Compositional CompCert

Gordon Stewart  Lennart Beringer  Santiago Cuellar  Andrew W. Appel
Princeton University
{jsseven,eberinge,scuellar,appel}@princeton.edu

Abstract
This paper reports on the development of Compositional CompCert, the first verified separate compiler for C.

Specifying and proving separate compilation for C is made challenging by the coincidence of: compiler optimizations, such as register spilling, that introduce compiler-managed (private) memory regions into function stack frames, and C’s stack-allocated addressable local variables, which may leak portions of stack frames to other modules when their addresses are passed as arguments to external function calls. The CompCert compiler, as built/proved by Leroy et al. 2006–2014, has proofs of correctness for whole programs, but its simulation relations are too weak to specify or prove separately compiled modules.

Our technical contributions that make Compositional CompCert possible include: language-independent linking, a new operational model of multilanguage linking that supports strong semantic contextual equivalences; and structured simulations, a refinement of Beringer et al.’s logical simulation relations that enables expressive module-local invariants on the state communicated between compilation units at runtime. All the results in the paper have been formalized in Coq and are available for download together with the Compositional CompCert compiler.

Categories and Subject Descriptors F.3.1 [Specifying and Verifying and Reasoning about Programs]: Mechanical verification

General Terms Verification

Keywords CompCert, Compiler Correctness

1. Introduction

Verified separate compilation is the process of independently translating a program’s components in a way that preserves correctness of the program as a whole. In the most general case, a verified separate compiler supports heterogeneous source programs, in which some modules are written in a high-level source language like C while others are written in a lower-level language such as assembly. A verified separate compiler in this context is one that preserves the behavior of the entire source program, comprising the C and assembly-language modules, when the C code is compiled.

Separate compilation has numerous practical benefits. It speeds up development cycles by enabling recompilation of just those source modules that have been edited by the programmer. It enables shared libraries—separately compiled modules mapped at runtime into the virtual address spaces of multiple processes. It promotes modularity in the large, by enabling programmers to write applications containing logically distinct translation units, each composed at a different level of abstraction or even in a different programming language altogether.

But perhaps equally important are applications of verified separate compilers to modular verification: proving a whole program correct by specifying and verifying its modules independently (with respect to the specifications of the other modules). Compiling verified modules with a semantics-preserving separate compiler results in correctness not only of each compiled module, but also of the target whole program linked and running on the machine.

In this paper, we present Compositional CompCert, a fully verified separate compiler from CompCert’s Clight language to x86 assembly. Each phase uses the exact same compilation function as CompCert 2.1, but a significantly strengthened specification that supports verified separate compilation of multi-module programs. In contrast, original CompCert’s specification is limited to whole programs and fails to account for general shared-memory interaction. That is, CompCert is not certified for calls to other modules, especially if two modules might access the same memory locations. We certify the correctness of such calls.

Compositional CompCert builds on work of Beringer et al. [2014], which showed how to adapt CompCert to support shared-memory interaction between a single compilation unit and its environment, under the restriction that the environment was not itself compiled. The technical innovations that enabled the adaptation to shared memory were twofold: First, a novel flavor of operational semantics, called interaction semantics (“core semantics” in Beringer et al. [2014]), for modeling a module’s interactions with its environments; and second, a new proof method, called logical simulation relations (LSRs), for proving correctness of compiler phases with respect to the interaction semantics interface. LSRs supported interaction reasoning between the source, intermediate, and target languages of an optimizing compiler by modeling all languages uniformly, as interaction semantics. LSRs also composed vertically, or transitively, which made it possible to break down the correctness proof of a multiphase compiler into
proves of the individual compiler phases. (Leroy’s original CompCert was also vertically compositional, but only because stack-allocated memory was not observable.)

The main deficiency of LSRs was that they did not compose horizontally: It was not possible, in general, to infer correct compilation of a whole program from the correct compilation of its modules. The reason was that LSRs, like CompCert’s original simulation relations, imposed assumptions (relies) on the evolution of memory over external function calls, such as “spilled local variables are unmodified,” but did not show that compiled code preserved the corresponding guarantees. This asymmetry between relies and guarantees made LSRs inapplicable whenever a module and its environment were both compiled—a situation that occurs not only when libraries are compiled, but also whenever there is a cyclic dependency between the modules of a program.

**Contributions.** In Compositional CompCert we overcome these difficulties, and achieve horizontal compositional, with the following innovations over previous work:

1. **Language-Independent Linking** is an extension of interaction semantics that gives the operational interpretation of horizontal composition. By abstracting from the details of how modules are implemented (or even in which language they are implemented), language-independent linking models the interactions of modules in different languages, even those with different calling conventions.

2. **Structured Simulations** refine LSRs to support the rely-guarantee relationship necessary for horizontal composition, while maintaining vertical composability. The key ingredients of structured simulations are: fine-grained *subjective invariants* on the state communicated between modules at runtime, and a *leakage* protocol that ensures that structured simulation proofs respect the reachability relation induced by C pointers.

Compositional CompCert’s top-level correctness theorem is a variant of contextual equivalence between source and compiled multilanguage programs in which contexts are specified semantically, as (nearly) arbitrary observations on the memory state at external call points. Contexts are not limited to programs in C or x86 but include mathematical relations expressed in Gallina.

In addition to sketching formal results that relate linking semantics, structured simulations, and contextual equivalence, we outline the instantiation of interaction semantics to the source and target languages of our compiler, Clight and x86, and present an overview of the adaptation of the relevant proofs of the CompCert phases.

All results in this paper have been proved formally in Coq, and are available, along with the Compositional CompCert compiler itself, on GitHub. Before proceeding with the technical details, we briefly discuss some key aspects of the above innovations.

1.1 **Key Ideas**

*Language-independent linking* provides a generic operator $\mathcal{L}$ for composing interaction semantics, independent of the language level of individual modules. Our statements of separate compilation and contextual equivalence (Sections 5 and 6) are *cross-language* in the sense that they apply even to the situation in which a multilanguage source program is compiled to a multilanguage target program.

Informally, imagine a source program $P_S$ consisting of source modules $S_1,\ldots,S_n$, each in a different language. Then if target modules $P_T = T_1\cdots T_n$ are shown related to $S_1\cdots S_n$ (e.g., by $n$ different structured simulations), the results of Sections 5 and 6 give us that the target “linked” program $P_T$ is contextually equivalent to the source “linked” program $P_S$ (under certain restrictions on the $S_i$ and $T_i$, explained in Section 3, for a strong semantic notion of program context. Of course, it’s not clear *a priori* what it means to link multilanguage programs, at least not in any operational sense; answering this question is the subject of the first part of Section 3. In principle, these results mean that the target modules $T_1\cdots T_n$ need not be generated by any particular compiler—our contextual equivalence results depend only on the existence of the simulations. In practice in Compositional CompCert, the $S_i$ are programs in Clight or hand-written assembly while the $T_i$ are x86 assembly programs.

**Structured simulations** evolve from LSRs by lifting the restriction to fixed environments, in two steps: First, we impose a rely-guarantee discipline (inspired in part by the rely-guarantee simulations used by Liang et al. (Liang et al. 2012) to prove contextual refinement of concurrent programs) on the interactions of program modules. The rely-guarantee discipline ensures that module compilation preserves the same properties that modules themselves assume about the behavior of external functions (those defined in other modules). This, in turn, makes it possible to implement external functions or libraries with code that is itself compiled, as in Figure 1. Second, we enrich the simulation relations with additional “ownership” data, which makes it possible to distinguish memory regions that are reorganized during compilation of distinct translation units. For example, the portion of the stack frame reserved for spilling during compilation of a function $A.f$ can be distinguished from the spill region reserved for a second function $B.g$, defined in a distinct translation unit $B$.

A key insight here is that the invariants that apply to distinct regions of memory—such as the regions reserved by the compiler for $A.f$’s and $B.g$’s spilled locals—are *subjective*: function $A.f$ can write to its own spilled locals but not to $B.g$’s, and vice versa for $B.g$ with respect to $A.f$’s spills. Structured simulations deal with this subjectivity by imposing an “us vs. them” discipline on compiler correctness invariants: Each structured simulation distinguishes the parts of the state that it controls (the “us”) from the parts of the state controlled by the environment (the “them”). This discipline is reminiscent in some ways of Ley-Wild and Nanevski’s subjective concurrent separation logic (SCSL) (Ley-Wild and Nanevski 2013), though here we apply a rely-guarantee discipline to the two-program invariants used to prove compiler correctness rather than to the verification of concurrent programs. To ensure that structured simulations validate contextual equivalences, and thus are reusable in many different program contexts, we make them parametric in nearly all state updates that can occur to the “them” portion of the state at external call points.

