{c_{ext}}$ into $\mathtt{c_{inj}}$ again and further turns $\mathtt{c_{inj}}$ into $\mathtt{c_{injp}}$ by applying $\mathtt{c_{injp}} \sqsubseteq \mathtt{c_{inj}}$ (property (2) in Lemma 4.4). In step (6), we compose $\mathtt{c_{injp}}$ by applying $\mathtt{c_{injp}} \equiv \mathtt{c_{injp}} \cdot \mathtt{c_{injp}}$. In step (7), we apply Lemma 4.6 again to eliminate the second $\mathtt{ro} \cdot \mathtt{c_{injp}}$. In step (8), we commute the two semantic invariants of the source semantics by Lemma 4.7. Finally, we merge $\mathtt{c_{injp}}$ with $\mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA}$ into CAinjp.

5 END-TO-END VERIFICATION OF HETEROGENEOUS MODULES

In this section, we give a formal account of end-to-end verification of heterogeneous modules based on direct refinements. Because of limit in space, we only discuss end-to-end verification of the running example in Fig. 4. More complicated examples with *mutually recursive* calls can be found in the technical report. We use server_opt.s instead of server.s to illustrate how optimizations are enabled by ro. The proof for the unoptimized server is similar with only minor adjustments.

5.1 Refinement for the Hand-written Server

We give a formal definition of LTS for L_S along with a transition diagram in Fig. 15a.

Definition 5.1. LTS of L_S:

$$\begin{tikzcd}
q^I_C \arrow[r, "I"] \arrow[d, leftrightarrow, "\mathbb{C}^q(w)"] & \texttt{Calle}\ i\ v_f\ m \arrow[r, "\texttt{alloc}\ sp", "\texttt{store}\ sp"'] \arrow[d, leftrightarrow, "R^a(w)"] & \texttt{Callp}\ sp\ v_f\ m_1 \arrow[r, "X"] \arrow[d, leftrightarrow, "R^b(w)"'] & q^O_C \arrow[r, squiggly] \arrow[d, leftrightarrow, "\mathbb{C}^q(w')"'] & r^O_C \arrow[r, "Y"] \arrow[d, leftrightarrow, "\mathbb{C}^r(w')"] & \texttt{Retp}\ sp\ m_2 \arrow[r, "\texttt{free}\ sp"] \arrow[d, leftrightarrow, "R^c(w)"] & \texttt{Rete}\ m_3 \arrow[r, "F"] \arrow[d, leftrightarrow, "R^d(w)"'] & r^I_C \arrow[d, leftrightarrow, "\mathbb{C}^r(w)"'] \\
q^I_{\mathcal{A}} \arrow[r, "I"] & (rs, tm) \arrow[r, "\texttt{Pallocframe}", "\texttt{...}\ \texttt{Pcall}"'] & (rs_1, tm_1) \arrow[r, "X"] & q^O_{\mathcal{A}} \arrow[r, squiggly] & r^O_{\mathcal{A}} \arrow[r, "Y"] & (rs_2, tm_2) \arrow[r, "\texttt{Pfreeframe}", "\texttt{Pret}"'] & (rs_3, tm_3) \arrow[r, "F"] & r^I_{\mathcal{A}}
\end{tikzcd}$$

Fig. 16. Open Simulation between the Server and its Specification

$$\begin{aligned}
S_S &:= \{\texttt{Calle}\ i\ v_f\ m, \texttt{Callp}\ sp\ v_f\ m, \texttt{Retp}\ sp\ m, \texttt{Rete}\ m\};\\
I_S &:= \{(\texttt{Vptr}(b_e, 0)[\texttt{int} \to \texttt{ptr} \to \texttt{void}]([i, v_f])@m, \texttt{Calle}\ i\ v_f\ m)\};\\
\to_S &:= \{(\texttt{Calle}\ i\ v_f\ m, \texttt{Callp}\ sp\ v_f\ m') \mid m' = m[sp \leftarrow (i\ \texttt{XOR}\ m[b_k])]\} \cup\\
&\ \{(\texttt{Retp}\ sp\ m, \texttt{Rete}\ m') \mid \texttt{free}\ m\ sp = m'\};\\
X_S &:= \{(\texttt{Callp}\ sp\ \texttt{Vptr}(b_p, 0)\ m, \texttt{Vptr}(b_p, 0)[\texttt{int} \to \texttt{void}]([\texttt{Vptr}(sp, 0)])@m)\};\\
Y_S &:= \{(\texttt{Callp}\ sp\ v_f\ m, res@m', \texttt{Retp}\ sp\ m')\};\\
F_S &:= \{(\texttt{Rete}\ m, \texttt{Vundef}@m)\}.
\end{aligned}$$

