Artifacts Evaluated acm Functional V1.1
Artifacts Evaluated acm Reusable V1.1

CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files

YUTING WANG[*], Shanghai Jiao Tong University, China
XIANGZHE XU[†], Nanjing University, China
PIERRE WILKE, CentraleSupélec, France
ZHONG SHAO, Yale University, USA

We present CompCertELF, the first extension to CompCert that supports verified compilation from C programs all the way to a standard binary file format, i.e., the ELF object format. Previous work on Stack-Aware CompCert provides a verified compilation chain from C programs to assembly programs with a realistic machine memory model. We build CompCertELF by modifying and extending this compilation chain with a verified assembler which further transforms assembly programs into ELF object files.

CompCert supports large-scale verification via verified separate compilation: C modules can be written and compiled separately, and then linked together to get a target program that refines the semantics of the program linked from the source modules. However, verified separate compilation in CompCert only works for compilation to assembly programs, not to object files. For the latter, the main difficulty is to bridge the two different views of linking: one for CompCert's programs that allows arbitrary shuffling of global definitions by linking and the other for object files that treats blocks of encoded definitions as indivisible units.

We propose a lightweight approach that solves the above problem without any modification to CompCert's framework for verified separate compilation: by introducing a notion of syntactical equivalence between programs and proving the commutativity between syntactical equivalence and the two different kinds of linking, we are able to transit from the more abstract linking operation in CompCert to the more concrete one for ELF object files. By applying this approach to CompCertELF, we obtain the first compiler that supports verified separate compilation of C programs into ELF object files.

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

Additional Key Words and Phrases: Verified Separate Compilation, Assembler Verification, Generation of Object Files

ACM Reference Format:
Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 197 (November 2020), 28 pages. https://doi.org/10.1145/3428265

[*] Part of this work was done while the author was an associate research scientist at Yale University.
[†] Part of this work was done while the author was a visiting undergraduate student in research at Yale University.

Authors' addresses: Yuting Wang, Shanghai Jiao Tong University, China, yuting.wang@sjtu.edu.cn; Xiangzhe Xu, Nanjing University, China, xxz@smail.nju.edu.cn; Pierre Wilke, CentraleSupélec, France, pierre.wilke@centralesupelec.fr; Zhong Shao, Yale University, USA, zhong.shao@yale.edu.

2475-1421/2020/11-ART197
https://doi.org/10.1145/3428265

1 INTRODUCTION

A grand challenge in the field of formal verification is to create a stack of verified system software that serves as a foundation on which realistic software systems can be developed and verified. In particular, the supported verification should be *end-to-end*, meaning that the behavior of their executable code running on actual hardware matches their behavior at the level of source code [Appel et al. 2017]. The key constituents of this stack include verified operating systems that provide abstraction over hardware on which verified binary programs are executed, and verified compilers that generate such binary programs.

The binary programs communicated between the compilers and OS must adhere to a common format that is specified in the complete application binary interface (ABI) of the OS, e.g. the Executable and Linkable Format (ELF) on Unix-like operating systems. The common practice of compilers is to separately compile modules into *relocatable* object files in this format and link them to form a complete executable file. Therefore, for verified compilers to interface with the verified OS, the *complete* compilation chain from the source programs to the object file format must be verified. Moreover, it must support *separate compilation* of individually written modules into relocatable object files for practical software development.

CompCert [Leroy 2009a] is the state-of-the-art verified compiler for the C programming language whose development is fully mechanized in the Coq proof assistant. Despite the merits and previous successes of CompCert [Appel 2011; Gu et al. 2016, 2018], it is still not adequate to serve as the verified compiler in the verified stack we mentioned above. One reason is that its verified compilation chain does not output object files. Instead, it outputs assembly programs formalized in Coq whose semantics does not conform to that of real assembly programs, e.g., its stack model is not continuous and finite. Further compilation of these assembly programs into binary code is carried out by first pretty printing them into assembly (`.s`) files and then assembling and linking these files into binary code using external tools (e.g. the GNU assembler and linker). Since all these processes are unverified, it is unclear whether the behavior of the executable code output by CompCert does correspond to that of the C source programs. Another shortcoming of CompCert is that its official support of separate compilation, known as *verified separate compilation* [Kang et al. 2016], also only covers compilation to assembly programs formalized in Coq, hence is too weak for modular end-to-end verification.

Previous work on extending CompCert partially solved the above problems. The most recent and relevant effort in this direction is Stack-Aware CompCert [Wang et al. 2019]. It provides an approach to compiling to realistic assembly programs that operate over a continuous and finite stack. These assembly programs are then compiled into a language called MC with a flat memory space. However, MC *1)* relies on unverified external processes to be compiled to actual object files, and *2)* is still a representation of *closed* programs that is quite far away from the standard relocatable and linkable object files. Therefore, Stack-Aware CompCert cannot support verified compilation to and linking of object files.

In this paper, we introduce CompCertELF, the first extension of CompCert that supports verified separate compilation from C source programs all the way to an object file format, i.e., the relocatable ELF format. We discuss the key features of CompCertELF in the following paragraphs.

Compilation to Relocatable Object Files. CompCertELF is built on top of Stack-Aware CompCert. In essence, it modifies and extends Stack-Aware CompCert's compilation chain with a *verified assembler* that transforms the realistic assembly programs output by Stack-Aware CompCert to sequences of bytes that represent relocatable ELF files. This assembler consists of several passes that successively merge code and data into atomic blocks (known as sections), generate information for relocation and linking (such as symbol tables and relocation tables), encode the assembly instructions and the global data into their binary forms, and finally embed them into the ELF format with meta-data (such as section headers and ELF headers). Like Stack-Aware CompCert, these exercises are carried out for CompCert v3.0.1 and on the 32-bit x86 architecture. [1]

Verified Separate Compilation to Relocatable Object Files. Verified separate compilation is based on the commutativity between linking and compilation of programs. Roughly speaking, if a finite set of C modules (files) $\{P_i | i \in 0 \ldots n\}$ are successfully compiled to assembly modules $\{A_i | i \in 0 \ldots n\}$ by *the same compiler* (e.g. CompCert) and $\{P_i | i \in 0 \ldots n\}$ are linked to form a complete program P at the source level, then $\{A_i | i \in 0 \ldots n\}$ can also be linked at the assembly level to form a program A which is the result of compiling P using the mentioned compiler.[2] As a result of this commutativity, the compiler's correctness theorem guarantees that A behaves the same as P.

We would like to extend the separate compilation theorem to cover the complete compilation chain of CompCertELF. For this, we need to prove compilation commutes with two different linkers: one used in CompCert and the other for relocatable object files. On one hand, in CompCert, the linker views programs as partial mappings from identifiers to global definitions. Therefore, it is insensitive to and may arbitrarily rearrange the order of definitions. On the other hand, to generate relocatable object files, the function and variable definitions are merged into atomic sections following some particular order. The ELF linker simply concatenates sections of the same type together like they are "black boxes." Here, the order in which definitions are merged and sections are concatenated significantly affects the result of linking. The consequence is that commutativity between linking and compilation breaks at the transitional phase from CompCert to the assembler where the latter links programs as relocatable objects.

A naive solution is to give a more concrete view of programs to CompCert's linker and change its implementation to match the behavior of the ELF linker. Then, the commutativity between compilation and the two linkers becomes obvious. However, this solution has two problems. First, linking at all levels of CompCert would be constrained by how linking of low-level object files works. Second, it involves heavy modification to the separate compilation framework and commutativity proofs for CompCert's passes.

We give a *complete and lightweight* solution to the above problem without changing any existing proof for separate compilation. The key ingredients are *1)* a new linker for CompCert programs that takes into account the order in which definitions are merged into and linked as sections, and *2)* a notion of *syntactical equivalence* between program modules that ignores the order of definitions. By inserting syntactical equivalence as a vacuous compiler pass and proving that it commutes with the original and new linkers, we are able to transit from the original linker to the new one, and then to the linker in the assembler. After proving the commutativity property for this transition, we proceed to verify separate compilation for the rest of the assembler as usual. Using this technique, all the newly introduced proofs fit into CompCert's original framework, and we are able to prove the correctness of separate compilation for the complete compilation chain of CompCertELF.

1.1 Contributions

We summarize our contributions as follows:

- We have developed CompCertELF which, to the best of our knowledge, is the first verified extension of CompCert that compiles from C source all the way down to a standard object

[1] For compatibility with Stack-Aware CompCert, this work is based on CompCert v3.0.1. We do not foresee any problem of porting this work to the newer versions of CompCert.

[2] For simplicity, we use a compiler in describing commutativity. As we shall see later, the compiler can be generalized to a relation between the source and target programs.

file format (ELF in our case). The key component of CompCertELF is a verified assembler that compiles CompCert assembly into binary code in the ELF format.

- We have generalized verified separate compilation in CompCert to work for the full compilation chain of CompCertELF. The technical novelty is a lightweight approach to bridging the abstract view of linking in CompCert and the more concrete view of static linking for relocatable object files without any modification to CompCert's separate compilation framework. To the best of our knowledge, this is the first realization of verified separate compilation that works for real-world object file formats.

1.2 Generality of Our Approach

Verified separate compilation piggybacks on and entirely reuses CompCert's correctness theorem, making it a very lightweight approach to verification of modular compilation. It is also transparently compatible with all the features of CompCert. Because its concept is independent of any particular compiler, it can be implemented in other compilers as well to derive the same benefits. It is important to note that, to realize verified separate compilation in any compiler targeting object files, we need to address the same problem manifested in CompCertELF, i.e., how to transit from the abstract view of linking at the source-level to the concrete one for binary object files. Our technique for proving commutativity between compilation and these two different linking operations provides a general solution to this problem.

The main limitation of verified separate compilation is its assumption on using the *same* compiler for compiling all modules (with some variants as described by Kang et al. [2016]). There exist other more general approaches to *verified compositional compilation* that break this limitation and support compilation and linking of heterogeneous modules [Stewart et al. 2015; Jiang et al. 2019; Song et al. 2020]. However, all these approaches only deal with compilation of source programs into CompCert's assembly programs and adopt the abstract view of linking in the vanilla CompCert. CompCertELF provides an approach to further verifying the separate compilation of assembly programs into binary object files. Therefore, it *complements* the above approaches. In the future, we would like to investigate how to combine CompCertELF with these approaches to realize end-to-end verified modular compilation to object files.

Finally, although we target ELF on 32-bit x86 architecture in this paper, we do not foresee any difficulty in adopting our approach to other architectures and other object file formats.

1.3 Outline

We organize the rest of the paper as follows. In Sec. 2 we introduce the basic concepts of CompCert and the relocatable ELF format that are necessary for our subsequent discussion. We then give an overview of the challenges in verified separate compilation to ELF and our approach to addressing them. In Sec. 3, we introduce the CompCertELF compiler and discuss in detail its key transformations that lead to relocatable binary code. In Sec. 4, we elaborate on our technique for implementing and verifying separate compilation for CompCertELF. In Sec. 5, we discuss the current implementation of CompCertELF and its limitations, and present an evaluation of our effort. We discuss related work in Sec. 6 and conclude in Sec. 7.