Another ingredient is a “leakage” protocol, which ensures that the views of the memory state imposed by the compiler invariants for different modules remain consistent. For example, when $A.f$ calls $B.g$ with arguments $\overline{v}$, we require that $A.f$’s compilation invariant “give up exclusive control” of all the memory regions reachable from $\overline{v}$ (i.e., following pointer chains rooted in $\overline{v}$). This condition represents the guarantee that, while later compilation stages of $A.f$ can still reorganize parts of the state reachable from $\overline{v}$ (e.g., by changing the order in which memory regions are allocated), they cannot remove these memory regions entirely (e.g., by dead code/memory analysis): the existence of the memory regions in question has been leaked irrevocably to the environment. Similarly, at external function return points, memory regions reachable from the return value are “leaked in” to the caller’s compilation invariant—representing the rely that these regions will never later be removed by compilation of the environment. Our language-independent linking semantics and contextual equivalence proof ensure that these conditions are in rely-guarantee relation.

https://github.com/PrincetonUniversity/compcomp
Interestingly, this leakage protocol bears much in common with the system-level semantics of Ghica and Tzevelekos (Ghica and Tzevelekos 2012). There, Ghica and Tzevelekos define a game semantics for a C-like language that avoids imposing so-called combinatorial (i.e., syntactic) restrictions on the moves of the environment, by applying what they call “epistemic” restrictions instead. These epistemic conditions, which parallel our leakage conditions, allow the environment to update the state in nearly any way as long as the updates are to memory regions leaked to the environment during previous interactions with the client program. This leads to a strong semantic notion of program context similar to the one we develop in Section 3. While Ghica and Tzevelekos were interested in modeling open C-like programs and their environments, not compiler correctness in this setting, we view the coincidence of our leakage conditions with their system-level semantics as evidence of the naturalness of our leakage protocol (Section 4).

Overview. We begin by reviewing interaction semantics, an operational model of shared-memory module and thread interaction. Section 2 employs interaction semantics to define the operational semantics of multilanguage linked programs, and semantic context equivalence. Then, we introduce structured simulations in Section 4. Section 5 presents the main theoretical results: vertical and horizontal compositionality, and context equivalence for Compcert. Section 6 describes the Compositional Compcert compiler itself, with a discussion of the effort required to port Compcert’s existing languages and proofs to structured simulations.

2. Interaction Semantics

Interaction semantics (called core semantics in (Beringer et al. 2014)) are a protocol-oriented operational semantics of thread interaction, for modeling both multithread concurrent and multimodule programs. Interaction semantics grew from the insight that interaction, between well-synchronized concurrent threads or between modules, can be viewed as occurring exclusively at external function call points, that is, via calls to functions declared in one module but defined in another. This gives a compiler or weakly consistent memory model the freedom to optimize between interaction points.

Here we give a brief overview of interaction semantics, which we build on in later sections to model language-independent linking. First, we describe the interaction semantics protocol. Then, we give representative encodings of interaction semantics, in this case, of Compcert’s C and x86 assembly languages. In Compositional Compcert, all language semantics are encoded in this way, as interaction semantics.

2.1 Protocol

In a multithread concurrent program, threads are spawned, take normal (i.e., unobservable) steps, yield at synchronization points (e.g., call to unlock), and eventually halt or (silently) diverge. The same protocol applies to module interactions: a C program is initialized by spawning a new “thread” with a function pointer to main. This sequential thread (which we sometimes call a core) takes normal, unobservable steps by evaluating main or by calling other internal functions defined in the same translation unit, “yields” to the environment by calling external functions defined in other modules, “blocks” until the external function returns, and halts or nomerminates just as a concurrent thread does. At external-call interaction points, nearly anything can happen: For example, the (shared) memory state might be updated arbitrarily by an external function.

Figure 2 summarizes the protocol. Each interaction semantics is parameterized by five types: $G$ is the type of global environment, $C$ is the type of internal, or “core” states. Core states can be instantiated to, e.g., the register file, instruction stream, and processor flags for a language like x86, or to local variable environment and control continuation for a higher-level language like C. $M$ is the type of memories. In the models of the CompCert languages that we employ in Compositional CompCert, $M$ is instantiated to mem, the type of CompCert memories. (In semantic models of program logics such as the Verified Software Toolchain’s Verifiable C (Appel et al. 2014), $M$ is instantiated to a step-indexed model of state used to model function pointer specifications and other higher-order features.) $F$ is the type of external function identifiers. $V$ is the type of values. $\mathcal{V}$ is usually just CompCert’s value type.

The five functions at the bottom of Figure 2, together with a few governing axioms (not shown), encode the interaction protocol described above. New cores are initialized with initial_core $g \ v \ \mathcal{V}$. The $v$ is a value, typically a function pointer, while $\mathcal{V}$ are the initial arguments to $v$, initial_core may fail with None when, e.g., the function spawned is not defined in the global environment $g$. at_external interrogates core state $c$ to determine whether $c$ is “blocked” at an external function call interaction point. When at_external succeeds, it does so with the name of the external function being called (of type $F$) and the arguments (of type list $V$). The after_external function is used to inject the return value of an external call into the calling at_external core state, producing a new core state as result, halted $c$ returns Some $v$ with return value $v$ if $c$ is halted, otherwise None. corestep gives the small-step internal transition relation of the interaction semantics. We use the syntax $ge \vdash c, \ m \rightarrow c', \ m'$ to denote this relation.

2.2 Examples: C&x86 Assembly

Figures 3 and 4 give the syntax of C and CompCert x86 assembly, the source and target language of Compositional CompCert, respectively. Both languages are adapted from CompCert’s original C and x86 and have straightforward operational semantics, which we do not present here (but see the code that accompanies this paper for the complete definitions). Here we focus on the adaptations required to turn these two languages into interaction semantics: First, we give the core, or internal, states for each language; then, we provide an overview of the definitions of the interface functions, e.g., at_external and after_external.
Figure 3. Syntax and semantics of Clight (excerpts). Continuations and core states appear only in the operational semantics.

Figure 4. Syntax and semantics of CompCert x86 assembly (excerpts). Core states appear only in the operational semantics. Int- 

floatness types τi used for value decoding) are int, float, long, or single.

Clight’s language of expressions, statements, internal function definitions, and external function declarations is given in the top half of Figure 3. Statements, core states, appear only in the operational semantics. We want to be able to model, at least abstractly, the interaction-semantics interface is straightforward. For example, here is the definition of Clight after_external:

\[
\begin{align*}
\text{after_external} & : \text{c} \in \text{orbits} \\
& \quad \text{case } \text{c} \text{ of CallState } f \Rightarrow \kappa \rightarrow \\
& \quad \quad \text{case } f \text{ of Internal } \rightarrow \text{ None} \\
& \quad \quad \quad \quad \text{External } idy \Rightarrow \tau \rightarrow \\
& \quad \quad \quad \quad \quad \text{case } \text{vopt} \text{ of } \\
& \quad \quad \quad \quad \quad \quad \text{None } \rightarrow \text{ Some } (\text{ReturnState } \text{vundef } \kappa) \\
& \quad \quad \quad \quad \quad \quad \text{Some } v \rightarrow \text{ Some } (\text{ReturnState } v \kappa) \\
& \quad \quad \quad \quad \text{ \_ } \rightarrow \text{ None}
\end{align*}
\]

First, we check whether \( c \) is a CallState, with continuation \( \kappa \). If it is, and the function that was being called was external, then we produce a ReturnState with return value \( \text{vundef} \) (whenever \( \text{vopt} \) was None) and \( v \) (whenever \( \text{vopt} \) was Some \( v \)). In all other cases, we just return None.