The LTS has four internal states as depicted in Fig. 15a. Initialization is encoded in I. If the incoming query q^I contains a function pointer $\texttt{Vptr}(b_e, 0)$ which points to encrypt, L_S enters $\texttt{Calle}\ i\ v_f\ m$ where i and v_f are its arguments. The first internal transition allocates the stack frame sp and stores the result of encryption i XOR $m[b_k]$ in sp where b_k contains key. Then it enters Callp which is the state before calling process. If the pointer $v_f = \texttt{Vptr}(b_p, 0)$ of the current state points to an external function, L_S issues an outgoing C query q^O with a pointer to its stack frame as its argument. After the external call, Y_S updates the memory with the reply and enters Retp. The second internal transition frees sp and enters Rete and finally returns. Note that complete semantics of L_S is accompanied by a local symbol table which determines the initial value of global variables (key) and whether it is a constant (read-only). The only difference between specifications for server_opt.s and server.s is whether key is read-only in the symbol table. The semantics of assembly module [[server_opt.s]] is given by CompCertO whose transition diagram is shown in Fig. 15b. All the states, including queries and replies, are composed of register sets and memory.

Theorem 5.2. $L_S \leqslant_{\mathbb{C}} [[\texttt{server\_opt.s}]]$

Given the open semantics, we need to prove the above forward simulation. We discuss the key ideas behind its proof and leave the complete proof in the technical report. The most important points are how ro enables optimizations and how injp preserves memory across external calls.

At the top level, we expand $\mathbb{C}$ to $\texttt{ro} \cdot \texttt{wt} \cdot \texttt{CAinjp} \cdot \texttt{asm}_{\texttt{injp}}$ and switch the order of ro and wt by Lemma 4.7. By the vertical compositionality (Theorem 2.3), we first prove $L_S \leqslant_{\texttt{wt}} L_S$ which is a straightforward self-simulation and $[[\texttt{server\_opt.s}]] \leqslant_{\texttt{asm}_{\texttt{injp}}} [[\texttt{server\_opt.s}]]$ is proved by Theorem 4.8. Then, we are left with $L_S \leqslant_{\texttt{ro}\cdot\texttt{CAinjp}} [[\texttt{server\_opt.s}]]$. By definition, we can easily show $L_S \leqslant_{\texttt{ro}} L_S$. Note that this self-simulation does not convey much information on its own, its real purpose is to propagate the protection in ro-valid to the simulation invariant defined below.

We are left with proving $L_S \leqslant_{\texttt{CAinjp}} [[\texttt{server\_opt.s}]]$, i.e., to show the simulation diagram in Fig. 16 holds which is a complete picture of the example in Fig. 7. Here, the assumptions in forward simulation and conclusions we need to prove are represented as black and red arrows, respectively. For this, we need an invariant $R \in \mathcal{K}_{W_{\texttt{ro}\cdot\texttt{CAinjp}}}(S_S, \texttt{regset} \times \texttt{mem})$. The most important point is that ro and injp play essential roles in establishing the invariant. First, ro-valid is propagated throughout R to prove that value of key read from the memory is 42 from Calle to Callp, hence matches the constant in server_opt.s. Second, injp is essential for deriving that memory locations in sp with offset o s.t. $o \leq 8$ and $16 \leq o$ are unchanged since they are designated out-of-reach by R. Therefore, the private stack values of the server are protected.