2 BACKGROUND AND APPROACH

2.1 An Introduction to CompCert

The verified compilation chain of CompCert takes a C module (parsed from a `.c` file) as input, transforms it through a sequence of compiler passes to an architecture-independent language called Mach, from which it finally generates an assembly program on a pre-configured architecture. This is

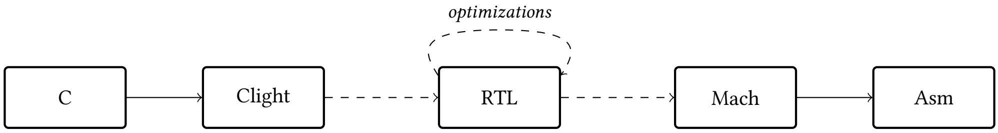

Fig. 1. The Verified Compilation Chain of CompCert

depicted in Fig. 1. CompCert supports a substantial subset of C and non-trivial optimization passes such as dead-code elimination, constant propagation, function inlining and tail-call optimizations (all implemented at the level of the Register Transfer Language or RTL), making it the state-of-the-art compiler for non-trivial verification projects that use C.

CompCert provides a uniform and general representation of programs. In every language of CompCert, a program is treated as a list of pairs that associates global definitions with unique identifiers, defined as the following record type in Coq:[3]

```
Record program (F V: Type) : Type := mkprogram { prog_defs: list (ident * (globdef F V)); }.
```

Here, `ident` is the type of identifiers; `globdef` is the type of global definitions parameterized by the types of functions and variables in a certain language (e.g. C or assembly). A global definition may be either a function or a variable definition, each of which may be either internal or external.

CompCert uses a single memory model for all of its languages [Leroy and Blazy 2008; Leroy et al. 2012]. In this model, a memory state consists of a set of disjoint memory blocks, each of which has a unique block id and a finite size. The set of memory blocks may grow infinitely and hence is *unbounded*.

CompCert's languages are given small-step operational semantics represented as transition systems between finite machine states. A machine state consists of a memory state and a language-specific state (e.g., register state for assembly). In any language $\mathcal{L}$, the semantics of a program P—denoted by $[\![P]\!]_{\mathcal{L}}$ or just $[\![P]\!]$ if $\mathcal{L}$ is clear from the context—is uniformly described by using a set of predicates that capture the initial and final states and the small-step transitions. Note that these semantics are defined only for closed programs: there must exist a main function as the entry point and any external references to functions or variables must have pre-defined semantics.

Every pass of CompCert is proved to preserve the semantics of closed programs. Semantics preservation is stated as either a *closed backward* or *closed forward simulation*, where the latter is much easier to prove and can be flipped into the former under certain conditions.[4] We write $[\![P_t]\!] \preccurlyeq [\![P_s]\!]$ ($[\![P_t]\!] \succcurlyeq [\![P_s]\!]$) to denote the target program P_t (the source program P_s) backward (forward) simulates P_s (P_t). Concretely, $[\![P_t]\!] \succcurlyeq [\![P_s]\!]$ holds if there exists an invariant I between machine states of P_s and P_t such that I holds of the initial states of P_s and P_t and every step of the source execution can be simulated by one or more steps of the target execution while maintaining I. The definition of $[\![P_t]\!] \preccurlyeq [\![P_s]\!]$ is roughly the inverse of the statement above. The part of the invariant I for memory states is described as a *memory injection* (denoted by j), a partial function from block identifiers to memory locations. By writing $\lfloor\rfloor$ to represent both the option type and its `Some` constructor and $\oslash$ to represent the `None` constructor, a memory injection j removes a block b from memory if $j(b) = \oslash$, or it injects b into the block b' at the offset δ if $j(b) = \lfloor(b', \delta)\rfloor$. Memory injection is highly flexible and capable of characterizing all kinds of memory invariants used in the verification of CompCert's passes, including the complicated transformation of stack memories [Leroy et al. 2012; Leroy 2009b].

[3] To simplify our discussion, we have omitted the fields containing the identifier of the main function and public symbols.
[4] Specifically, when the target semantics is *determinate* and the source semantics is *receptive*.

The verification of a compiler pass C is broken down into two steps. Firstly, a binary relation $\mathcal{R}$ between source and target programs of C—called its *matching program relation*—is defined to capture the essence of the transformation. It is shown that C is subsumed by $\mathcal{R}$, i.e., $\forall P_s\ P_t, C(P_s) = \lfloor P_t \rfloor \Longrightarrow \mathcal{R}(P_s, P_t)$. This one-way implication indicates that $\mathcal{R}$ may be a generalization of C in the sense that it can relate more than the inputs and outputs of C for some passes (e.g., elimination of unused static definitions). Secondly, it is shown that a forward (or backward) simulation is derivable from $\mathcal{R}$, i.e., $\forall P_s\ P_t, \mathcal{R}(P_s, P_t) \Longrightarrow [\![P_t]\!] \succcurlyeq [\![P_s]\!]$. Note that for passes where $\mathcal{R}$ is a generalization of C CompCert proves more than the semantics preservation between the inputs and outputs of C.

Let $C_{\texttt{compcert}} : \texttt{C.program} \rightarrow \lfloor \texttt{Asm.program} \rfloor$ denote the CompCert compiler, i.e., the transitive composition of its passes $C_1, \ldots, C_n$. Let $\mathcal{R}_i$ be the matching program relation for C_i, and $\mathcal{R}_{\texttt{compcert}} = \mathcal{R}_1 \circ \ldots \circ \mathcal{R}_n$ be the transitive composition of $\mathcal{R}_1, \ldots, \mathcal{R}_n$. By transitively composing simulation proofs for C_i (with necessary flipping of forward simulations into backward ones), the following theorem can be derived:

Theorem 1 (Correctness of CompCert for Compiling Closed Programs).

$$\forall P_s\ P_t, C_{\texttt{compcert}}(P_s) = \lfloor P_t \rfloor \Longrightarrow [\![P_t]\!] \preccurlyeq [\![P_s]\!].$$

2.2 Verified Separate Compilation

In reality, a software system is almost always composed of a collection of modules. CompCert provides an abstract notion of syntactic linking based on its general representation of programs. By using this notion it supports verified compilation and linking of individually written modules, following the approach to *verified separate compilation* proposed by Kang et al. [2016]. We describe these concepts below.

The Syntactic Linking Operation. A single linking operation (i.e., CompCert's linker) is defined for all languages of CompCert. It merges global definitions associated with the same identifiers in two input programs to form a new program, as follows:

```
Definition link_prog {F V: Type} ($P_1$ $P_2$: program F V) :=
  if link_prog_check $P_1$ $P_2$ then
    let t := PTree.combine link_def (get_def_tree $P_1$) (get_def_tree $P_2$) in
    Some {| prog_defs := PTree.elements t |}
  else None.
```

Here, linking of global definitions is defined by `link_def` and works as expected: any two definitions d_1 and d_2 may be linked into a definition d by `link_def` only if at most one of them is an internal definition and they are either both function definitions or both variable definitions. The function `link_prog_check` checks that definitions in P_1 and P_2 indeed satisfy this condition. If so, by calling `get_def_tree`, the linker converts P_1 and P_2 into `PTree` structures which can be thought as *partial mappings* from identifiers to their associated definitions. It then performs the critical step of linking: calling `PTree.combine` to combine the two partial mappings into one and flattening the result back into a list by calling `PTree.elements`. Concretely, $t = \texttt{PTree.combine}\ f\ t_1\ t_2$ is defined as follows:

$$t(id) = \begin{cases} \lfloor t_1(id) \rfloor & t_1(id) \neq \emptyset \wedge t_2(id) = \emptyset \\ \lfloor t_2(id) \rfloor & t_1(id) = \emptyset \wedge t_2(id) \neq \emptyset \\ f(t_1(id), t_2(id)) & t_1(id) \neq \emptyset \wedge t_2(id) \neq \emptyset \\ \emptyset & \text{Otherwise} \end{cases}$$

By definition, the linker preserves definitions that appear only in P_1 or P_2 (but not both) and links definitions associated with the same identifiers that appear both in P_1 and P_2 by using `link_def`. We shall use $\oplus_{\mathrm{p}}$ to denote `link_prog` and write $P_1 \oplus_{\mathrm{p}} P_2$ for `link_prog` P_1 P_2. By repeatedly applying $\oplus_{\mathrm{p}}$ we can link a set of modules into a complete program.

A Framework for Commutativity. Separate compilation is based on a general notion of commutativity between linking and binary relations, which is defined as follows.

Definition 2. *Let $\mathcal{R} : A \longrightarrow B \longrightarrow \mathbb{P}$ be a binary relation between the types A and B and $\oplus_A : A \longrightarrow A \longrightarrow \lfloor A \rfloor$ and $\oplus_B : B \longrightarrow B \longrightarrow \lfloor B \rfloor$ be two linking operators. Then $\oplus_A$* and *$\oplus_B$* commute with *$\mathcal{R}$ if the following proposition is true:*

$$\forall P_1\ P_2\ P\ P_1'\ P_2', P_1 \oplus_A P_2 = \lfloor P \rfloor \Longrightarrow \mathcal{R}(P_1, P_1') \Longrightarrow \mathcal{R}(P_2, P_2') \Longrightarrow \exists P', P_1' \oplus_B P_2' = \lfloor P' \rfloor \wedge \mathcal{R}(P, P')$$

We write it as $\texttt{Commute}(\oplus_A, \oplus_B, \mathcal{R})$.

From Definition 2 we can easily prove that commutativity is transitively and horizontally composible, stated as follows:

Lemma 3 (Transitive composition of commutativity).

$$\forall \oplus_A \oplus_B \oplus_C \mathcal{R}_1\ \mathcal{R}_2, \texttt{Commute}(\oplus_A, \oplus_B, \mathcal{R}_1) \Longrightarrow \texttt{Commute}(\oplus_B, \oplus_C, \mathcal{R}_2) \Longrightarrow \texttt{Commute}(\oplus_A, \oplus_C, \mathcal{R}_1 \circ \mathcal{R}_2)$$

where $(\mathcal{R}_1 \circ \mathcal{R}_2)(x, y)$ holds if any only if $\exists z, \mathcal{R}_1(x, z) \wedge \mathcal{R}_2(z, y)$.

Lemma 4 (Horizontal composition of commutativity). *Given* $\texttt{Commute}(\oplus_A, \oplus_B, \mathcal{R})$, *the following is provable*

$$\begin{aligned} &\forall (P_i\ P_i', 1 \leqslant i \leqslant m)\ P, P_1 \oplus_A \ldots \oplus_A P_m = \lfloor P \rfloor \Longrightarrow (\forall 1 \leqslant i \leqslant m, \mathcal{R}(P_i, P_i')) \\ &\quad \Longrightarrow \exists P', P_1' \oplus_B \ldots \oplus_B P_m' = \lfloor P' \rfloor \wedge \mathcal{R}(P, P'). \end{aligned}$$

Proving Separate Compilation. The separate compilation theorem we mentioned in Sec. 1 is formally stated as follows:

Theorem 5 (Correctness of Separate Compilation for CompCert).

$$\begin{aligned} &\forall (P_i\ P_i', 1 \leqslant i \leq m)\ P, P_1 \oplus_{\mathsf{p}} \ldots \oplus_{\mathsf{p}} P_m = \lfloor P \rfloor \Longrightarrow (\forall 1 \leqslant i \leqslant m, C_{\texttt{compcert}}(P_i) = \lfloor P_i' \rfloor) \\ &\quad \Longrightarrow \exists P', P_1' \oplus_{\mathsf{p}} \ldots \oplus_{\mathsf{p}} P_m' = \lfloor P' \rfloor \wedge [\![P']\!] \leqslant [\![P]\!]. \end{aligned}$$

Its proof proceeds as follows. By applying Theorem 1 and Lemma 4, we reduce this theorem to the commutativity property $\texttt{Commute}(\oplus_{\mathsf{p}}, \oplus_{\mathsf{p}}, \mathcal{R}_{\texttt{compcert}})$. $\texttt{Commute}(\oplus_{\mathsf{p}}, \oplus_{\mathsf{p}}, \mathcal{R}_{\texttt{compcert}})$ is then proved by showing $\oplus_{\mathsf{p}}$ commutes with the matching program relation of every pass of CompCert, i.e., $\texttt{Commute}(\oplus_{\mathsf{p}}, \oplus_{\mathsf{p}}, \mathcal{R}_i)$ holds for $1 \leqslant i \leqslant n$, and transitively composing the results by Lemma 3.