The definition of initial_core \( \text{c} e v \Rightarrow \) is simple, since function arguments are passed not on the stack but abstractly, without reference to memory: we check that \( v \) is a valid pointer to a defined function \( f \), check that the arguments \( \Rightarrow \) are defined and match \( f \)'s type signature, then introduce state CallState \( \Rightarrow \text{StateCL} f \text{ id } k \text{ \rho } \text{p} \text{c} \text{ ReturnState } v \kappa \) (or internal \( \Rightarrow \text{Kstate} \), sequential composition

\( \Rightarrow \text{Kloop } s \text{ k} \text{ \rho } \text{p} \text{c} \) loop continuation

\( \Rightarrow \text{Kswitch } k \) catch switch break

\( \Rightarrow \text{Kcall optid } fi \text{ \rho } \text{p} \text{c} \) catch function return

Core States

\( \Rightarrow \text{StateSMM } \text{r} \text{ s} \text{ } \text{rs} \) normal states

\( \Rightarrow \text{MarshallStateCore } \text{id } \Rightarrow \) marshall args. in

\( \Rightarrow \text{MarshallStateCoreOut } (\text{id } \Rightarrow r s) \Rightarrow \) marshall args. out

In instructions

\( \Rightarrow \text{MOVr } r i \text{ \text{MOVil } r i } \text{ moves}

\( \Rightarrow \text{JMP } l \text{ } \text{JMP } s \text{id } \text{ } \text{JMP } c \text{ cond } l \text{ moves}

\( \Rightarrow \text{CALL } s \text{id } \text{ } \text{CALL } R \text{id } \text{ RET } \text{ calls/return}

\( \Rightarrow \text{moves with conversion, integer arithmetic, etc.}

Core States

\( \Rightarrow \text{StateSMM } \text{r} \text{ s} \text{ } \text{rs} \) normal states

\( \Rightarrow \text{MarshallStateCore } \text{id } \Rightarrow \) marshall args. in

\( \Rightarrow \text{MarshallStateCoreOut } (\text{id } \Rightarrow r s) \Rightarrow \) marshall args. out

In instructions

\( \Rightarrow \text{MOVr } r i \text{ \text{MOVil } r i } \text{ moves}

\( \Rightarrow \text{JMP } l \text{ } \text{JMP } s \text{id } \text{ } \text{JMP } c \text{ cond } l \text{ moves}

\( \Rightarrow \text{CALL } s \text{id } \text{ } \text{CALL } R \text{id } \text{ RET } \text{ calls/return}

\( \Rightarrow \text{moves with conversion, integer arithmetic, etc.}

Core States

\( \Rightarrow \text{StateSMM } \text{r} \text{ s} \text{ } \text{rs} \) normal states

\( \Rightarrow \text{MarshallStateCore } \text{id } \Rightarrow \) marshall args. in

\( \Rightarrow \text{MarshallStateCoreOut } (\text{id } \Rightarrow r s) \Rightarrow \) marshall args. out

In instructions

\( \Rightarrow \text{MOVr } r i \text{ \text{MOVil } r i } \text{ moves}

\( \Rightarrow \text{JMP } l \text{ } \text{JMP } s \text{id } \text{ } \text{JMP } c \text{ cond } l \text{ moves}

\( \Rightarrow \text{CALL } s \text{id } \text{ } \text{CALL } R \text{id } \text{ RET } \text{ calls/return}

\( \Rightarrow \text{moves with conversion, integer arithmetic, etc.}

Core States

\( \Rightarrow \text{StateSMM } \text{r} \text{ s} \text{ } \text{rs} \) normal states

\( \Rightarrow \text{MarshallStateCore } \text{id } \Rightarrow \) marshall args. in

\( \Rightarrow \text{MarshallStateCoreOut } (\text{id } \Rightarrow r s) \Rightarrow \) marshall args. out

In instructions

\( \Rightarrow \text{MOVr } r i \text{ \text{MOVil } r i } \text{ moves}

\( \Rightarrow \text{JMP } l \text{ } \text{JMP } s \text{id } \text{ } \text{JMP } c \text{ cond } l \text{ moves}

\( \Rightarrow \text{CALL } s \text{id } \text{ } \text{CALL } R \text{id } \text{ RET } \text{ calls/return}

\( \Rightarrow \text{moves with conversion, integer arithmetic, etc.}
state MarshallState\textsubscript{in}, id\textsubscript{f} \triangleright \mathbb{T}, which immediately steps to a running State. As a side effect of this step, however, we allocate a “dummy” stack frame in memory in which we store the incoming arguments \(\mathbb{T}^i\), in right-to-left cdecl order as expected by CompCert and gcc. (MarshallState\textsubscript{out} performs the symmetric step of marshalling arguments out of memory.)

Concretely, for x86 modules all sharing the same calling convention, this modeling step does not occur on a real machine (nor does the compiler output any “marshalling” code). But by sticking to the abstract “calling convention” imposed by \texttt{initial\_core}, in which values are passed abstractly instead of in memory and registers according to a particular calling convention, we gain flexibility to model the interactions of modules in a wide variety of languages, not only Clight and x86 but also x86 modules following different calling conventions, such as, e.g., cdecl and Microsoft fastcall.

### 3. Linking and Contextual Equivalence

In the previous section, we introduced interaction semantics as a means of interpreting the behavior of isolated modules. In this section, we define an abstract operator \(\mathcal{L}brack\{[S_0], [S_1], \ldots, [S_{N-1}]\}\) over interaction semantics that defines the linked behavior of a set of interacting modules, as given by a multimodule program \(P = S_0, S_1, \ldots, S_{N-1}\).

As input, \(\mathcal{L}\) takes \(N\) interaction semantics, each with (perhaps) a different global environment and core state type (i.e., modules programmed in perhaps different languages). The output of \(\mathcal{L}\) is a new interaction semantics \([P]\) = \(\mathcal{L} \rbrack\{[S_0], [S_1], \ldots, [S_{N-1}]\}\) that models the execution of the linked program by maintaining as its own core state a (heterogeneous) stack of the modules’ core states. Each “frame” on the stack corresponds to a runtime invocation of one of the modules in the program. Cross-module function calls result in new cores being pushed onto the stack (initialized via \texttt{initial\_core}); returning from such a function pops the top core from the stack and injects the return value into the state of the caller, using \texttt{after\_external}.

The modules \(S_i\) are written in different languages, whose states may have different (Coq) types. In order to treat these modules uniformly in \(\mathcal{L}\), we wrap their interaction semantics by existentially quantifying over the core state types of each module, an operation we encapsulate in the type Modsem.

\[
\text{Modsem} \triangleq \begin{cases} 
F, V, C & : \text{Type} \\
\text{ge} & : \text{Genv F V} \\
\text{sem} & : \text{Semantics (Genv F V)} \text{ C mem}
\end{cases}
\]

In this dependently typed record, the types of \text{ge} and \text{sem} depend on \(F, V, \) and \(C\). This module is written in programming language F (e.g., Clight or x86), whose global variables have type-specification language \(V\) (e.g., Clight types or unit); and whose core states have type \(C\) (e.g., Clight nonaddressable locals and control stack, or x86 register bank). We also existentially bind the global environment \(g\) that was statically initialized for this module. It maps addresses to global variables and function-bodies \(\triangleright\).

The final component is \text{sem}, an interaction semantics. It defines the interface functions \texttt{initial\_core}, \texttt{at\_external}, etc., as well as a step relation \(\mathbb{G} \vdash c, m \rightarrow c', m'\). Modules in the same language will typically have identical \(\vdash \rightarrow \rightarrow \rightarrow \) relations, specialized by different \text{ge} components that map disjoint sets of addresses to internal function bodies (as opposed to external function declarations). In what follows, we use \(\triangleright\) to refer interchangeably to the interaction semantics of modules and their Modsem wrappers.

The output of \(\mathcal{L}\) is an interaction semantics in the LinkedState “language.” LinkedState is parameterized by \texttt{modules}, a map from module indices in the range \([0, N]\) to module semantics, where \(N\) is the (nonzero) number of translation units in the program.

\[
\begin{align*}
\text{Core} & \left( N : \text{pos} \right) \left( \text{modules} : I_N \rightarrow \text{Modsem} \right) \triangleq \\
& \begin{cases} 
\text{id} & : I_N \\
\text{core} & : C \left( \text{modules \ idx} \right)
\end{cases}
\end{align*}
\]

Core models the runtime state of a sequential execution thread. \(I_N\) is the (dependent) type of integers in range \([0, N]\). The \text{idx} of \(C\) is the index of the module from which the core was initialized.

The runtime state of a linked program is then:

\[
\begin{align*}
\text{LinkedState} & \left( N : \text{pos} \right) \left( \text{modules} : I_N \rightarrow \text{Modsem} \right) \triangleq \\
& \begin{cases} 
\text{plt} & : \text{ident} \rightarrow \text{option I}_N \\
\text{stack} & : \text{Stack} \left( \text{Core N modules} \right)
\end{cases}
\end{align*}
\]

The two fields of LinkedState are: the procedure linkage table \texttt{plt}—mapping function names (type \texttt{ident}) to the indices of the modules in which the functions are defined, if any (option \(I_N\))— and a stack of cores. We model the \texttt{plt} as a field in the LinkedState record, as opposed to deriving it from \(N\) and \texttt{modules}, to retain flexibility to do dynamic linking in the future. The stack is always nonempty; all cores except the topmost one are \texttt{at\_external} (\(\forall c \in \left(\text{pop stack}\right)\), \texttt{at\_external} \(c\) = \texttt{Some \_}.)