Table 2. Comparison between VCC based on CompCert

	Direct Refinement	V. Comp	H. Comp	Adequacy	End-to-end Verification	Free-Form Heterogeneity
CompComp	No	Yes	Yes	No	No	Yes
CompCertM	No	RUSC	RUSC	Yes	Yes	Yes
CompCertO	No	Trivial	Yes	Yes	Unknown	Yes
CompCertX	No	CAL	CAL	Yes	CAL	No
This Work	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

5.2 End-to-end Correctness Theorem

Lemma 5.3 (Source-level Refinement). $L_{\mathsf{CS}} \leqslant_{\mathtt{ro}\cdot\mathtt{wt}\cdot\mathtt{c}_{\mathtt{injp}}} [[\mathtt{client.c}]] \oplus L_{\mathsf{S}}$

We prove the above source-level refinement where L_{CS} is the top-level specification (defined in the technical report). Its proof follows the same pattern as Theorem 5.2. The proof is considerably simpler because the source and target semantics share the same C interface. The simulation invariant R is mainly about the accessibility of injp. We omit the details due to limits in space.

As depicted in Fig. 4, to prove the forward simulation between the top-level specification and the target linked program. We exploit the horizontal compositionality and adequacy of assembly linking already provided by CompCertO's framework.

Theorem 5.4 (H. Compositionality and Adequacy).

$$\forall\ (L_1, L_2, L_1', L_2'),\ L_1 \leqslant_{\mathbb{R}\twoheadrightarrow\mathbb{R}} L_2 \Rightarrow L_1' \leqslant_{\mathbb{R}\twoheadrightarrow\mathbb{R}} L_2' \Rightarrow L_1 \oplus L_1' \leqslant_{\mathbb{R}\twoheadrightarrow\mathbb{R}} L_2 \oplus L_2'$$
$$\forall\ (M_1, M_2 : \mathsf{Asm}),\ [[M_1]] \oplus [[M_2]] \leqslant_{\mathtt{id}} [[M_1 + M_2]]$$

Lemma 5.5. $[[\mathtt{client.c}]] \oplus L_{\mathsf{S}} \leqslant_{\mathbb{C}} [[\mathtt{client.s}]] \oplus [[\mathtt{server\_opt.s}]]$

Proof. Immediate from Theorem 4.9, Theorem 5.2, and Theorem 5.4. □

For end-to-end open semantics with direct convention $\mathbb{C}$, we need to absorb Theorem 5.3 into it. The following theorem is easily derived by inserting injp and applying Lemma 4.5, 4.6 and 4.7.

Lemma 5.6. $\mathbb{C} \equiv \mathtt{ro} \cdot \mathtt{wt} \cdot \mathtt{c}_{\mathtt{injp}} \cdot \mathbb{C}$

The final end-to-end simulation is immediate by vertically composing Lemma 5.3, Lemma 5.5 and refining the simulation convention using Lemma 5.6.

Theorem 5.7. $L_{\mathsf{CS}} \leqslant_{\mathbb{C}} [[\mathtt{client.s} + \mathtt{server\_opt.s}]]$

6 RELATED WORK

Our entire Coq development took about 7 person months. We added 15k lines of code on top of CompCertO. A detailed evaluation is in the technical report. Below we compare our work with other frameworks for compositional compiler verification and program verification.

6.1 Verified Compositional Compilation for First-Order Languages

In this work, we are concerned with VCC of first-order imperative programs with global memory states and support of pointers. A majority of work in this setting is based on CompCert. We compare them from the perspectives listed in Table 2, including whether they support refinements that directly relate physical semantics of open modules and are ignorant of internal compilation processes, whether they support vertical and horizontal composition, whether they are adequate for physical semantics, whether they support end-to-end verification based on refinement, and whether they support heterogeneous modules that may be defined in different languages, complied through different passes and freely invoke each other, including mutually recursive calls. An answer that is not a simple “Yes” or “No” denotes that special constraints are enforced to support the specific feature. Note that our work is the only one that simultaneously supports all the features.

Compositional CompCert. CompComp supports VCC based on *interaction semantics* which is a specialized version of open semantics that only works with C interfaces [Stewart et al. 2015]. We have already talked about its merits and flaws in Sec. 1.2. It is interesting to note that CompComp can also be obtained based on our approach. If we adopt C-level simulation convention with `injp` (i.e., $\mathtt{c}_{\mathtt{injp}}$) for every compiler pass and compose them together by transitivity of $\mathtt{c}_{\mathtt{injp}}$, we basically get an alternative implementation of CompComp without its extra mechanisms or complexity.