Verifying Separate Compilation under Expansions of Contexts. Several important optimization passes of CompCert perform value analysis in certain contexts. To support separate compilation, their proofs must take into account the possible expansion of contexts after linking. Concretely, this means that a matching program relation must be parameterized by a context, making the proof of commutativity more complex than described before. Nevertheless, the framework we presented above is sufficient for illustrating the key challenges of verified separate compilation to object files which we will discuss later. Readers interested in the details of the generalized separate compilation framework can consult the paper written by Kang et al. [2016].

2.3 Relocatable ELF Object Files

The target programs of CompCertELF are relocatable ELF object files. As shown in Fig. 2, a relocatable ELF object file is a sequence of bytes starting with a fixed-size header, followed by a list of sections and ending with a list of fixed-size section headers that store attributes (meta-data) for these sections.

```c
extern int get(void);
extern void incr(void);
int limit = 10;
int rcd[10] = {0};

void record(int c) { rcd[c] = c; }

int main() {
  int i = get();
  while (i < limit) { incr(); i = get(); }
  return 0;
}
```

(a) main.c

```c
extern void record (int);
int c = 0;

int get() {return c;}

void incr() {
  record(c);
  c++;
}
```

(b) counter.c

Fig. 3. The Running Example: C Modules

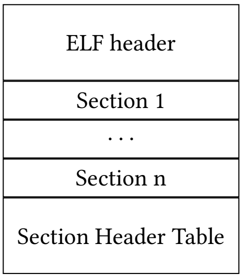

Fig. 2. Relocatable ELF Format

Sections contain information about program modules and how they should be linked to form an executable file. Concretely, sections are *atomic* lists of bytes that should not be analyzed or divided—the linking operation should simply concatenate them together untouched, while their interpretation are determined by their headers. Examples include code and data sections that store encoded instructions and data. Two types of sections are provided for relocation and linking: the symbol table sections and the relocation table sections. A symbol table contains a list of *symbol entries* for symbols that are defined or declared in the object file. Each entry contains information about a symbol such as its type, location and size. A relocation table contains a list of *relocation entries*. Each relocation entry points to a location in code or data sections that refers to a symbol; it also tells how the binary value of the symbol reference should be calculated and filled into this location when the concrete addresses of symbols are fixed by linking. There exists a relocation table for every section that needs relocation.

2.4 Challenges in Verified Separate Compilation into ELF

Our goal is to extend CompCert so that it compiles to ELF, and supports verified separate compilation into and static linking of relocatable ELF files. To illustrate the challenges in achieving these goals, we shall use a running example consisting of two C modules (depicted in Fig. 3) in the rest of this paper. In Fig. 3, the module `counter.c` contains a straightforward implementation of a counter with two methods for increasing and getting the counter's value. Every time `incr` is invoked, it also calls an external function to record the value of the counter in an array. The main function in `main.c` repeatedly invokes `incr` until a pre-defined limit is reached. It also provides the function and array for recording counter values used in `counter.c`. Note that each module contains multiple function and variable definitions, together with external declarations that mutually refer to definitions in the other module. This creates an interesting scenario where linking is non-trivial and the intricacy of verified separate compilation is exposed. We now describe the major challenges below which mirror the main features of CompCertELF described in the introduction.

Compilation to Relocatable ELF Files. An immediate challenge is to prove a simulation between the semantics of CompCert programs which operates with an unbounded set of memory blocks and that of machine code which works with finite memory space on actual hardware. This is partially solved by Stack-Aware CompCert which compiles to programs with a finite stack [Wang et al.

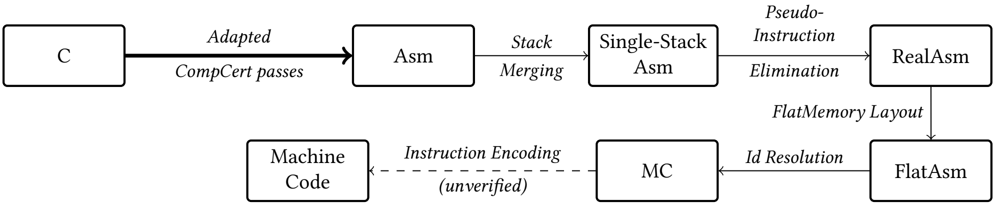

Fig. 4. Stack-Aware CompCert

```
  ...
record:
  subl   $12, %esp
  leal   16(%esp), %eax
  movl   0(%eax), %eax
  movl   %eax, rcd(,%eax,4)
  addl   $12, %esp
  ret
main:
  subl   $12, %esp
  leal   16(%esp), %eax
  call   get
  ...
```

(a) main.s

```
  ...
get:
  subl   $12, %esp
  leal   16(%esp), %eax
  movl   c, %eax
  addl   $12, %esp
  ret
incr:
  subl   $12, %esp
  leal   16(%esp), %eax
  ...
```

(b) counter.s

Fig. 5. The Running Example: Assembly Modules

2019]; its structure is shown in Fig. 4. Contradicting to the view of the stack as a continuous and finite piece of memory, the "stack" in CompCert is a linked list of memory blocks. Moreover, the size of this list is not bound. As a result, CompCert's assembly relies on pseudo-instructions for allocation and deallocation of stack frames (called `Pallocframe` and `Pfreeframe`) that do not exist in the actual instruction sets. Stack-Aware CompCert solves this problem by augmenting the memory model of CompCert with an abstract data type that represents the finite and continuous stack. Using this memory model, it extends CompCert with passes that merge the list of stack blocks into a finite and continuous stack and replace pseudo-instructions with instructions that adjust the stack pointer using pointer arithmetic. This results in assembly programs containing only real assembly instructions and with a more realistic semantics; they are called RealAsm programs. For example, the two C modules in Fig. 3 are compiled by Stack-Aware CompCert into the RealAsm modules depicted in Fig. 5. Here, the stack pointer `esp` is adjusted directly for allocation and deallocation of stack frames instead of using `Pallocframe` and `Pfreeframe`.

The instructions shown in Fig. 5 actually correspond to formalized assembly instructions in Coq. We use the AT&T x86 syntax to represent them for readability. In the rest of the paper, we shall use Fig. 5 as the running example for discussing the compilation of assembly programs into ELF files. Note that we have only shown some snippets of the generated code and omitted the data declarations completely. Nevertheless, this suffices for our discussion.

Unfortunately, Stack-Aware CompCert does not target *relocatable* (i.e., open and linkable) programs with a finite memory model. It compiles RealAsm programs into a language called MC. An MC program is basically a *closed* program where all the symbols must be resolved to concrete memory addresses. As such, neither `main.s` nor `counter.s` can be compiled to an MC program because they contain external functions. Furthermore, the instructions and data are not encoded into binary forms in MC. Further compilation to executables is handled by external tools such as the RockSalt x86-model [Morrisett et al. 2012] and is not formally verified.

$$\begin{tikzcd}
P_1 & \oplus_{\mathsf{p}} & \cdots & \oplus_{\mathsf{p}} & P_m \arrow[d, "\mathcal{R}"] & = & \lfloor P \rfloor \arrow[d, "\mathcal{R}"] \\
B_1 \arrow[from=1-1, d, "\mathcal{R}"] & \oplus_{\mathsf{b}} & \cdots & \oplus_{\mathsf{b}} & B_m & = & \lfloor B \rfloor
\end{tikzcd}$$

Fig. 6. The Desired Commutativity Property for Separate Compilation to ELF Files

Definitions	`limit`	`c`	`rcd`	`record`	`get`	`incr`	`main`
Identifiers	3	5	9	10	12	18	22

Fig. 7. The Identifiers Assigned to the Definitions in the Running Example

To completely realize verified compilation to relocatable ELF, we first need to give a formal representation and a semantics with finite memory for relocatable programs. Then, on top of stack merging provided by Stack-Aware CompCert, we need to implement and verify the complete compilation chain from realistic assembly programs to relocatable ELF, including the generation of the symbol table, sorting and merging instructions and data into sections, generating necessary relocation information and meta-data, and encoding all the sections and meta-data into the relocatable ELF format. As we shall see later in Sec. 3, the verification of these processes involves complex proofs for establishing memory injections between unbounded and finite memories. The correctness proof of instruction encoding is also highly complex because the different fields of the encoded instructions may have various dependency relations between each other and the encoding process itself is dependent on its context. None of the above issues has been addressed in previous work on extending CompCert.

Verified Separate Compilation to Relocatable ELF Files. According to the discussion in Sec. 2.2, to show that the complete compilation chain from C to relocatable ELF files supports verified separate compilation, it suffices to show that it commutes with linking. This commutativity property is described in Fig. 6 where $\mathcal{R}$ is the matching program relation of compilation, $P_1, \ldots, P_m$ are source C programs and $B_1, \ldots, B_m$ are the target binary ELF files and $\oplus_{\mathsf{b}}$ is the linker for ELF files.

Unfortunately, the above property does not hold in general. To give a counterexample, consider the compilation and linking of our running example. At the source level, the linking operator $\oplus_{\mathsf{p}}$ of CompCert converts `main.c` and `counter.c` into partial mappings from identifiers (of type `ident` which is internally the type of positive numbers in CompCert) to global definitions, merges them into a single mapping, and flattens the result back to a list of definitions associated with identifiers. Assume that the global definitions in these modules are given the identifiers shown in Fig. 7. Moreover, assume that as a result of flattening the global definitions are sorted in the increasing order of their identifiers after linking. Then, commutativity between compilation and $\oplus_{\mathsf{p}}$ and $\oplus_{\mathsf{b}}$ breaks, as depicted in Fig. 8. In this example, we assume that the compilation of `main.c` (or `counter.c`) transforms and concatenates its internal global definitions into code and data sections following to the order they occur in their modules. The ELF linker $\oplus_{\mathsf{b}}$ simply concatenates code or data sections to form a single section. In the end, commutativity fails to hold because the order of definitions in the linked source modules does not match with that in the linked sections. Specifically, note that at the source level `get` and `incr` are inserted between `record` and `main` and `c` is inserted between `limit` and `rcd` by $\oplus_{\mathsf{p}}$. Similar operations are impossible at the target level because code and data sections are not divisible.