Figure 5 gives the step relation. There are three rules. The \texttt{Step} rule deals with the case in which the topmost core on the call stack \((c = \texttt{peek l\_stack})\) takes a normal internal step \((ge_c \models c, m \rightarrow c', m')\). \(ge_c\) is the global environment associated with the module from which \(c\) was initialized. In this case, we just propagate the new core state \(c'\) and memory \(m'\) to the result state of the overall linking judgment. The notation \(\mathcal{L} \triangleright \text{with} \{ \text{stack} := \text{push c'} \left( \text{pop l\_stack} \right) \}\) updates the topmost core state on the stack. For readability, we elide the operations required to propagate the \text{idx} field of Core records.

The second rule, \texttt{Call}, handles the case in which the topmost core on the stack is \texttt{at\_external} (\(\exists \text{some \ (idi, \mathbb{T})}\) making a cross-module function call. In this case, we use the \texttt{initial\_core} function of the module semantics that defines function \texttt{idi} (\(l, \text{plt}, idf = \text{Some idx}\)) to initialize a new core state to handle the function call \((\text{initial\_core \ (modules \ idx)} \left( \text{Vptr \ by} \mathbb{T} \right) = \text{Some c'}\)). The core \(c'\) is then pushed onto the stack \((\text{with} \{ \text{stack} := \text{push c' l\_stack} \})\) to become the new running core.

The \texttt{Return} rule models external function returns. Here, the core state \(c\) is halted with return value \(v\) \((l \text{ with} \{ \text{stack} := \text{push c' l\_stack} \})\). To resume execution, we use the \texttt{after\_external} function exposed by the caller’s semantics \(c'' = \text{peek \ (pop l\_stack)}\) to inject the return value \(v\) \((\text{after\_external} \left( \text{Some v} \right) c'' = \).

\footnote{All the inputs to \(\mathcal{L}\) must have \text{ge} functions that map exactly the same global addresses (modules that fail to declare some unused external global variables or functions can always be made to do so, by safety monotonicity).}
ge \equiv c, m \implies c', m'

\begin{align*}
& c = \text{peek l.stack} \quad ge_c \equiv c, m \implies c', m' \\
& ge \equiv l, m \implies l \text{ with } \{ \text{stack := push c' (pop l.stack)} \}, m^* \quad \text{(STEP)}
\end{align*}

\begin{align*}
& c = \text{peek l.stack} \quad \text{at external } c = \text{Some } (id_y, \overline{v}) \\
& \text{l.plt idy } = \text{Some idx} \quad ge idy = \text{Some } b_t \\
& \text{initial core (modules idx)} (\text{Vptr } b_t) = \text{Some } c' \\
& ge \equiv l, m \implies l \text{ with } \{ \text{stack := push c' l.stack} \}, m \\
& \text{(CALL)}
\end{align*}

\begin{align*}
& \text{size l.stack } > 1 \quad e = \text{peek l.stack} \\
& \text{halted } e = \text{Some } v \quad c' = \text{peek (pop l.stack)} \\
& \text{after external (Some } v) \quad c' = \text{Some } c'' \\
& ge \equiv l, m \implies l \text{ with } \{ \text{stack := push c' (pop (pop l.stack))} \}, m \\
& \text{(RETURN)}
\end{align*}

\textbf{Definition 1} (Contextual Equivalence).

\[ P_S \sim P_T \triangleq \forall C. \ L(C, [P_S]) \downarrow \iff L(C, [P_T]) \downarrow \]

The context \( C \) observes the state of memory (and the arguments to external calls) when the program interacts with the environment. To distinguish \( P_S \) and \( P_T \), \( C \) can, e.g., get stuck (as opposed to safely terminating) at one of these interaction points if the memory state and arguments fail to satisfy a specified predicate.

\section{Structured Simulations}

Section\textsuperscript{4} showed how to define the semantics of open multimodule programs, and what contextual equivalence meant in that setting. Now we show how to prove contextual equivalences for CompCert. We briefly review logical simulation relations (Beringer et al. 2014), and then show our new enhancement, structured simulations.

LSRs established compiler correctness by showing that compilation preserved the protocol structure of interaction semantics, using CompCert’s original match relations \( \sim \), with memory injections \( f \) to relate source and target states. For internal execution steps, they followed CompCert’s forward simulation proofs. For external calls, they asserted that the two modules call the same function with related arguments, and that the simulation relation is reestablished at return points whenever the environment provides related return values, subject to a few constraints on how memory could evolve over the external calls.

The two most crucial of these constraints were that (1) in the source execution, external calls did not modify any memory region the compiler wished to remove\textsuperscript{4} and that (2) in the target execution, external calls did not modify locations that were unreachable from the source memory (an “unreachable” target location is one that does not correspond to a readable location in the source memory). Condition (2), in particular, enabled the proof of compiler phases such as spilling, which introduces new unreachable spill locations into a target program’s stack frames. A deficiency of CompCert’s simulation proofs and of LSRs was that they assumed conditions (1) and (2) at external calls, but did not prove that these properties were preserved by compilation.

Directly imposing constraints (1) and (2) onto the simulation clauses for internal steps does not work, however. A compiled function should be allowed to write to its own spill locations—just not to those of its caller.

To capture the difference in perspective between caller and callee, we make three adjustments to the LSR framework. First, to index the match relation \( \sim \), we use \textit{structured injections} \( \mu \) instead of CompCert’s original injections \( f \). The additional structure in \( \mu \) maintains the block-level ownership information necessary to tell a callee’s (or other environmental) blocks apart from caller blocks. Second, we decorate the internal step relation of interaction semantics with \textit{modification effects} \( E \) such that locations not contained in \( E \) are guaranteed not to be modified \( (i.e \text{ written to, or freed by}) \) the step in question. Third, we impose a \textit{restriction} axiom on \( \sim \) that ensures compilation invariants depend only on memory regions either allocated by the module being compiled, or leaked to it via pointers returned from external calls.

The details are as follows.

\textbf{Structural Injections.} In CompCert, memory is allocated in regions, or \textit{blocks}. Within each block, memory bytes are addressed using integer offsets (pointer arithmetic is allowed only within blocks). CompCert’s memory injections \( f : \text{block} \to \)}
Ownership $\triangleq$ Priv | Pub | Frgn | Invis | None

$\mu \in \text{StructuredInjection} : \text{Type} \triangleq$

- $\text{own}_S : \text{block} \rightarrow \text{Ownership}$
- $\text{own}_T : \text{block} \rightarrow \text{Ownership}$
- $\text{f}_\text{pub} : \text{block} \rightarrow \text{option} (\text{block} \times \mathbb{Z})$
- $\text{f}_\text{then} : \text{block} \rightarrow \text{option} (\text{block} \times \mathbb{Z})$
- $\text{owned}, \triangleq \{ b \mid \text{own}(b) \in \{ \text{Priv}, \text{Pub} \} \}$, $i \in \{S, T\}$
- $\text{shared}, \triangleq \{ b \mid \text{own}(b) \in \{ \text{Pub}, \text{Frgn} \} \}$, $i \in \{S, T\}$
- $\text{vis}, \triangleq \text{owned}, \cup \text{shared}$, $i \in \{S, T\}$

$f \mid X \triangleq \lambda b, i \mapsto \text{if } b \in X \text{ then } f \text{ else } \text{None}$

$\mu \mid X \triangleq \mu \text{ with } \{ f_\text{pub} := f_\text{pub} \mid X \}, \{ f_\text{then} := f_\text{then} \mid X \}$

**Figure 7.** Structured injections.

option $(\text{block} \times \mathbb{Z})$ source relate and target memories. For example, the memory injection that maps $b$ to Some $(b', \delta)$ associates source address $(b, 8)$ with target address $(b', 8 + \delta)$.

Structured injections $\mu$ (Figure 7) strengthen CompCert’s memory injection relations with additional ownership structure. They have four components: Two ownership functions $\text{own}_S, \text{own}_T : \text{block} \rightarrow \text{Ownership}$, which map blocks (in the source and target memories, respectively, of a related pair of program states) to values of an inductive Ownership type; and two CompCert-style memory injections: $f_\text{pub}$ and $f_\text{then}$ record the source–target mapping of blocks that were allocated by the current module; $\text{f}_\text{pub}$ maps external blocks (those allocated by other modules).