CompCertM. CompCertM supports adequacy and end-to-end verification of mixed C and assembly programs. A distinguishing feature of CompCertM is Refinement Under Self-related Contexts or RUSC [Song et al. 2020]. A RUSC relation is a *fixed* collection of simulation relations. By exploiting the property of contexts that are self-relating under all of these simulation relations, horizontal and vertical compositionality are achieved. However, such compositionality is not extensional and difficult to use for even small examples. For example, the complete refinement relation $\leqslant_{R_1+\ldots+R_9}$ in CompCertM should carry all 9 RUSC relations $R_1, \ldots, R_9$ (6 for individual compiler passes and 3 for source-level verification). To prove that a refinement between hand-written assembly program `a.s` and its specification L_S can be horizontally composed with another refinement $\leqslant_{R_1+\ldots+R_9}$ (e.g., obtained from compilation), one needs to prove L_S and $[\![\mathtt{a.s}]\!]$ are self-simulating over *all* 9 simulation relations (the self-simulation for assembly language can be proved once and for all; that for L_S can only be proved manually). This can quickly get out of hand as more modules and more compiler passes are introduced. By contrast, we only need to prove direct refinement *for once* and the resulting refinement is open to further horizontal or vertical composition. On the other hand, CompCertM supports behavior refinement of *closed* programs which we do not support yet. For this, we need to close the open simulation with program loading. We plan to finish this and get a behavior refinement theorem like in the original CompCert.

CompCertO. Vertical composition is a trivial parallel pairing of simulations in CompCertO, which exposes internal compilation steps. CompCertO tries to alleviate this problem via ad-hoc refinement of simulation conventions. The resulting convention for the overall compilation is $\mathbb{C}_{\mathtt{CCO}} = \mathcal{R}^* \cdot \mathtt{wt} \cdot \mathtt{CL} \cdot \mathtt{LM} \cdot \mathtt{MA} \cdot \mathtt{asm}_{\mathtt{vainj}}$ where $\mathcal{R} = \mathtt{c}_{\mathtt{injp}} + \mathtt{c}_{\mathtt{inj}} + \mathtt{c}_{\mathtt{ext}} + \mathtt{c}_{\mathtt{vainj}} + \mathtt{c}_{\mathtt{vaext}}$ is a sum of conventions parameterized over KMRs where $\mathtt{c}_{\mathtt{vaext}}$ is an ad-hoc combination of KMR and internal invariants for optimizations. $\mathcal{R}^*$ means that $\mathcal{R}$ can be used for an arbitrary number of times. Note that the top-level summation of KMRs is similar to RUSC in CompCertM. Therefore, to horizontally compose these refinements, we need to go through the same process as in CompCertM, only more complicated because the necessity to reason about how simulations conform to internal invariants of optimizations and how such simulation can be repeated over all possible combination of KMRs indefinitely. Therefore, it is unknown if the correctness theorem of CompCertO suffices for end-to-end program verification. In this work, we have completely fixed these problems.

CompCertX. CompCertX [Gu et al. 2015; Wang et al. 2019] realizes a weaker form of VCC that only allows assembly contexts to invoke C programs, but not the other way around. Therefore, it does not support horizontal composition of modules with mutual recursions. The compositionality and program verification are delegated to Certified Abstraction Layers (CAL) [Gu et al. 2015, 2018]. Furthermore, CompCertX does not support stack-allocated data (e.g., our server example). However, its top-level semantic interface is similar to our interface, albeit not carrying a symmetric rely-guarantee condition. This indicates that our work is a natural evolution of CompCertX.

VCC for Concurrent Programs. Compositional verification of compilation for concurrent programs is a topic different from traditional VCC as it assumes working with multiple threads and execution environments. CASCompCert is an extension of CompComp that supports compositional compilation of concurrent programs with no (or benign) data races [Jiang et al. 2019]. To make CompComp's approach to VCC work in a concurrent setting, CASCompCert imposes some restrictions including not supporting stack-allocated data and allowing only nondeterminism in scheduling threads. A recent advancement based on CASCompCert is about verifying concurrent programs [Zha et al. 2022] running on weak memory models using the promising semantics [Kang et al. 2017; Lee et al. 2020]. We believe the ideas in CASCompCert are complementary to this work and can be adopted to achieve VCC for concurrency with cleaner interface and less restrictions.