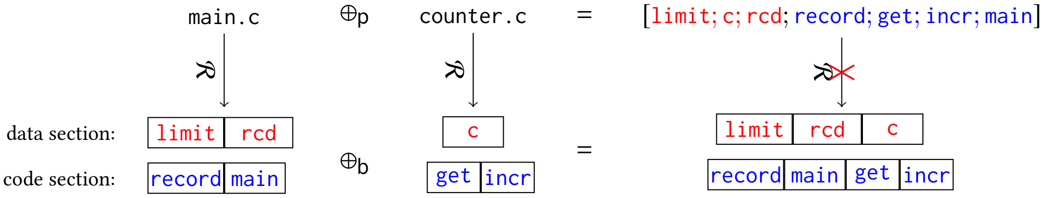

Fig. 8. A Broken Commutativity Property for Separate Compilation to ELF Files

In essence, the failure of commutativity in Fig. 8 is the combined effect of two facts: *1)* the same compilation chain is used for compiling all modules which changes the order of definitions in a uniform way (in case of the above example the order is preserved), and *2)* there is a mismatch between the view of linking at the source level and the view of linking at the binary level. On one hand, at the source level, because the linker treats programs as partial mappings, it is insensitive to the order of definitions and is free to arbitrarily rearrange this order. On the other hand, at the binary level, even before linking the orders of definitions *within* sections have already been fixed by compilation. Since the ELF linker must treat sections as black boxes and can only concatenate them together, the orders of definitions within these sections are preserved by linking. In the final result, the encoded definitions have to follow these particular orders which may very likely deviate from the orders at the source level.

On the surface, it looks like that the above issue can be easily fixed by changing the implementation of $\oplus_p$ to match the behavior of $\oplus_b$. However, this does not only make the high-level linking operation unnecessarily constrained by low-level behaviors, but also requires heavy modification to the separate compilation framework and to the proofs of CompCert's passes, especially the proofs of optimization passes which are further complicated by expansions of contexts.

One may also think this issue can be fixed by modifying the definition of commutativity (Definition 2) so that it permutes the output of the source linker to match with that of the target linker. However, this extra permutation conflicts with the basic requirement of using a uniform compilation chain in verified separate compilation and hence breaks the transitive and horizontal compositionality of commutativity (Lemma 3 and Lemma 4).

In summary, there is no trivial solution to the problem of bridging the abstract and concrete views of linking. The naive solutions either break or require heavy modification to the framework of verified separate compilation. Since this problem is universal for any compiler that intends to support both verified separate compilation and generation of object files, it is important to provide a satisfying solution to it.

2.5 An Overview of Our Approach

To realize verified compilation from C programs to the relocatable ELF format, we extend Stack-Aware CompCert with additional passes to form CompCertELF. The key component of CompCertELF is a verified assembler that compiles RealAsm to an intermediate representation called *relocatable programs* and finally to binary files in the ELF format. We prove that the complete compilation chain of CompCertELF preserves the semantics of closed programs. These efforts will be discussed in Sec. 3.

We then extend the separate compilation theorem of Stack-Aware CompCert to cover the newly introduced passes. For this, we develop a new linker $\oplus_r$ for relocatable programs that behaves similarly to $\oplus_b$. Because the behavior of this new linker contradicts that of CompCert's linker $\oplus_p$

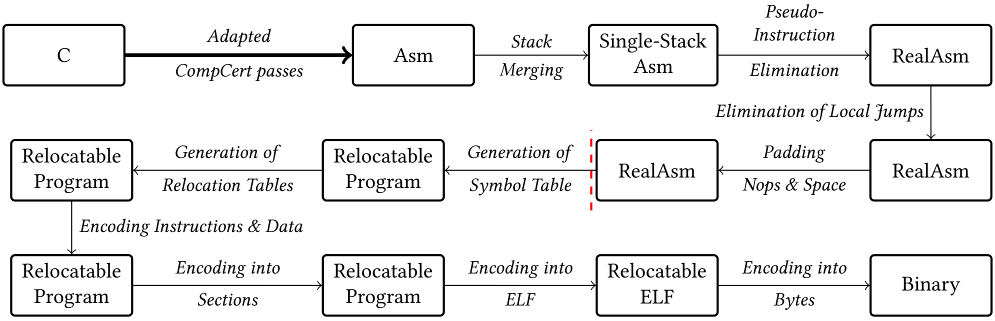

Fig. 9. CompCertELF

for reasons described in Sec. 2.4, commutativity between linking and compilation fails to hold for the transitional pass from RealAsm to relocatable programs.

We develop a solution to this problem without introducing any change to the existing proofs. The main idea is to introduce a notion of *syntactical equivalence* between programs and insert it as a vacuous "compiler pass" into the compilation chain. Then, by introducing an alternative linker $\oplus_{\mathsf{o}}$ for CompCert programs whose behavior matches $\oplus_{\mathsf{r}}$ and by proving commutativity between syntactical equivalence and $\oplus_{\mathsf{p}}$ and $\oplus_{\mathsf{o}}$, we are able to recover the commutativity property for the transitional pass and switch the linker from $\oplus_{\mathsf{p}}$ to $\oplus_{\mathsf{o}}$ and finally to $\oplus_{\mathsf{r}}$. After that, separate compilation is verified following the standard pattern for the remaining passes. These developments will be discussed in Sec. 4.

3 VERIFIED COMPILATION OF C PROGRAMS INTO RELOCATABLE ELF FILES

3.1 An Overview of CompCertELF

The complete compilation chain of CompCertELF is depicted in Fig. 9. The compiler passes of Stack-Aware CompCert are reused up to the pass that eliminates pseudo-instructions and generates RealAsm programs. The remaining passes are newly introduced. The first new pass eliminates all the labels used by intra-procedure jumps by replacing them with relative offsets from source instructions to target labels. The relative offsets are calculated by using the function `instr_size` which was previously an oracle in Stack-Aware CompCert and now instantiated by our instruction encoder which we will discuss in Sec. 3.5. The second new pass is a necessary preparation for the verification of the following pass. To prove the following pass correct, we need to show it is possible to concatenate a list of global functions and variables into continuous sections. This in turn requires establishing a memory injection from memory blocks for individual functions and global variables to their starting offsets in the corresponding sections. The definition of memory injections requires the injected offsets to be aligned at the word size. This pass pads the functions with `Nop` instructions and initialization data with empty space to meet this requirement.

The passes after the red dashed line form an assembler for RealAsm. They successively perform merging of global functions and variables into sections, generation of the symbol table and the relocation tables, encoding of instructions and data, generation of meta-data, and final encoding into the ELF files. They work with three new representations of programs. The *relocatable programs* (denoted by R) are programs with relocation information. The *relocatable ELF* (denoted by E) is the formalization of a subset of the relocatable ELF format in Coq that is sufficient for our compilation. The final output is simply a list of bytes encoded from relocatable ELF programs (denoted by B).

We prove that every newly introduced pass preserves semantics. Combining these results with Stack-Aware CompCert, we get the following correctness theorem of CompCertELF for compiling closed programs, where $C_{\texttt{compcertelf}}$ is the complete compilation chain of CompCertELF:

Theorem 6 (Correctness of CompCertELF for Compiling Closed Programs).

$$\forall P\ B, C_{\texttt{compcertelf}}(P) = \lfloor B \rfloor \implies [\![B]\!] \preccurlyeq [\![P]\!].$$

For the rest of this section, we first introduce the formalization of relocatable programs and ELF programs, then discuss the implementation and verification of the passes composing our assembler. We omit the discussion of other new passes because their implementation and verification are straightforward.

3.2 Relocatable Programs and ELF

Relocatable Programs. A relocatable program contains a list of sections, a symbol table which is a list of symbol entries and a mapping from section indexes to relocation tables. It is defined as the following Coq type.

```
Record reloc_program : Type := { prog_sectable: list section;
                                 prog_symbtable: symbtable;
                                 prog_reloctables: PTree.t reloctable }.
```

Sections are defined using the following inductive type:

```
Inductive section : Type := | sec_null                      | sec_text (code: list instruction)
                            | sec_data (init: list init_data) | sec_bytes (bs: list byte).
```

A section is either null, a code section containing a list of formalized assembly instructions, a data section containing a list of formalized initialization data, or a list of bytes. Before the instruction encoding pass, code and data sections contain formalized instructions and data, respectively. After that, they are converted into bytes.

A symbol table of type `symbtable` is a list of symbol entries of the following type (where `Z` is the type of integers in Coq):

```
Record symbentry : Type := { symbentry_id: option ident;
                             symbentry_type: symbtype;
                             symbentry_bind: bindtype;
                             symbentry_value, symbentry_size : Z;
                             symbentry_secindex: secindex }.
```

A symbol entry is translated from a global definition. It contains the original identifier of the translated definition, the type of this symbol (e.g., function or data), the type of its binding (e.g., global or local), the value and size of the symbol and the index of the section. Section indices are defined as the following inductive type:

```
Inductive secindex : Type := | secindex_normal (idx:N) | secindex_undef | secindex_comm.
```

Depending on whether the symbol entry points to an internal definition, an external one or a "common" symbol, its index field is either `secindex_normal idx` where `idx` is the index to the section the symbol resides in, `secindex_undef` or `secindex_comm`; its value is either the offset into the section the symbol resides in, 0 or its alignment, respectively.

A relocation table of type `reloctable` is a list of relocation entries of the following type (where `N` is the type of natural numbers in Coq):

```
Record relocentry : Type := { reloc_offset, reloc_addend: Z;
                              reloc_type: reloctype;
                              reloc_symb: N; }.
```

A relocation entry contains the starting offset of the location to be relocated, an addendum that is a constant adjustment to the relocated value, the type of the relocation, and the index to the target symbol in the symbol table. In particular, the relocation type `reloc_type` decides what binary value should be filled into the relocatable location when the concrete address of the target symbol `reloc_symb` is fixed. We will elaborate on this point in Sec. 3.4.

Relocatable ELF. We encode a relocatable ELF program as a record of the following type:

```
Record elf_file := { elf_head : elf_header;
                     elf_sections : list section;
                     elf_section_headers : list section_header }
```

where the sections only contain bytes. An ELF program generated by CompCertELF has a specific layout. Roughly speaking, it contains the data and code sections, a symbol table, a string table holding the names of symbols, the relocation tables for the data and code sections, and a string table holding the names of sections. Note that it is sufficient to work with this subset of ELF because it is the only type of programs CompCertELF will generate.

3.3 Generation of the Symbol Table

Implementation. This pass transforms CompCert's assembly programs into relocatable programs. We shall use C_{symb} to denote this pass in the rest of this paper. The essence of C_{symb} is to merge *internal definitions* into sections that should stay as unbreakable units from this point on, and to record necessary information about symbols in the symbol table for linking and relocation. For the latter, we use the following function to iteratively generate symbol entries.

```
Definition get_symbentry (ds cs:N) (dof cof: Z) (id:ident) (def: (globdef Asm.fundef unit)) : symbentry :=
  match def with
  | (Gfun (Internal f)) ⇒
    {| symbentry_id := Some id;                    symbentry_bind := get_bind_ty id;
       symbentry_type := symb_func;                symbentry_value := cof;
       symbentry_secindex := secindex_normal cs;   symbentry_size := code_size (fn_code f) |}
  | ...    end.
```

The first two arguments of `get_symbentry` are the indices to the data and code sections. The following two arguments are the offsets to the next internal definitions to be added into either section; they are updated accordingly after a new symbol entry is generated. The next two arguments are the definition whose symbol is to be generated and its identifier. As an example, the above code snippet shows the case of translating an internal function. The resulting symbol entry has the index to the code section and has the offset to the next definition in the code section as its value. We omit other cases as they have a similar structure.

As a concrete example, consider the transformation of `main.s` in Fig. 5. It generates a data section containing the initialization data for the global variable `limit` and `rcd`, and a code section containing a list of instructions by concatenating the instructions in `record` and `main`. It also generates a symbol table with 6 entries, one for each of the global definitions including the external ones. For example, suppose the size of instructions in `record` is n. Then, the entry for `main` has an index to the code section and the value n which is also the starting offset of `main` in the code section.