The Ownership modes are: Priv, for memory regions (blocks) allocated by the module being compiled but which haven’t been leaked to the environment; Pub, for allocated blocks that have been leaked at a previous interaction point; Frgn, for foreign blocks leaked into $\mu$ at external calls; Invis, for blocks that have been allocated (by another module) but not leaked in; and None for blocks that may not yet have been allocated. A block is (locally) owned by $\mu$ in the source or target memory when $\text{own}_S(b)$ (resp. $\text{own}_T(b)$) is either Pub or Priv. External blocks in source and target are those mapped by $\text{own}_S, \text{own}_T$ to Frgn or Invis. Likewise, a block is shared if its ownership is either Pub or Frgn. The visible source blocks of $\mu$ are those in the set $\text{vis} \triangleq \text{owned} \cup \text{shared}$ (and likewise for $\text{vis}_T$). We use notation $\text{foreign}(S, T)$ and $\text{public}(S, T)$ to denote the blocks with foreign and public ownership, respectively.

We track ownership of blocks, rather than ownership byte-by-byte, because the CompCert languages and memory model permit pointer arithmetic within blocks. Once a location within a block has been made public, the whole block is made public as well.

Complementing the data in Figure 7 are axioms that ensure proper interaction of ownership, leakage, and compilation. These axioms (not shown) enforce that $\text{f}_\text{pub}$ and $\text{f}_\text{then}$ (1) operate exclusively on blocks of appropriate ownership (i.e. $\text{f}_\text{pub}$ only maps owned blocks, to owned blocks, and similarly for $\text{f}_\text{then}$ and external blocks); and (2) are total on their portion of shared blocks: $\text{f}_\text{pub}$ must map all Pub blocks, and must map them to Pub blocks, and similarly for $\text{f}_\text{then}$ and Frgn. The result is that blocks which have been leaked to/from the environment in one compilation stage cannot be removed by later stages.

At interaction points between a module and its environment, we adjust the structured injections so that (at these points) the shared regions are closed under pointer arithmetic and dereferencing (there are no pointers from the shared to the nonshared region). We maintain as an additional invariant that the source visible set $\text{vis}_S$ is always closed under pointer dereferencing and pointer arithmetic.

**Simulation Structures.** Figure 8 presents the two core clauses of structured simulations $\preceq$, those for internal (i.e. unobservable) steps (Internal Steps) and for external interactions with the environment (External Steps) (the clauses for initial_core, at_external, and halted are not shown). In the figure, $\sim_\mu$ is existentially quantified. There is no single definition of $\sim_\mu$, but instead, one per compilation phase—Figure 8 defines the laws that each such $\sim_\mu$ relation must satisfy.

The structure of the internal diagram (which is simplified in the figure to elide stuttering source steps) is familiar from traditional forward simulation proofs: Assume we are in matching initial states $(c, m) \sim_\mu (d, tm)$ and we take a source step $ge_S \vdash c, m \rightarrow c', m'$ with effect $E_S$. Then there exists a matching $d', tm'$, and Kripke-extended structured injection $\mu$ such that $ge_T \vdash d, tm \rightarrow d', tm'$ and $(c', m') \sim_\mu (d', tm')$. Clause (1) (Kripke extension, $\mu \subseteq \mu'$) says that $\mu'$ may map more owned blocks than $\mu$ (in order to deal with allocations) but otherwise is equal to $\mu$. Clauses (2) and (3) are side conditions that are not important for understanding the key ideas.

Clause (6) is the guarantee condition. Clause (6a) asserts that the target effects $E_T$ are contained in $\text{vis}_T \mu$, assuming that $E_S \subseteq \text{vis}_S \mu$. In most Compositional CompCert phases, the $\preceq_\mu$ relation is equality up to the injection of memory regions, the addition, removal, and merging of certain memory regions, and invariants on private memory regions. As in original CompCert, we say that nonpointer values such as integers are related only if they are actually equal.
Let $v \triangleq \mu$. In other words, the compiler preserves the property of “writing to, and freeing, only visible locations.”

Clause (6b) guarantees that writes to (and frees of) memory locations in the target that are not owned by $\mu$ ($b_i \notin \text{owned}_T\mu$) can be “tracked back” to corresponding writes and frees in the source ($3b_i\delta, f_{\text{then}}(b_i) = \text{Some} \ (b_i, \delta)$ and $(b_i, z_i - \delta) \in E_S$). Writes/frees of locations in blocks owned by the module being compiled are always permitted, which enables the compiler to introduce reloading code (for spilled variables) or to add function prologue/epilogue code that saves/restores callee-save registers.

The $E_S$ and $E_T$ that appear in clause (5) and in step judgments are effect annotations. For example, $E_S(c, m) \triangleq s, c, m \in \text{blocksOf } c', m', \text{ means: configuration } c, m \text{ steps to } c', m', \text{ writing to or freeing exactly the locations } E_S. \text{ Locations not contained in this set are guaranteed not to be modified.}$

We state these “does not modify” guarantees intentionally in this way, as effect annotations, in order to prove vertical compositional. The problem with a more extensional interpretation of effects (e.g., as input–output “unchanged on” conditions) is that effects no longer “decompose”: If a program takes two steps, from $m$ to $m''$ with effect set $E_1$ and from $m''$ to $m'''$ with effect set $E_2$, with overall extensional effect $E$, it may be the case that $E_1 \cup E_2 \not\subseteq E$ if, for example, the second step restored a value that was overwritten by the first step. Decomposition is used in the internal step case of the proof that structured simulations compose vertically.

We track only write and free effects, and not read effects, because compiler correctness invariants do not, in general, depend on which locations another module merely reads. Proving more general program refinements (e.g., between multiple implementations of an ADT), or that the compiler does not introduce additional memory reads—a property useful in security contexts—would most likely require a generalization to read effects.

The external step diagram occupies the bottom half of Figure 9. It relates an at_external source–target configuration pair $(c', m') \sim_{\mu'} \langle d, tm' \rangle$ with the after_external configuration pair $(c', m') \sim \mu' \langle d', tm' \rangle$ that results from making an external call. The basic premise is: For any source–target return values $v_i, v_i$, return memories $m'$ and $tm'$, and structured injection $\nu'$ satisfying the listed conditions, it’s possible to inject $v_i$ and $v_i$ into states $c$ and $d$, resulting in the new states $c'$ and $d'$ which match in $\mu'$, $m'$, and $tm'$ $(c', m') \sim_{\mu'} \langle d', tm' \rangle$. The $\nu \triangleq \text{then } \nu'$ is dual to the $\nu_\text{out}$ condition used in the internal step diagram. It says that $\nu'$ may map more external blocks than $\nu$—in order to deal with allocations performed by the environment—but otherwise is equal to $\nu$. The other non-bolded conditions are adapted from CompCert, and follow in our Coq proofs directly from symmetric conditions on the match-state relation and the internal step diagram.

The conditions listed in bold together compose the structured simulation rely. The predicate unchanged_on $U \ m \ m'$ specifies that memories $m$ and $m'$ are equal (same contents and permissions) at the locations in set $U$. In the source execution, we use unchanged_on $(\{ b, z \} \mid \text{own}_S\nu \ b \in \text{Priv}) \ m \ m'$ to ensure that $m$ and $m'$ are equal at locations in the private blocks of the injection $\nu$, which is built from $\mu$ by updating leakage information as described below. The target-execution condition unchanged_on $(\text{local_out_of_reach } \nu \ m) \ tm \ tm'$ says that $tm$ and $tm'$ are equal at owned target locations that either (1) do not correspond to readable source locations, or (2) are mapped from private source locations. By using unchanged_on here, we stipulate the nonmodification conditions of the rely extensionally.

The structured injection $\nu$ is built from $\mu$—the injection that originally related at_external states $(c, m) \sim_{\mu} \langle d, tm \rangle$—using the leak_out function depicted graphically in Figure 9 and defined in Figure 10. The idea is: leak_out “leaks” to the public (other modules) blocks that are reachable by following pointer paths either from the arguments $\mu_S$ to the external call (blocksOf $\mu_S$) or from blocks that were previously shared (shared $\mu$). This is a consistency condition: It says that structured simulations may not assume anything about the contents of leaked blocks (the unchanged_on conditions that form the rely satisfied by the environment apply only to private blocks). The functions reach and REACH defined at the top of Figure 10 calculate the transitive closure of the points-to relation on CompCert memories. In the definition of leak_out, we use the auxiliary function export to update the ownership functions of an injection $\mu$ to map blocks in the reachable set toPub.
The leak\_in function used to define $\mu'$ at the end of the external step diagram plays a role analogous to that of leak\_out, except that here, we are leaking into $\mu'$ new foreign blocks reachable from the return value $v_0$ of the external call. Likewise, the import function is almost equivalent to export, except that it updates the ownership functions of a structured function to map the block set $B$ to $Frgn$ as opposed to Pub.