6.2 Verified Compositional Compilation for Higher-Order Languages

Another class of work on VCC focuses on compilation of higher-order languages. In this setting, the main difficulty comes from complex language features together with higher-order states. A prominent example is the Pilsner compiler [Neis et al. 2015] that compiles a higher-order language into some form of assembly programs. The technique Pilsner adopts is called *parametric simulations* that evolves from earlier work on reasoning about program equivalence via bisimulation [Hur et al. 2012]. Another line of work is multi-language semantics [Patterson and Ahmed 2019; Patterson et al. 2017; Perconti and Ahmed 2014; Scherer et al. 2018] where a language combining all source, intermediate and target language is used to formalize the semantics and compiler correctness is stated using contextual equivalence or logical relations. It seems that our techniques are not directly applicable to those work because relational reasoning on higher-order states cannot deterministically fix the interpolating states. An interesting direction to explore is to combine our techniques with those work to deal with linear memory space with pointers in higher-order states.

The high-level idea of constructing interpolating memory state for proving transitivity of `injp` can also be found in some of the work on proving program equivalence using bisimulation [Chung-Kil Hur and Vafeiadis 2012] and logical relations [Ahmed 2006]. However, these work has considerably simpler memory model and does not deal with pointers. Furthermore, previous work considers it extremely difficult to construct interpolating states without extra mechanisms such as annotation of private and public memory. We discover that the notion of public and private memory inferred from `injp` which already exists in the vanilla CompCert suffices for this purpose. There is no need to make any changes to program semantics and verification framework for this.

6.3 Frameworks for Compositional Program Verification

Researchers have proposed frameworks for compositional program verification based on novel semantics [Chappe et al. 2023; He et al. 2021; Sammler et al. 2023; Xia et al. 2019] and *separation logics* [Song et al. 2023]. These frameworks aim at broader goals compared to VCC and may be combined with our approach to generate stronger end-to-end verification techniques. A detailed comparison is in the technical report.

7 CONCLUSION

Existing approaches to verified compositional compilation have difficulties in supporting open modules as their correctness theorems either do not connect with physical semantics or inevitably depend on internal working of compilation. We have proposed an approach to compositional compiler correctness via construction of refinements relating the source and target semantics at their native interfaces and that do not depend on any assumption on compilation, which overcomes the weaknesses of existing approaches. We have realized our approach in CompCert and demonstrated its effectiveness through end-to-end verification of non-trivial heterogeneous programs.

REFERENCES

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *ESOP*. 69–83.

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proc. ACM Program. Lang.* 7, POPL, Article 61 (jan 2023), 31 pages. https://doi.org/10.1145/3571254

Derek Dreyer Chung-Kil Hur, Georg Neis and Viktor Vafeiadis. 2012. *The Transitive Composability of Relation Transition Systems*. Technical Report, MPI-SWS-2012-002. MPI-SWS.

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)*. ACM, New York, 595–608. https://doi.org/10.1145/2775051.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjober, 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)*. ACM, New York, 646–661. https://doi.org/10.1145/3192366.3192381

Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. 2021. A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 135 (oct 2021), 29 pages. https://doi.org/10.1145/3485512

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *POPL'12*. 59–72.

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proc. 40th ACM Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, New York, 111–125. https://doi.org/10.1145/3314221.3314595

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. *SIGPLAN Not.* 52, 1 (jan 2017), 175–189. https://doi.org/10.1145/3093333.3009850

Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1095–1109. https://doi.org/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 *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362–376. https://doi.org/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

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 *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada). ACM Press, 166–178. https://doi.org/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 (jul 2019), 29 pages. https://doi.org/10.1145/3341689

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. *SIGPLAN Not.* 52, 6 (jun 2017), 495–509. https://doi.org/10.1145/3140587.3062347

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP'14*. 128–148.

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 775–805.

Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. 2018. Fabous Interoperability for ML and a Linear Language. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 146–162.

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 (2013), 22:1–22:50. https://doi.org/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 (Jan. 2020), 31 pages. https://doi.org/10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1121–1151.

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, 275–287. https://doi.org/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 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290375

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 903–917. https://doi.org/10.1145/3519939.3523734