Verification. In the original CompCert, to describe the semantics of a program, a distinct memory block is allocated for each of its definitions. In the semantics of relocatable programs, distinct memory blocks are still allocated for the external definitions. However, since the internal definitions are merged into sections, a single continuous memory block is allocated for each section. An internal definition now resides at a certain offset of a section indicated by its symbol entry.

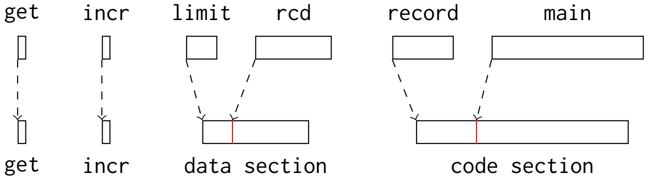

Fig. 10. The Memory Injection for Symbol Table Generation for the Running Example

Semantics preservation for this pass is formulated as a forward simulation. To prove it, we show that there exists an invariant I between the machine states of the source program P and the target program R. The key component of I is a memory injection j between the separate memory blocks for global definitions in P and the continuous memory blocks for sections in R. For example, the memory injection for the compilation of `main.s` in our running example is depicted in Fig. 10.

Given the memory injection j, the key is to show that *1)* it holds between the initial states of the source and target programs; and *2)* it is preserved throughout the entire execution of the source and target programs in lockstep. The proof for the first part is actually quite non-trivial. To understand it, note that before the transformation a memory block is allocated for each definition, assigned appropriate permission and finally initialized with proper values. After the transformation, a single memory block for all internal function (variable) definitions is allocated *at once*. Then, as we traverse the definitions, the memory regions for them are successively assigned permissions and initialized. We need to prove that *1)* even after the merging of memory blocks the target memory operations for individual definitions do not interfere with each other, and *2)* the reordered memory operations have the same effect as before and in the end generate an initial target memory state relatable to the initial source memory state by j. It turns out that both parts require very complicated reasoning about low-level properties of memory states and memory injections. However, once j is shown to be an invariant, the remaining proof is finished in a straightforward manner.

3.4 Generation of the Relocation Tables

Types of Relocation. We first describe the types of relocation supported by CompCertELF. Given a relocation entry `r`, let S be the concrete address of the target symbol `r.reloc_symb`, P be the concrete address of the relocatable location (i.e., the concrete address of the section `r` is associated with plus `r.reloc_offset`), and A be the value of `r.reloc_addend`. Our implementation supports two types of relocation. The first one is relocation of absolute references. In this case, the field `r.reloc_type` is set to the constructor `reloc_abs`. The relocatable location will be filled in with $S + A$ which is the absolute address of the target symbol. The second one is relocation of PC-relative references. In this case, the field `r.reloc_type` is set to `reloc_rel`. The relocatable location will be filled in with $S + A - P$ which is the offset between the target symbol and the relocatable location plus the addendum.

Implementation. In this pass, one relocation table is generated for each of the code and data sections. In the following discussion we use the generation for the code section as an example; the other cases are similar.

For each instruction that refers to an identifier of a global definition, a relocation entry is generated. For example, 5 relocation entries will be generated for `main.s`: 3 for call instructions applied to `get` and `incr` and 2 for accessing the global variables `limit` and `rcd`.

Prefix	**Opcode**	**ModRM**	**SIB**	**Imms**

Fig. 11. The Format of the x86 Instructions

For any control instruction (e.g. jump or function call) whose argument is a global identifier, we generate a relocation entry for a PC-relative reference by calling the following function (below we use the syntax of the error monad provided by CompCert):

```
Definition compute_instr_rel_relocentry (sofs:Z) (i:instruction) (symb:N) :=
  do iofs ← instr_reloc_offset i;
  do addend ← instr_addendum i;
  OK {| reloc_offset := sofs + iofs;   reloc_type := reloc_rel;
       reloc_symb := symb;   reloc_addend := addend |}.
```

Here, `sofs` is the starting offset of the instruction `i` in the code section; `symb` is the index to the target symbol. The value `iofs` is the starting offset of the relocatable location in `i`, computed by calling `instr_reloc_offset`. The addendum is computed by calling `instr_addendum`.

As an example, consider the `call get` instruction in `main.s` in Fig. 5. It will be encoded as `E8 00 00 00 00` where `E8` is the opcode for the short-call instruction that jumps to relative offsets in the code segment and the following 4 bytes need to be relocated to the offset between the concrete address of the function `get` and the concrete address at the end of `call get`. Here, `instr_reloc_offset (call get)` will return 1 and `instr_addendum (call get)` will return -4. Because the relocation is for a PC-relative reference, the relocated value is equal to `db − 4 − (dc + sofs + 1)` where `db` and `dc` are the concrete addresses of `get` and the code section, respectively. Therefore, it is the expected offset `db − (dc + sofs + instr_size(call get))`.

The remaining instructions that need relocation are those for memory access or computation of memory addresses that take global symbols as arguments. For these instructions, relocation entries for absolute references are generated. We omit a discussion of the relocation process for them since it is similar to the one above.

Verification. The semantics of relocatable programs after this pass has changed in the sense that symbols are no longer used directly. Instead, for each section a mapping from the offsets of the relocatable locations in this section to relocation entries is generated from its relocation table. When an instruction with a symbol argument is encountered in execution, the generated mappings are used to query the relocation entry associated to the offset to this argument. The actual symbol is then found in the `reloc_symb` field of this entry.

To prove semantics preservation for this pass, the key is to show that the extra steps to access symbols described above form a "round trip", i.e., the obtained symbols are the same as the original ones. For this, the main difficulty is to show that the generated mappings are one-to-one, .i.e, the offsets in relocation entries are unique. This property is proved by inspecting how the offsets are computed using the functions `instr_size`, `instr_reloc_offset` and `instr_addendum`.

3.5 Instruction and Data Encoding

This pass encodes the x86-32 instructions in the code section and the initialization data in the data section into a list of bytes. We shall discuss the instruction encoding in detail and omit a discussion of data encoding as it is straightforward.

X86 Instruction Format. For the upcoming discussion, we briefly introduce the format of the x86 instructions as depicted in Fig. 11. The x86 instructions could be divided into several parts. The first part is a prefix. In the subset of x86 we are dealing with, the prefix either does not exist or could only be 66, indicating that the width of operands is 16-bit instead of 32-bit by default. Following the prefix byte are the opcode bytes. Then there is a byte named ModRM. The ModRM byte indicates what addressing modes are used for the operands of this instruction. These addressing modes, which follow the ModRM byte, include SIB (Scale, Index and Base) and an immediate displacement.

Implementation. We have manually written an encoder by following the Intel manual of x86 architecture. A snippet of this encoder is shown as follows:

```
Definition encode_instr (ofs:Z) (rtbl:reloctable) (i: instruction) : res (list byte) :=
  match i with
  | Pmovl_mr a rs ⇒ do abytes ← encode_addrmode ofs rtbl a rs;
                     OK (HB["89"] :: abytes)
  | ...   end.
```

The implementation is straightforward: we compute different fields of the instruction separately and concatenate them together. Note that the encoding process is context sensitive: the offset of the instruction in the code section and the relocation table for the code section are passed as arguments. This is because the displacement in the addressing mode may either be a constant or a global identifier. In the latter case, as required by the ELF linker, we need to store the addendum into the relocatable location by inquiring the relocation entry at the expected offset.

Consider the instruction `movl %eax, rcd(,%eax,4)` in `main.s`, where `rcd` is a reference to the global array `rcd` which will be relocated later and `rcd(,%eax,4)` is an addressing mode. Internally, it is represented as the CompCert assembly instruction

```
Pmovl_mr (Addrmode None (eax,4) (rcd,0)) eax.
```

The order of arguments to this instruction is inverted because CompCert internally uses the Intel x86 syntax instead of the AT&T syntax. The arguments to `Addrmode` are the base register, the index register and the scale, and the displacement. Note that the above displacement `(rcd,0)` represents the address of `rcd` plus a 0 offset.

The encoder generates the opcode `89`, and calls `encode_addrmode` to inspects the encoded addressing mode `(Addrmode None (eax,4) (rcd,0))` to generate the ModRM byte `04` and the SIB byte `85`. The call to `encode_addrmode` should also return the displacement which should be the addendum for relocating `rcd` as required by the ELF linker. For this, it finds the relocation entry in `rtbl` that has the same offset as that of the symbol `rcd`. It then reads the addendum, say `00 00 00 00`, from this entry and uses it as the displacement value. The final result is `89 04 85 00 00 00 00`.

Verification. To verify the encoder, it is necessary to define the semantics for the encoded programs. We do this by providing a decoder that parses encoded bytes back to a list of assembly instructions; the semantics of encoded bytes is simply the semantics of the resulting instructions.

Our decoder is a recursive descent parser. Given a list of bytes, it reads the first few bytes of the list representing the opcode to determine the type of the encoded instruction. It then dispatches the remaining bytes to a function that specifically decodes this type of instructions. A snippet of the decoder function `decode_instr` is shown as follows:

```
Definition decode_instr (ofs:Z) (rtbl: reloctable) (bs: list byte) : res (instruction * list byte) :=
  match bs with
  | nil ⇒ Error(msg "Nothing to decode")
  | h:: t ⇒ if Byte.eq_dec h HB["89"] then decode_movl_mr ofs rtbl t
            else if Byte.eq_dec h HB["90"] then
              ...
  end.
```

Note that decode_instr consumes the bytes for the first encoded instruction and returns the remaining bytes for further decoding. Like encoding, decoding is context sensitive: decode_instr takes the starting offset of bs in the code section and the relocation table for the code section as arguments, which are used in the parsing of addressing modes as we will discuss shortly.

We use the decoding of `movl %eax, rcd(,%eax,4)` to illustrate how decode_instr works. The decoder first recognizes that this is a move instruction and dispatches the remaining operand bytes `04 85 00 00 00 00` to the following function that decodes move instructions with one addressing mode operand and one register operand:

```
Definition decode_movl_mr (ofs:Z) (rtbl:reloctable) (bs: list byte) : res(instruction * list byte) :=
  do '( r, adrmode, bs') ← decode_addrmode ofs rtbl bs;
  OK(( Pmovl_mr adrmode r), bs').
```

Here, the implementation of decode_addrmode is the inverse of encode_addrmode. It is non-trivial because it needs to not only resolve the complicated dependencies between the different fields in the encoded instruction, but also extract context-sensitive information from the relocation table to get back the original addressing mode. Specifically, the displacement in an addressing mode may either be a constant number or an identifier. This information needs to be recovered by first looking up the relocation table to determine if a relocation entry at the offset of this displacement exists. If so, we know the original displacement refers to the symbol in the found entry. Otherwise, the original displacement is a constant. Following this idea, decode_addrmode returns the addressing mode `(Addrmode None (eax,4) (rcd,0))` and the register eax, as expected.

Semantics preservation of this pass is formulated as a forward simulation. The key is to prove that our encoder and decoder are *consistent* with each other. That is, for each instruction i encoded into a sequence of bytes s, our decoder decodes s back to an assembly instruction i' that is equivalent to i where the equivalence is encoded using the predicate instr_eq. We may not get back exactly the same instruction because our encoder is not injective: a set of multiple CompCert assembly instructions may be encoded into the same sequence of bytes. The decoder can only decode it back to *some* instruction in this set. However, we have proved the decoded instruction can be related to the original input by instr_eq and instructions related by instr_eq are semantically equivalent. This suffices for proving semantics preservation. This consistency property is stated as follows:

```
Lemma encode_decode_instr_refl: ∀ ofs rtbl i s l,
  encode_instr ofs rtbl i = OK s →
  ∃ i', decode_instr ofs rtbl (s ++ l) = OK (i', l) ∧ instr_eq i i'.
```

Note that the encoder and decoder must use the same offset and relocation table and the decoder exactly consumes the encoded bytes and returns the remaining input unchanged.