**Restriction.** The final consistency condition is: the simulation relation $\sim_\mu$ is independent of the Inv\_vis (and None) blocks. This condition is used, e.g., in the proof of vertical composition (transitivity, Theorem 1). Technically, we enforce Inv\_vis-independence by requiring that $\sim_\mu$ be closed under restrictions to reach-closed supersets of the visible blocks, an operation defined in Figure 7 as $\langle \mu | X \rangle$ (with $X$ a block set), $\mu | X$ denotes the structured injection obtained by restricting the maps $f_\mu$ and $f_{\hor}$ to the domain $X$. The closure condition says that if $(c, m) \sim_\mu (d, tm)$, then $(c, m) \sim_\mu (d, tm)$ for any block set $X$ that contains at least $vis_\mu$ and is closed under pointer dereferencing and arithmetic in $m$.

5. **Main Results: Compositionality and Contextual Equivalence**

**Vertical Composition (Transitivity).** One can compose compiler phases (Figure 13). The proof that structured simulations compose vertically follows the same outline as that of LSRs (Beringer et al. 2014). As discussed in Section 4, the proof of transitivity of the internal-step diagram is tightly dependent on our treatment of effect annotations. Proving transitivity of the external-call clause (lower half of Figure 8) requires the construction of an interpolating function over the return address, which in turn requires a structured simulation from $S_1$ to $L_3$, then there exists a structured simulation $L_2 \leq L_3$ from $L_2$ to $L_3$. Since $L_2$ is closed on its own.

**Horizontal Composition (Linking).** The second kind of compositionality is horizontal: We would like to know that composing the simulation relations established by independently compiling the modules in a program results in an overall simulation between the (linked) multimodule source and target programs. We give the theorem statement first, then explain some of the subtleties, in particular, the restriction to reach-closed source semantics, which enforces the single-program conditions corresponding to the structured simulation guarantees of Section 4 and to valid target semantics (a technical property related to the CompCert memory model, explained below).

**Theorem 2 (Linking).**

- If $P_\mu = S_0, S_1, \ldots, S_{N-1}$ is a multimodule program with $N$ translation units, each of which is reach-closed, and $P_\mu$ is compiled to $P = T_0, T_1, \ldots, T_{N-1}$ (possibly by $N$ different compilation functions) such that $|S_i| \leq |T_i|$ for each source–target pair, and each of the $T_i$ is valid, then
- there is a simulation relation $L([P_\mu]) \leq L([P_T])$ between the source and target programs that result from linking the $S_i$ and independently linking the $T_i$.

The $\leq$ in the theorem denotes forward simulation on whole programs. Whole program simulations are as in Figure 8 but with out clauses (1-3) in the internal step diagram, without clauses for at\_external and after\_external, and without effects. As Corollary 1 will show, establishing $\leq$ is sufficient for proving contextual equivalence of open multimodule programs. A valid semantics is one that never stores invalid pointers into memory. Invalid pointers, in CompCert parlance, are those that refer to memory regions that have not yet been allocated (freed pointers are never invalid). All CompCert x86 programs are valid in this way. Validity is required, in the proof of Theorem 3 to maintain the invariant that the set of valid target blocks is reach-closed.

**Reach-Closed Semantics.** The restriction to reach-closed semantics (Figure 11) is best motivated with an example. Consider the program:

```c
//Module A
void g(void);.gblglob .g
void h(int x) (}; _g:
void f(void) { pushl %ebp
int a=0;movl %ebp, %ebp
if (a) (h(a));}movl 42, (0x28ac1c)
g(a);popl %ebp
void main(void)(f());} ret
```

in which $A.f$ calls (assembly function) $B.g$, passing no arguments. The strange bit is $B.g$: All it does is write the value 42 into memory at address $0x28ac1c$, which just happens to be the address at which local variable $a$ is allocated on the stack in Windows.

Now imagine we compile module $A$ through a compiler phase like dead code elimination, which results in $a$ (which was previously addressed in dead code $if\ (a)\ (h(a))$ and therefore stack-allocated) being removed from memory. Since $0x28ac1c$ is a’s address before dead code elimination, the unoptimized program above does not get stuck (the write succeeds, to location $a$, without significant effect). After optimization, the program will fail, probably because the write to $a$ overwrites the return address stored in $g$’s stack frame. (Incidentally, this program succeeds when compiled with $gcc -00$ but seg-faults under $gcc -01$. This is not a bug in $gcc$; instead, it is evidence that $gcc$, developers agree with us that $Module B$ is an ill-formed program context.)

One might object that $if\ (a)\ (h(a))$ is actually not dead code, because it results in a being stack-allocated, which in turn results in the safe execution of the (admittedly contrived) overall program. But that way madness lies. The point of a compositional compiler is to enable local modular compilation, which should depend only on translation-unit-local analyses. Correctness of optimizations like dead code elimination should be independent of the larger program context in which a module is executed.

The challenge, then, is coming up with a characterization of the source modules $S_0, S_1, \ldots, S_{N-1}$ that does admit linking as in Theorem 2. We do this in general, for arbitrary interaction semantics, by observing that the write to $0x28ac1c$ is ill-formed not because it goes wrong (though it will lead to going wrong in most program contexts) but because it’s a write to a location that the assembly program shouldn’t have known about in the first place. Put another way, address $0x28ac1c$ was not reachable via pointer arithmetic either from $g$’s initial arguments, from global variables, or from the return values of external calls $g$ may have made previously. This condition—no writes or frees to locations that are not “visible”—is the analogue of the $ES_\subseteq vis\_\mu$ in clause (6) of Figure 8 but stated as a single-program property, independent of any particular structured interaction $\mu$. We formalize the notion of a semantics that respects this characterization of visible locations as an extension of interaction semantics called reach-closed semantics, defined by the existence of an invariant $R$ satisfying the laws in Figure 11.

From the perspective of compiler correctness proofs, the
Reach-Closed Invariant
\[ \mathcal{R} : C \rightarrow \text{mem} \rightarrow \text{set block} \rightarrow \text{Prop} \]

Reach-Closed Initial Core
\[
\text{initial\_core } ge \ v \ \vdash = \text{Some } c \rightarrow \forall m. \ c \in m \ (\text{blocksOf } \vdash)
\]

Reach-Closed Step
\[ \text{roots } (ge : G) \ (B : \text{set block}) \triangleq \text{globalBlocks } ge \cup B \]

\[
\begin{align*}
\mathcal{R} & \in \text{REACH } m \ (\text{roots } ge B) \\
(1) & \ E \subseteq \text{REACH } m \ (\text{roots } ge B) \\
(2) & \mathcal{R} \ c m B' (\text{REACH } m' (\text{freshblks } m m') \\
& \cup \text{REACH } m (\text{roots } ge B))
\end{align*}
\]

Reach-Closed After External
\[ \begin{align*}
\mathcal{R} & \in \text{REACH } m B \\
\text{at\_external } c & = \text{Some } (id_f, \overrightarrow{v}) \\
\text{after\_external } v_{opt} c & = \text{Some } c' \\
\text{let } B' & \triangleq \text{case } v_{opt} \text{ of } \\
| \text{None} & \rightarrow B \\
| \text{Some } v & \rightarrow \text{blocksOf } (v :: \text{nil}) \cup B
\end{align*} \]

\[ \text{in } \mathcal{R} \ c m B' \]

Figure 11. Reach-closed semantics maintain an additional internal invariant \( \mathcal{R} \) on states \( c \), memories \( m \), and block sets \( B \) that satisfies the laws above. The definitions are parameterized by types \( G \) and \( C \), by a global environment \( ge : G \), and by an interaction semantics type \( \text{Semantics } G \ C \text{ mem} \) that defines step relation \( \rightarrow \) and functions \( \text{after\_external} \) and \( \text{initial\_core} \) (at\_external and halted are elided).

Restriction to reach-closed contexts is what enables program transformations: It would be unsafe, for example, to remove a dead memory allocation if the larger program context depended on it, as in the example program above.

The \( \mathcal{R} \) invariant of reach-closed semantics quantifies over: Core states of the argument semantics \( c : C \), the memory \( m : \text{mem} \), and a set \( B \) that records the blocks exposed to the semantics at interaction points (via pointers in the argument list, in the return values of external function calls, and by local allocation).

We use the notation \( \text{sem} \), for reach-closed semantics, to denote a semantics \( \text{sem} \) that exhibits such an \( \mathcal{R} \).

The roots of a block set \( B \) and global environment \( ge \) are the union of \( B \) and the global blocks of \( ge \). The operative conditions of reach-closed semantics are those that characterize the reach-closed step relation (clauses 1 and 2). In particular, clause (1), which ensures that reach-closed semantics satisfy the structured simulation guarantees of Section 4, instruments the step relation of the underlying semantics with the additional condition that the effects \( E \) produced by the step above clause (1) are a subset of the locations reachable in \( m \) from the current roots.

Clause (2) asserts that the invariant can be reestablished after the step for: the blocks reachable (in \( m' \)) from newly allocated blocks (freshblks \( m m' \)). If any, as well as from the blocks that were originally reachable in \( m \) (REACH \( m \) (roots \( ge c ) \)). This last condition ensures that the reachable set grows monotonically at each step, by not “forgetting” locations that were previously reachable.

The other interface laws modify \( B \) as specified above. For example, the clause for \( \text{after\_external} \) asserts that \( \mathcal{R} \) can be reestablished for \( B' \) equal to \( B \) union the blocks exposed by the return values of external calls (blocksOf \( v :: \text{nil} \)). \text{initial\_core} asserts that the invariant can be established initially, with \( B \) equal to the blocks exposed in the initial arguments.

As a corollary of Theorem 2, we get the following contextual equivalence result when the source modules are reach-closed, stated in terms of a variation of Definition 1 in which contexts satisfy a few additional properties.

Definition 2 (Reach-Closed Contextual Equivalence).
\[
P_S \sim_{\text{rc}} P_T \triangleq \forall C. \ C \subseteq C \land \det C \land \text{valid } C \land \text{safe } L(C, [P_S]) \rightarrow L(C, [P_T]) \equiv_\uplus \text{L}(C, [P_S]) \equiv_\uplus \text{L}(C, [P_T])
\]

Corollary 1 (Simulation Implies Contextual Equivalence). Let

- \( P_S = S_0, S_1, \ldots, S_{N-1} \); and
- \( P_T = T_0, T_1, \ldots, T_{N-1} \)

for reach-closed source modules \( S_0, S_1, \ldots, S_{N-1} \) and valid deterministic target modules \( T_0, T_1, \ldots, T_{N-1} \). If for each \( i \), \( [S_i] \subseteq [T_i] \), then \( P_S \sim_{\text{rc}} P_T \).

In the above, we assume closing contexts \( C \) (those that do not themselves call external functions not defined by any of the modules; callbacks into \( P_S \) and \( P_T \) are permitted). \( C \) must also be valid. Safety of the source linked program and determinism of the target modules are required to prove the backward direction of the equivalence (the forward direction holds without these assumptions). The \( C \subseteq C \) condition says that \( C \) commutes with memory injections: If \( C \) is initialized twice with injected arguments, both executions either go wrong, nonterminate, or equiterminate with injected results. Although this condition follows directly from the form of Theorem 2, it is strongly motivated: We should not allow contexts to distinguish source and target programs based solely on objective renamings of memory blocks exposed to the context (pointer arithmetic is not allowed between blocks, only within blocks). The consistency conditions on structured injections and simulations that we described in Section 4 mean that in the proof of \( C \subseteq C \), the context may assume that all public blocks leaked by the program are mapped from source to target (they are never removed during compilation of the program).

Nor is the reach closure condition imposed above an unrealistic proof obligation. One can show, for example, that all Clight programs satisfy the restrictions imposed in Figure 11.

Theorem 3 (Safe Clight Programs are Reach-Closed). There exists an \( \mathcal{R} \), specialized to Clight states \( c \) and the Clight step relation, that satisfies the laws given in Figure 12.

The proof of this (perhaps counterintuitive) theorem relies on the fact that Clight programs never fabricate nonnull pointers, e.g., by casting an integer to a pointer and then dereferencing it. (Even in standard C, casting an integer to a pointer, or vice versa, is only implementation defined, except when the pointer is null. See, e.g., the C11 standard [11][2011].)

The main difficulty in proving Theorem 2 and Corollary 1 is in devising a simulation invariant to relate the stacks-of-cores runtime states of the linked programs \( P_S \) and \( P_T \). The situation is presented schematically in Figure 12. In the source linked program, we have a stack of core states, growing downwards, with \( c \) in callee position with respect to a (direct or indirect) caller core \( c_0 \), which may be implemented in a different language. We must relate this stack of cores to the corresponding stack in the target linked program. We use \( \mu \) to denote the structured simulation that relates the callees \( c \) and \( d \), and \( v \) to denote the injection that relates callers \( c_0 \) and \( d_0 \). For simplicity, we elide the memories (for callers, the memory at the call point is existentially quantified). A callee core may be a callee with respect to another callee higher on the callstack.
Figure 12. Schematic representation of the stacks-of-cores linking invariant. The inner white boxes are core states. Source core $c$ and target core $d$ are callees at the bottom of the LinkedState callstack, related by structured injection $\mu$ (memory is elided). Cores $c_0$ and $d_0$ are caller cores related by $\nu$.

The key rely–guarantee condition is to ensure that blocks labeled as foreign, or leaked-in, by callee injections $\mu$ are always labeled as public by caller injections $\nu$:

$$\text{foreign}_S \mu \cap \text{owned}_S \nu \subseteq \text{public}_S \nu$$  \hspace{1cm} (1)

From the fact that source modules are reach-closed

$$E_S \subseteq \text{REACH} (\text{roots ge } B)$$  \hspace{1cm} (2)

we then can show that the memory effects of the running callee core at the top of the callstack are confined to callee-allocated (owned) and foreign blocks. This implies that private caller memory regions in $\nu$, which are disjoint from the blocks marked as public by $\nu$, remain unmodified.

A difficulty here is how to relate the root sets of source modules to the visible sets $\text{vis}_S$ used in the simulation relations. We do this by maintaining the following two invariants:

$$\text{roots ge } B \subseteq \text{vis}_S \mu$$  \hspace{1cm} (3)

$$\text{REACH} (\text{vis}_S \mu) \subseteq \text{vis}_S \mu$$  \hspace{1cm} (4)

Invariant (3) says that the root set of the source semantics is a subset of the visible source blocks in $\mu$. This invariant holds initially, for incoming block set $\text{REACH} (\text{roots ge } (\text{blocksOf } \nu))$, and is maintained at external function calls and returns. Condition (4), which we maintain as an invariant of all structured simulations, says that the visible set is closed under reachability. These two conditions, plus (3) and monotonicity of the REACH relation, imply that $E_S$ is a subset of $\text{vis}_S \mu$. This fact, together with condition (1) above, is sufficient to prove the unchanged on relies of Figure 8 at the point at which the running core returns to its calling context.

6. Compositional CompCert

The proved-correct phases of the Compositional CompCert compiler are shown in Figure 13 with optimization phases in gray. The main differences with standard CompCert are: (1) We compile Clight to x86 assembly, whereas standard CompCert compiles a slightly higher-level language (CompCert C) to multiple assembly targets (x86, PowerPC, and ARM); and (2) standard CompCert includes three additional RTL-level optimizations (common subexpression elimination, constant propagation, and function inlining); the adaptation of their proofs is ongoing work. The toplevel theorems we prove are the following.

Theorem 4 (Compiler Correctness). Let CompCert denote the compilation function that composes the phases in Figure 7 in order. If CompCert$(S) = \text{Some } T$, for Clight module $S$ and x86 module $T$, then $[S] \leq [T]$.

Proof. By transitive composition of the simulation proofs for the individual phases in Figure 13 using Theorem 1 □

Corollary 2 (Compositional Compiler Correctness). Let $S_0, S_1, \ldots, S_{N-1}$ be a set of Clight modules such that CompCert$(S_i) = \text{Some } T_i$ for each $i$. Then $S_0, S_1, \ldots, S_{N-1} \Rightarrow_{e} T_0, T_1, \ldots, T_{N-1}$.