3.6 Encoding into the ELF Files

The last three passes encode the remaining sections, generate necessary headers, and finally output the sequence of bytes in the relocatable ELF format. The proofs for them are similar to instruction encoding in the sense that we define a decoder, describe the semantics of the target programs by first decoding them and then using the semantics of the source programs, and derive semantics preservation by proving the consistency between the encoder and decoder. Because the remaining ELF sections have a fixed format (unlike x86 instructions), the consistency properties between their encoders and decoders are easier to prove.

4 VERIFIED SEPARATE COMPILATION TO RELOCATABLE ELF FILES

With the whole program correctness of CompCertELF in place, we extend the separate compilation theorem to cover all the passes of CompCertELF. That is, given a collection of C modules, if we can

$$\begin{tikzcd}
P_1 \arrow[d, "C_{\mathtt{compcertelf}}"] & \oplus_{\mathsf{p}} & \cdots & \oplus_{\mathsf{p}} & P_m \arrow[d, "C_{\mathtt{compcertelf}}"] & = & \lfloor P \rfloor \mathbin{\text{->}} [\![P]\!] \arrow[d, phantom, "\rotatebox{90}{$\preccurlyeq$}"] \\
B_1 & \oplus_{\mathsf{b}} & \cdots & \oplus_{\mathsf{b}} & B_m & = & \lfloor B \rfloor \mathbin{\text{->}} [\![B]\!]
\end{tikzcd}$$

Fig. 12. Verified Separate Compilation of CompCertELF

link them at the source level and separately compile them into relocatable ELF object files using CompCertELF, then we can also link the ELF object files to generate a single file that preserves the semantics of the linked source modules. This is depicted in Fig. 12.

To prove this property, the key is to show linking commutes with the newly introduced passes in CompCertELF. Commutativity is easily proved for passes that deal with RealAsm programs which use CompCert's linker. However, for the transitional pass from RealAsm to relocatable programs, i.e., C_{symb}, commutativity breaks for reasons we described in Sec. 2.4. In this section, we first present an approach to recovering this commutativity property and discuss its application to the transitional pass. We then talk about verified separate compilation for the remaining passes, and finally conclude with the separate compilation theorem for CompCertELF. As we shall see, our approach fits completely in the existing separate compilation framework and does not require any change to its proofs.

4.1 Verifying Separate Compilation for the Generation of Symbol Tables

We first define the linker for relocatable programs—which we denote as $\oplus_{\mathsf{r}}$—as follows:

```
Definition link_reloc_prog (p1 p2: reloc_program) : option reloc_program :=
    match link_sectable (prog_sectable p1) (prog_sectable p2),
          link_symbtable (prog_symbtable p1) (prog_symbtable p2)
          link_reloctables (prog_reloctables p1) (prog_reloctables p2) with
    | Some sectbl, Some symbtbl, Some reloctbls ⇒ Some {| prog_sectable := sectbl;
                                                          prog_symbtable := symbtbl;
                                                          prog_reloctables := reloctbls; |}
    | _, _, _ ⇒ None end
```

This definition has three parts: the linking of sections, symbol tables and relocation tables. The first part simply concatenates sections of the same type in input modules into a single section by calling `link_sectable`; it reflects the fact that sections should be treated as "black boxes" by linking. The second part calls `link_symbtable` to merge and link the symbol entries like the linking of global definitions in $\oplus_{\mathsf{p}}$ (i.e., by treating symbol tables as partial mappings from identifiers to symbols and merging them, albeit with necessary adjustments to the offsets of symbols). The third part calls `link_reloctables` to concatenate the relocation tables of the same kind of sections together. As we can see, $\oplus_{\mathsf{r}}$ works in a way similar to the ELF linker $\oplus_{\mathsf{b}}$. Therefore, the commutativity between linking and compilation from RealAsm to relocatable programs (i.e., C_{symb}) is not directly provable as we have elaborated in details in Sec. 2.4.

We introduce an approach to solving the above problem. At a high level it works as follows. To match the behavior of $\oplus_{\mathsf{r}}$, we introduce an alternative linker for CompCert programs (denoted by $\oplus_{\mathsf{o}}$) that treats internal definitions as if they have already been merged into sections. We then introduce a notion of syntactical equivalence for both CompCert programs and relocatable programs. By inserting syntactical equivalence relations before and after C_{symb} as vacuous "compiler passes", we are able to recover the commutativity between linking and compilation by first transiting from

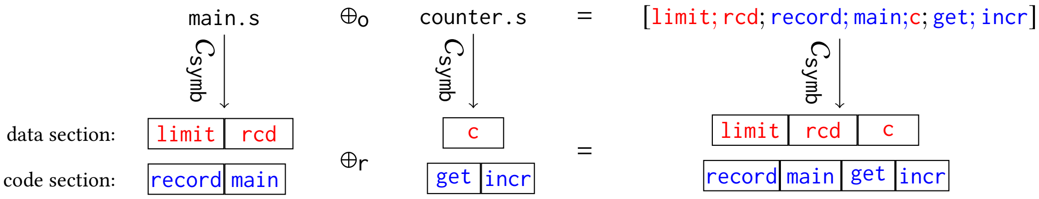

Fig. 13. An Example of Commutativity for Symbol Table Generation using $\oplus_o$

$\oplus_p$ to $\oplus_o$ and then from $\oplus_o$ to $\oplus_r$. By showing that syntactically equivalent programs are also semantically equivalent and that syntactical equivalence subsumes identity transformations, we are able to prove semantics preservation for the extended compilation chain which subsumes C_{symb} as a special case. In the end, separate compilation using C_{symb} is verified by combining these proofs with the recovered commutativity property. We elaborate on these ideas below.

An Alternative Linker for CompCert. The first ingredient of our solution is a new linking operator $\oplus_o$ for CompCert programs which we call *ordered linking*. It is defined as follows:

```
Definition link_prog_ordered P1 P2 :=
  if link_prog_check P1 P2 then
    let t := PTree.combine link_def (get_def_tree P1) (get_def_tree P2) in
    let ids1 := collect_internal_def_ids P1 in
    let ids2 := collect_internal_def_ids P2 in
    match PTree_extract_elements (ids1 ++ ids2) t with
    | None ⇒ None
    | Some (defs, t') ⇒ Some {| prog_defs := PTree.elements t' ++ defs |}  end
  else None.
```

Compared to the definition of $\oplus_p$ in Sec. 2.2, it performs the following extra operations. It calls `collect_internal_def_ids` to collect the identifiers of the internal definitions in the input modules into lists. The order these identifiers occur in the lists are the same as that of their associating definitions in the input modules. After merging the input definitions into a partial mapping `t` like in $\oplus_p$, `link_prog_ordered` divides the internal definitions in `t` from the external ones by calling `PTree_extract_elements` on (`ids1 ++ ids2`) and `t` (note that upon successful linking (`ids1 ++ ids2`) contains exactly the identifiers of internal definitions in `t`). Specifically,

```
PTree_extract_elements (ids1 ++ ids2) t = Some (defs, t')
```

holds if and only if `defs` is a list of pairs of identifiers and internal definitions in `t` where the identifiers are in the same order as (`ids1 ++ ids2`), and `t'` is the result of removing `defs` from `t` (which contains exactly the external definitions in `t`). In the end, these operations effectively emulate how $\oplus_r$ works: the internal definitions in input modules are compacted as atomic sections and concatenated together by $\oplus_o$ with their orders within the input modules preserved.

Consider the separate compilation of `main.s` and `counter.s` in our running example by C_{symb}, the failed commutativity described in Sec. 2.4 is recovered if we use $\oplus_o$ instead of $\oplus_p$, as shown in Fig. 13.

Syntactical Equivalence between Programs. The second ingredient for solving the commutativity problem is a notion of syntactical equivalence between programs. Specifically, two CompCert programs P_1 and P_2 are syntactically equivalent if one can be obtained from the other by permuting its definitions; we write it as $P_1 =_p P_2$. Two relocatable programs R_1 and R_2 are syntactically equivalent if they have exactly the same sections and their relocation tables and symbol tables can be obtained from each other via permutation; we write it as $R_1 =_r R_2$. Note that $=_r$ enforces rigid equality between unanalyzable components (sections) while allowing for flexibility in comparing analyzable components (symbol and relocation tables).

Obviously, both $=_p$ and $=_r$ are reflexive. It means that they are valid matching program relations for any identity transformation $C(P) = \lfloor P \rfloor$ and may be inserted into the compilation chain as vacuous "compiler passes." Such compiler passes preserve semantics, as stated below:

Lemma 7 (Semantics Preservation for Syntactical Equivalence).

$$\forall P_1\ P_2, P_1 =_p P_2 \Longrightarrow [\![P_2]\!] \geqslant [\![P_1]\!] \wedge [\![P_2]\!] \leqslant [\![P_1]\!]$$
$$\forall R_1\ R_2, R_1 =_r R_2 \Longrightarrow [\![R_2]\!] \geqslant [\![R_1]\!] \wedge [\![R_2]\!] \leqslant [\![R_1]\!]$$

Proof. We define a memory injection j that captures the permutation of memory blocks incurred by syntactical equivalence. We conclude the proof by showing that j is an invariant. □

Recovering Commutativity and Verifying Separate Compilation. We assume that there are extra identity transformations before and after C_{symb}. Then, by using $=_p$ and $=_r$ as their matching program relations, we define the matching program relation of C_{symb} as follows:

Definition 8. *$\mathcal{R}_{symb}$ is a binary relation between CompCert assembly and relocatable programs. $\mathcal{R}_{symb}(P, R)$ holds if and only if there exists P' and R' such that $P =_p P'$, $R' =_r R$ and $C_{\mathsf{symb}}(P') = \lfloor R' \rfloor$.*

The extra syntactical equivalence relations provide the flexibility to transit from $\oplus_{\mathsf{p}}$ to the order linking $\oplus_{\mathsf{o}}$ and finally to $\oplus_{\mathsf{r}}$ through the use of the following commutativity properties:

Lemma 9. *$\oplus_{\mathsf{p}}$ and $\oplus_{\mathsf{o}}$ commutes with syntactical equivalence. That is,* `Commute`$(\oplus_{\mathsf{p}}, \oplus_{\mathsf{o}}, =_p)$ *holds.*

Proof. Given $P_1 \oplus_{\mathsf{p}} P_2 = \lfloor P \rfloor$, $P_1 =_p P_1'$ and $P_2 =_p P_2'$, we need to show there exists P' s.t. $P_1' \oplus_{\mathsf{o}} P_2' = \lfloor P' \rfloor$ and $P =_p P'$. This is proved by first showing that merging of definitions using `PTree.combine` commutes with permutation and `PTree_extract_elements ids t` always returns some result if `ids` are in the domain of `t` and the result is a permutation of the definitions in `t`. □