Proof. By Corollary 1, Theorem 3, and determinism and validity of CompCert x86 assembly. □

The process of making the proof of a transformation phase compositionally typically proceeded as follows: We refined CompCert’s internal match–state notion $\sim_{r}$ (and the auxiliary relations for activation records, frame stacks, etc.) to relations $\sim_{\mu}$ indexed by structured injections. In particular, because external function call interactions may introduce memory regions related by memory injections in Compositional CompCert, the simulation relations of passes that were previously proved as memory equality or memory extension phases had to be reformulated as injection phases. Particular care was needed to assign correct ownership and visibility information to compiler-introduced memory blocks.

In addition, we had to add to each $\sim_{r}$ relation the clauses: $\text{vis}_S$ is closed under reachability, and the relation $\sim_{r}$ is closed under restriction to the visible set ($\mu \cap \text{vis}_S$). To ensure that global blocks were always mapped by each compiler phase, we treated them as Frgn to all modules. While the addition of these extra invariants proceeded in a mostly uniform manner across all phases, the refinement of $\sim_{r}$ to $\sim_{\mu}$ was phase-by-phase, due to the considerable internal differences between the various CompCert passes.

An issue that required special attention was the treatment of compiler builtins. Here we had to sharpen the distinction between, on the one hand, processor-specific 64-bit helpers and memcpy—these functions are typically inlined, and should never yield at external despite being axiomatized as external calls by mainline CompCert—and true external functions, which are never inlinable, on the other. To sharpen this distinction, we modified the definition of the CminorSel language to ensure that the compiler-introduced calls to 64-bit helpers were unobservable.

All in all, porting the CompCert phases in Figure 13 to structured simulations took approximately 10 person-months. Much of this time was spent at the “boundaries” of the proof, updating the interfaces that connected our linking semantics and proofs to structured simulations. In general, the porting time decreased as the project went on. Adapting the first few phases of the compiler took a few weeks to a month per phase, whereas the later phases went much more quickly (a day or two per phase). This was due in part to greater familiarity with CompCert, but also to the accumulation of a library of general-purpose lemmas that will remain useful as we continue to adapt the last few optimization passes.

As another measure of effort, we give lines-of-code for representative files in the development (Figure 14). The proof of individual phases (“new”) were on the order of 5klocs. By contrast, CompCert 2.1’s (“old”) proofs are about 2× smaller. The increase in proof lines is due mostly to the additional invariants we prove. However, we have not yet applied much proof automation at all, so we believe there is room for improvement. The increase in specification size is due to the use of duplicate language definitions: In order to add effects to the CompCert languages we duplicate the step relation of each semantics (once with, and once without, effects), then prove that the two semantics coincide. This results in specification counts that are larger than necessary.
7. Related Work

Compiler verification is one of the “big problems” of computer science, as evidenced by the large body of research it has spawned in the 45 years or so since McCarthy and Painter (McCarthy and Painter 1967). We cannot hope to give a complete survey here (see Dave 2003). Instead, we focus on the most closely related work.

**Verified Whole-Program Compilers.** Moore (Moore 1989) was one of the first to mechanically verify a programming language implementation (a compiler for a language called Piton). The most well-known work in this vein since Moore is Leroy’s CompCert C compiler in Coq (Leroy 2009), upon which Compositional CompCert is based. Chipala has also built verified compilers in Coq—first, from lambda calculus to idealized assembly language (Chipala 2007), and then, later, for an impure functional language (Chipala 2010). But both Chipala and Leroy’s compilers were limited to whole programs—they did not provide correctness guarantees, as we do in this work, about the behavior of separately compiled multmodule programs.

**Compositional Compilation.** Benton and Hur were two of the first to explicitly do compositional specification of compilers and low-level code fragments, first for a compiler from a simply typed function language to a variant of Landin’s SECD machine (Benton and Hur 2009), then for a functional language with polymorphism (Benton and Hur 2010). Benton and Hur’s work was followed by a string of papers by Dreyer, Hur, and collaborators—that resulted in refinements of the basic techniques (step-indexed logical relations and biorthogonality). The refinements included extensions to step-indexed Kripke logical relations, for dealing with state in the context of more realistic ML-like languages (Hur and Dreyer 2011), and more recently, to relation transition systems (RTSs) (Hur et al. 2013), and the related parametric bisimulations (Hur et al. 2013). RTSs demonstrated that it was possible to do bisimulation-style reasoning in the possible-worlds style of Kripke logical relations and state transition systems; parametric bisimulations refined RTSs by removing some technical restrictions. Both parametric bisimulations and RTSs compose transitively, like our structured simulations but unlike Kripke logical relations.

Although the context of their work is different, some of the techniques used by Benton, Dreyer, Hur, and their collaborators draw interesting parallels in our own work. Our “us vs. them” protocol is at least superficially similar to the “local vs. global knowledge” distinction that’s made in RTSs. One difference is, we distinguish between local and external invariants on the state shared by modules, whereas in RTSs the local vs. global distinction is really about different notions of term equivalence. Also, our “them” invariants—which encapsulate one structured simulation’s view of the memory regions allocated by external functions—are not quite “global” in the same sense as Hur et al.’s global knowledge. Perhaps more fruitfully, one can view interaction semantics—and the structured simulations that are “indexed to” interaction semantics—as an analogue of the type structure used to index standard logical relations, but here applied to imperative languages with impoverished type systems: C, x86, and the other languages of CompCert. As in Kripke logical relations, structured simulations use Kripke-style possible worlds to model, e.g., memory allocation.

An alternative to language-independent interaction semantics is **multi-language semantics** (Ahmed and Blume 2011), which combines several languages of a compiler into a single host language via syntactic boundary casts in the style of Matthews and Finder (Matthews and Finder 2007). This makes it possible to state the correctness of a separate compiler as contextual equivalence in the combined language, as Perconti and Ahmed have recently done for a two-phase compiler from System F with existential and recursive types (Perconti and Ahmed 2014). But where Perconti and Ahmed define contexts syntactically, as one-hole terms in the combined language, we define contexts semantically, as interaction semantics. McKay’s variation of Perconti and Ahmed’s approach replaces explicit boundary conversion with programmatic conversion expressed as terms of the combined language, but considers only a single transformation, closure conversion (McKay 2014). Recently, Wang et al. (2017) built a compositional compiler from a restricted C-like language, Cito, to Bedrock. In contrast to our work, compiler correctness in (2017) is tied to the specifics of the Cito program logic.

**Concurrency.** Liang et al.’s work (Liang et al. 2012) on verifying concurrent program transformations inspired our use of a rely-guarantee discipline, but the complexity of stack frame management, spilling, and block coalescing in CompCert made it difficult to apply their ideas directly in our setting. Ley-Wild and
Nanevski’s SCSL (Ley-Wild and Nanevski 2013), which we mentioned in the introduction, used subjective rely-guarantee invariants on auxiliary state to verify coarse-grained concurrent programs, such as parallel increment. Later work by Nanevski et al. extended the techniques to support verification of fine-grained concurrent programs (Nanevski et al. 2014). These subjective invariants made their proofs robust to the thread structure of the environment. Our “us vs. them” invariants serve a similar purpose—to prevent module-structured simulations from being sensitive to the exact composition of their environment (other modules).

Löchbühler verified a whole-program compiler for multithreaded Java (Löchbühler 2012). Sevcik et al. built CompCertTSO (Sevcik et al. 2013), which adapted CompCert’s correctness proofs to x86-TSO in order to reason about compilation of racy C code. Mansky’s PTRANS framework (Mansky 2014) models optimizations as rewrite operations on parallel control flow graphs, specified using temporal logic formulae. While all these three projects are whole-program, there are some similarities with our work. For example, both CompCertTSO and PTRANS lift program refinements from individual threads to whole programs, as we do for interacting modules, under certain noninterference conditions on shared state. A difference from our work is that PTRANS and CompCertTSO state the noninterference conditions as whole-system invariants. Our horizontal composition results instead rely only on a module-local characterization of noninterference, in the form of reach-closed semantics. That said, it would be interesting to investigate whether the compositional compilation approach we advocate could be applied to compilation with weak memory models.

8. Conclusion
CompCert is one of the great successes of formal methods for software verification. But as the authors of CompCert put it: “[CompCert’s]… formal guarantees of semantic preservation apply only to whole programs that have been compiled as a whole by [the] CompCert C [compiler].” (Leroy 2014) We overcome this restriction.

Acknowledgments
We thank the POPL anonymous reviewers, the members of the Princeton and Yale PL groups, Amal Ahmed, Robert Dockins, Dan Ghica, Robbert Krebbers, Xavier Leroy, and David Pichardie for valuable feedback. We especially thank Tahina Ramananandro for numerous insightful conversations. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References