Lemma 10. *$\oplus_{\mathsf{o}}$ and $\oplus_{\mathsf{r}}$ commutes with the combination of symbol table generation and syntactical equivalence. That is, letting $\mathcal{R}'_{symb}(P, R)$ hold iff there exists R' such that $C_{\mathsf{symb}}(P) = \lfloor R' \rfloor$ and $R' =_r R$,* `Commute`$(\oplus_{\mathsf{o}}, \oplus_{\mathsf{r}}, \mathcal{R}'_{symb})$ *holds .*

Proof. Given $P_1 \oplus_{\mathsf{o}} P_2 = \lfloor P \rfloor$, $\mathcal{R}'_{symb}(P_1, R_1)$ and $\mathcal{R}'_{symb}(P_2, R_2)$, we need to show there exists R s.t. $R_1 \oplus_{\mathsf{r}} R_2 = \lfloor R \rfloor$ and $\mathcal{R}'_{symb}(P, R)$. The proof has two parts. Firstly, we need to show that the sections generated by compiling P is the concatenation of the sections in R_1 and R_2. This is true because sections are generated by concatenating internal definitions and the internal definitions in P is exactly the concatenation of the internal definitions in P_1 and P_2 by the definition of $\oplus_{\mathsf{o}}$. Secondly, we need to show that the symbol table generated by compiling P is a permutation of the result of merging symbol table in R_1 and R_2. This is true by similar reasoning in the proof of Lemma 9, but in a reversed order because we are transiting from ordered linking back to "unordered" linking. □

In essence, Lemma 9 holds because $=_p$ can relate the input and output definitions of $\oplus_{\mathsf{p}}$ and $\oplus_{\mathsf{r}}$ through *different* permutation relations. These differences exactly capture the change to the orders of definitions made by $\oplus_{\mathsf{o}}$. Lemma 10 holds because the linking of sections in $\oplus_{\mathsf{r}}$ matches with the linking of definitions in $\oplus_{\mathsf{o}}$ as we explained before. Note that this matching relation is not affected by $=_r$ because by definition it relates exactly the same sections. However, because linking of symbol tables in $\oplus_{\mathsf{r}}$ mirrors that of definitions in $\oplus_{\mathsf{p}}$, we need $=_r$ to play a role like $=_p$ to absorb

$$\begin{tikzcd}
P_1 \arrow[d, phantom, "=_p"] & \oplus_{\mathsf{p}} & P_2 \arrow[d, phantom, "=_p"] & = & \lfloor P \rfloor \arrow[d, phantom, "=_p"] \\
P_1' \arrow[d, "C_{\mathsf{symb}}"'] & \oplus_{\mathsf{o}} & P_2' \arrow[d, "C_{\mathsf{symb}}"'] & = & \lfloor P \rfloor \arrow[d, "C_{\mathsf{symb}}"'] \\
R_1 \arrow[d, phantom, "=_r"] & & R_2 \arrow[d, phantom, "=_r"] & & \lfloor R \rfloor \arrow[d, phantom, "=_r"] \\
R_1' & \oplus_{\mathsf{r}} & R_2' & = & \lfloor R' \rfloor
\end{tikzcd}$$

Fig. 14. The Structure of the Proof of Commutativity for Symbol Table Generation

the differences between the orders of definitions in the input and output of $\oplus_{\mathsf{o}}$ and the orders of symbol entries in the input and output of $\oplus_{\mathsf{r}}$.

Now, we derive the commutativity between linking and symbol table generation:

Lemma 11.

$$\texttt{Commute}(\oplus_{\mathsf{p}}, \oplus_{\mathsf{o}}, \mathcal{R}_{symb})\mathit{holds}.$$

Proof. Proved by transitively composing Lemma 9 and Lemma 10. □

The structure of the above proof is depicted in Fig. 14. Note that we have used a uniform relation $\mathcal{R}_{symb}$ for representing the compilation of all modules and shown it commutes with linking. Therefore, the new proofs fit completely into CompCert's separate compilation framework. Moreover, by keeping the source linker $\oplus_{\mathsf{p}}$ as it is and using $=_p$ to transit from $\oplus_{\mathsf{p}}$ to $\oplus_{\mathsf{o}}$, we have avoided any modification to the existing proofs before C_{symb}.

To complete the verification, we need to prove that $\mathcal{R}_{symb}$ preserves semantics. This is easily achieved by transitively composing the semantics preservation proofs of $=_p$ and $=_r$ (Lemma 7) and that of C_{symb} discussed in Sec. 3.3.

Note that the notations and techniques we have introduced to bridge the abstract and the concrete views of linking do not rely on particular languages or compiler features. Therefore, our approach is generally applicable to any other compiler that aims to generate object files while supporting verified separate compilation

4.2 Verifying Separate Compilation for the Remaining Passes

The commutativity properties for the remaining passes can be proved by following the standard pattern because they all use linkers that behave similarly to $\oplus_{\mathsf{r}}$. From these properties it is easy to derive the correctness theorems of separate compilation. We briefly discuss them below.

Commutativity for the Generation of Relocation Tables. Let C_{reloc} be this pass. The commutativity between linking and generation of relocation tables is then stated as $\texttt{Commute}(\oplus_{\mathsf{r}}, \oplus_{\mathsf{r}}, C_{reloc})$. To prove it, we need to show that, for any two code (data) sections S_1 and S_2, the relocation table generated from the concatenation of S_1 and S_2 is the same as the concatenation (with adjustment of relocation offsets) of the relocation tables individually generated from S_1 and S_2. This is proved by analyzing the definition of relocation table generation in a straightforward manner.

Commutativity for Instruction and Data Encoding. Let C_{encode} be this pass. We need to show that $\texttt{Commute}(\oplus_{\mathsf{r}}, \oplus_{\mathsf{r}}, C_{encode})$ holds. We observe that the only difference between the linking operations before and after this pass is that the former links instructions and data formalized in CompCert

Table 1. Evaluation of Specifications and Proofs

Components	Spec	Proof	Sum
Programs & Semantics	1207	1709	2916
Linking Infrastructure	806	838	1644
Elimination of Local Jumps	451	734	1185
Padding of Nops & Space	113	283	396
Generation of Symbol Table	2980	7166	10146
Generation of Relocation Tables	260	1122	1382
Encoding Instructions & Data	966	1356	2322
Encoding into Sections	1037	696	1733
Encoding into ELF	289	379	668
Encoding into Bytes	303	151	454
Total (lines of code in Coq)	8412	14434	22846

and the latter links the bytes they are encoded into. By inspecting the structure of instruction and data encoding, we derive $\mathtt{Commute}(\oplus_{\mathsf{r}}, \oplus_{\mathsf{r}}, C_{encode})$ straightforwardly.

Commutativity for the Encoding Passes. The remaining passes perform various encoding of sections and meta-data. The linker for these encoded programs first decode them back to the original programs; they then perform linking using $\oplus_{\mathsf{r}}$ and re-encode the programs. By the consistency between the encoding and decoding processes which we have proved in Sec. 3.6, we know that individually encoded modules can always be decoded back to the original modules. The commutativity between linking and encoding easily follows from these observations.

4.3 The Separate Compilation Theorem of CompCertELF

We compose the above proofs together to get the separate compilation theorem for CompCertELF:

Theorem 12 (Correctness of Separate Compilation for CompCertELF).

$$\forall (P_i\ B_i, 1 \leqslant i \le m)\ P, P_1 \oplus_{\mathsf{p}} \ldots \oplus_{\mathsf{p}} P_m = \lfloor P \rfloor \Longrightarrow (\forall 1 \leqslant i \leqslant n, C_{\mathtt{compcertelf}}(P_i) = \lfloor B_i \rfloor)$$
$$\Longrightarrow \exists B, B_1 \oplus_{\mathsf{b}} \ldots \oplus_{\mathsf{b}} B_n = \lfloor B \rfloor \wedge [\![B]\!] \leqslant [\![P]\!].$$

Its proof is similar to Theorem 5. The only differences are that we make use of the commutativity properties proved above and the fact that syntactically equivalent programs are also semantically equivalent.

5 EVALUATION AND LIMITATIONS

Before talking about our evaluation, we first discuss some limitations of the current implementation of CompCertELF. First, it only targets the 32-bit x86 backend. Second, its output format is a limited subset of ELF and the ELF linker $\oplus_{\mathsf{b}}$ only works for this subset. For evaluation that involves linking of programs with library files, we use GNU `ld` instead of $\oplus_{\mathsf{b}}$. Third, it does not support linking of static symbols in separate compilation, which requires a mechanism for renaming static global variables that the original CompCert also lacks. None of these limitations is fundamental. We plan to further expand the capability of CompCertELF to remove them in the future.

We divide our evaluation into the following parts:

Specifications and Proofs. On top of Stack-Aware CompCert, CompCertELF introduces almost 23k lines of code (LOC) in Coq including specifications and proofs (calculated using `coqwc`). It took a little more than 1 person year to finish. The statistics of the development effort is shown in Table 1.

Table 2. Evaluation of Performance

Name	CompCertELF		CompCert(v3.0.1)		Name	CompCertELF		CompCert(v3.0.1)	
	time	size	time	size		time	size	time	size
aes	0.72	21894	0.71	18980	lists	0.62	3381	0.58	1760
almabench	0.64	14087	0.63	11636	mandelbrot	0.96	3351	0.85	1792
binarytrees	0.74	4054	0.72	2228	nbody	6.30	5404	6.22	3520
bisect	3.31	6661	3.30	4260	nsievebits	0.56	3079	0.54	1480
chomp	1.16	8238	1.16	5632	nsieve	0.89	2850	0.82	1308
fannkuch	2.94	3476	2.47	1660	perlin	23.00	7312	22.77	5060
fft	1.19	5629	1.15	3460	qsort	0.93	3369	0.89	1664
fftsp	0.74	5788	0.63	3600	sha1	0.84	5830	0.70	3536
fftw	0.89	3999	0.67	2316	sha3	0.91	8798	0.90	5992
fib	0.64	2540	0.58	1092	siphash24	2.92	7301	2.52	4812
integr	0.03	3041	0.03	1528	spectral	5.08	3825	5.07	2064
knucleotide	0.99	7956	0.88	5348	vmach	1.45	4074	1.33	2304

The first row is for the specs and semantics of the new types of programs and various properties about them. The second row is for the linking infrastructure, which contains the definitions of new linking operators, syntactical equivalence, and various properties about them and also about linking in CompCert in general. The remaining rows show the effort for developing every new compiler pass in CompCertELF, including the proofs of semantics preservation for closed programs and the proofs of separate compilation. Unsurprisingly, the development for the generation of symbol tables is the most complicated one: it accounts for almost half of the total LOC. The development for instruction encoding takes the second place due to the complexity in establishing consistency between the encoder and decoder. The specs and proofs for the remaining passes are relatively straightforward and account for a lesser portion of the total effort.

Performance. For this we use the test cases under the directory `test/c` of CompCert. Each of the C file in this directory is a self-contained program. We ran Ubuntu18.04 on a machine with Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz and 16 GB memory. We first compiled these test cases to relocatable binary ELF (`.o` files) using CompCertELF and then linked them into executable ELF files (the linking with C library is carried out by GNU `ld` for reasons described before). We measured the executing time (in seconds) and the sizes of generated `.o` files (in bytes) and compared the results with those obtained by using the original CompCert (v3.0.1, which uses the GNU assembler).

We found that our relocatable ELF files are bigger than those generated by CompCert as shown in Table 2. This is because our instruction encoding is bare-bones: we encode each type of assembly instructions in CompCert into its most general form with a fixed length, while the GNU assembler will encode an instruction into specialized and more compact forms depending on the specific arguments the instruction has.

To measure the executing time, we run each program in the benchmark 30 times and take the average execution time; the results are shown in Table 2. Note that for all the programs, the variance in their execution time among these 30 runs is less than 0.01s. As a result, we believe that it is sufficiently accurate to analysis their performance by using the average time. Overall, as we expected, the ELF programs output by CompCertELF run slower than those output by CompCert because our assembler is bare-bones compared to the GNU assembler. However, some programs such as aes, chomp and integr have close or equal performance. We conjecture that this is because *1)* they are small enough that the sophistication of GNU assembler does not bring much increase in performance, and *2)* they are all computation-intensive programs while the ELF programs generated by CompCertELF perform better (and sometimes as good as those generated by the original CompCert) when they are computation-intensive rather than IO-intensive. The latter is likely caused by the lack of support for `.rodata` sections by CompCertELF which is important for good I/O performance since read-only data will seldom be exchanged out of the processor cache compared to writable data. We plan to address the above problems by developing a more sophisticated assembler which will require more engineering effort.

Separate Compilation. To demonstrate CompCertELF's capability for separate compilation, we have successfully used it to compile and link some small examples like the running example in Fig. 3. We have also used it to compile the big test case `Compression` in CompCert with multiple C files in `test/compression`. `Compression` consists of a collection of programs that do sampling using an arithmetic encoding/decoding library. CompCertELF compiles each of the 12 source files separately into the ELF object (`.o`) files and then links them into executable files. This shows CompCertELF can successfully perform separate compilation like any production C compiler.

6 RELATED WORK

We discuss related work from the following perspectives:

Compilation to Machine Code. Although there exists a lot of work on extending CompCert, very few of them have considered further compilation of assembly programs into binary machine code. A major obstacle is that CompCert's assembly semantics use an unbounded list of memory blocks to represent the stack instead of a finite and continuous stack like on real machines. There has been previous work on translating from the infinite memory model of CompCert to some kinds of finite memory models. CompCertS [Besson et al. 2017] extends CompCert to support low-level manipulation of pointer values in a finite memory space. CompCert-TSO [Sevcík et al. 2013, 2011] extends CompCert with concurrency based on the shared memory model TSO by building a notion of finite memory into all levels of CompCert. Mullen et al. [Mullen et al. 2016] defines a lower-level semantics for CompCert assembly that models pointers as 32-bit values for the verification of peephole optimizations. However, none of them supports merging of the stack blocks into a finite and continuous stack and elimination of pseudo-instructions for stack manipulation that do not exist on real machines.

Stack-Aware CompCert [Wang et al. 2019] is the first extension that removes the above obstacles. Still, Stack-Aware CompCert compiles to an internal representation of closed programs called MC which is then translated into executable ELF format using an unverified program in OCaml. This program uses the RockSalt x86 model [Morrisett et al. 2012; Tan and Morrisett 2018] for instruction encoding. RockSalt is not suitable for verified compilation because its grammar is ambiguous: the same sequence of bytes may be decoded into different instructions with no obvious semantic relation between them, making it difficult to prove the consistency between instruction encoding and decoding. As pointed out by Tan and Morrisett [2018], one source of ambiguity comes from the "alternative" (or "sum") grammar of the form `alt g1 g2`. For example, consider the grammar `g` which is `alt (char c) (char c)`. The result of parsing a single character `c` with `g` (represented by `parse g c`) is non-deterministic and may return either `inl c` or `inr c`. Semantically, these two values may be different because they have different tags (which may correspond to different opcodes in the case of instruction encoding). Specifically, the following property holds where `pretty−print` is the encoding function in RockSalt:

$$\exists\, \mathtt{r}, \mathtt{pretty{-}print}\ \mathtt{g}\ (\mathtt{inl}\ \mathtt{c}) = \mathtt{Some}\ \mathtt{r} \wedge \mathtt{parse}\ \mathtt{g}\ \mathtt{r} = \mathtt{inr}\ \mathtt{c}.$$

Comparing to RockSalt, the decoder in CompCertELF is deterministic. Application of first encoding and then decoding to instructions must result in instructions that are semantically equivalent to the original inputs. Therefore, the above example is forbidden with our instruction encoding and decoding.

The official release of CompCert provides a tool named `Valex` for checking the correspondence between CompCert's assembly with its binary outputs based on translation validation [Leroy 2020]. Since this tool is not formally verified, it does not provide a formal guarantee of correctness like CompCertELF.

Cerco (Certified Complexity) [Amadio et al. 2014] is a compiler whose front-end is based on CompCert and whose back-end is developed independently of CompCert. Its main goal is to relate the time and space complexity of the compiled object code with the source code so as to perform complexity analysis of object code at the source level. For this purpose, it does not implement complicated optimizations like CompCert and only generates object code for 8-bit processors including Intel 8051/8052. It also has not considered compilation to a binary file format or the support of separate compilation.

CakeML [Kumar et al. 2014; Tan et al. 2019] is a formally verified compiler for a substantial subset of Standard ML. As a compiler for functional languages, its compilation chain is very different from that of CompCert. It outputs binary object code for various architectures such as x86, ARM and RISC-V. The latest work on CakeML also develops a verified stack that targets a verified processor called Silver [Lööw et al. 2019]. Comparing to CompCertELF, CakeML only works for closed programs and uses an internal format for representing object code instead of a standard binary file format [Tan et al. 2019]. Concretely, it translates closed ML programs into "flat labeled" assembly in a language called LabLang, i.e., a straight line of assembly code with labels as the targets for jump and call instructions. It then translates the assembly code into a sequence of bytes representing executable machine code on a designated target architecture—this binary machine code does not contain information for linking and relocation. As a result, the correctness theorem of CakeML does not support separate compilation of linkable file formats.

Compositional Compilation. Verified separate compilation [Kang et al. 2016] is the only approach to compositional compilation officially supported by CompCert. However, its current realization is not very useful because it only covers compilation to assembly languages with an unrealistic machine model. Stack-Aware CompCert partially addressed this problem by compiling to assembly languages with a single and continuous stack. It does support a kind of separate compilation for MC programs. The idea is to keep enough information in MC programs so that they can be decompiled back to RealAsm programs, then to link them using CompCert's linker and recompile the results into MC programs. However, it has not considered the problem of compiling to relocatable object files or performing actual linking at the binary level. CompCertELF is the first one that supports separate compilation down to binary object files like the production C compilers, and its verification.

Verified separate compilation is limited in that it does not support compilation of heterogeneous modules (i.e., modules written in different languages). Various techniques for more general compositional compilation in CompCert have been proposed. Notable examples include Compositional CompCert [Stewart et al. 2015; Stewart 2015], CompCertX [Gu et al. 2015], CASCompCert [Jiang et al. 2019] and CompCertM [Song et al. 2020]. All these extensions compile only to CompCert's assembly language. Our work addresses the problem of further compiling CompCert's assembly programs into object files. Therefore, it should be possible to combine our work with those techniques to realize end-to-end verified compilation of heterogeneous modules. A straightforward solution would be to extend the generally compositional compilers with the support of finite stacks using the ideas of Stack-Aware CompCert and combine the extended compilers with our verified assembler using verified separate compilation. However, there are new theoretical and technical challenges for CompCert to support full-blown interoperation between C programs, assembly code and ELF binaries together with compositional compilation and linking of these programs. This is a topic worth of a separate paper and is left for future work.

Formalization of the ELF Format. The final output of CompCertELF is in a subset of ELF. The linker provided by CompCertELF also works on this subset. Kell et al. [2016] have provided the first formalization of the static linking of ELF in the specification language Lem and the proof assistant Isabelle/HOL which covers a significantly larger subset of ELF. However, their formalization of ELF linking has only been used to verify the correctness of relocation for a single instruction program. By comparison, the correctness of our ELF linker is given by the fact that it supports the separate compilation theorem of CompCertELF (i.e., Theorem 12).

In theory, because the ELF linker from Kell et al. aims to work with ELF files in their most general form, it should subsume our linker which only works for the sub-format generated by CompCertELF. We conjecture that we can develop further compilation passes to transit from our ELF linker to theirs'. However, this requires a Coq implementation of their linker which—to the best of our knowledge—is currently lacking. Therefore, the connection between the two is left for future work.

7 CONCLUSION

We have introduced CompCertELF, a verified compiler that transforms C programs into ELF object files and a lightweight approach to supporting verified separate compilation to relocatable object files. To the best of our knowledge, this is the first extension of CompCert that has achieved these goals. The future work will be to connect CompCertELF with other verified compositional compilers, to generalize our compiler to support other architectures besides x86-32 and to connect it with verified operating systems such as CertiKOS [Gu et al. 2016].

ACKNOWLEDGMENTS

We would like to thank Jérémie Koenig and anonymous referees for helpful feedback that improved this paper significantly. This research is based on work supported in part by NSF grants 1521523, 1715154, and 1763399. The fourth author is a co-founder of and has an equity interest in CertiK Global Ltd. CertiK has licensed Yale University's intellectual property, which is related to the NSF grants 1521523, 1715154, and 1763399. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2014. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis*, Ugo Dal Lago and Ricardo Peña (Eds.). Springer International Publishing, Cham, 1–18. https://doi.org/10.1007/978-3-319-12466-7_1

Andrew Appel. 2011. Verified Software Toolchain. In *Proc. 20th European Symposium on Programming (ESOP'11)*, Gilles Barthe (Ed.). LNCS, Vol. 6602. Springer, Saarbrucken, Germany, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position Paper: the Science of Deep Specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20160331. https://doi.org/10.1098/rsta.2016.0331

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017. CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics. In *Interactive Theorem Proving (ITP'17)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer International Publishing, Cham, 81–97. https://doi.org/10.1007/978-3-319-66107-0_6

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

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jeremie 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

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, NY, USA, 111–125. https://doi.org/10.1145/3314221.3314595

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proc. 43rd ACM Symposium on Principles of Programming Languages (POPL'16)*. ACM, New York, 178–190. https://doi.org/10.1145/2837614.2837642

Stephen Kell, Dominic P. Mulligan, and Peter Sewell. 2016. The Missing Link: Explaining ELF Static Linking, Semantically. In *Proc. 2016 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, New York, NY, USA, 607–623. https://doi.org/10.1145/3022671.2983996

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proc. 41st ACM Symposium on Principles of Programming Languages (POPL'14)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2005–2020. The CompCert Verified Compiler. http://compcert.inria.fr/.

Xavier Leroy. 2009a. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2009b. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

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

Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformation. *Journal of Automated Reasoning* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proc. the 40th ACM Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proc. 2012 ACM Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, USA, 395–404. https://doi.org/10.1145/2254064.2254111

Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *Proc. 37th ACM Conference on Programming Language Design and Implementation (PLDI'16)*. ACM, New York, NY, USA, 448–461. https://doi.org/10.1145/2980983.2908109

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-Memory Concurrency and Verified Compilation. In *Proc. 38th ACM Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, 43–54. https://doi.org/10.1145/1926385.1926393

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

Gordon Stewart. 2015. *Verified Separate Compilation for C*. Ph.D. Dissertation. Princeton University.

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

Gang Tan and Greg Morrisett. 2018. Bidirectional Grammars for Machine-Code Decoding and Encoding. *Journal of Automated Reasoning* 60, 3 (2018), 257–277. https://doi.org/10.1007/s10817-017-9429-1

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The Verified CakeML Compiler Backend. *Journal of Functional Programming* 29 (2019), e2. https://doi.org/10.1017/S0956796818000229

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