An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code

YUTING WANG, Yale University, USA
PIERRE WILKE, Yale University, USA and CentraleSupélec, France
ZHONG SHAO, Yale University, USA

A key ingredient contributing to the success of CompCert, the state-of-the-art verified compiler for C, is its block-based memory model, which is used uniformly for all of its languages and their verified compilation. However, CompCert's memory model lacks an explicit notion of stack. Its target assembly language represents the runtime stack as an unbounded list of memory blocks, making further compilation of CompCert assembly into more realistic machine code difficult since it is not possible to merge these blocks into a finite and continuous stack. Furthermore, various notions of verified compositional compilation rely on some kind of mechanism for protecting private stack data and enabling modification to the public *stack-allocated data*, which is lacking in the original CompCert. These problems have been investigated but not fully addressed before, in the sense that some advanced optimization passes that significantly change the ways stack blocks are (de-)allocated, such as tailcall recognition and inlining, are often omitted.

We propose a lightweight and complete solution to the above problems. It is based on the enrichment of CompCert's memory model with an abstract stack that keeps track of the history of stack frames to bound the stack consumption and that enforces a uniform stack access policy by assigning fine-grained permissions to stack memory. Using this enriched memory model for all the languages of CompCert, we are able to reprove the correctness of the *full* compilation chain of CompCert, including all the optimization passes. In the end, we get Stack-Aware CompCert, a complete extension of CompCert that enforces the finiteness of the stack and fine-grained stack permissions.

Based on Stack-Aware CompCert, we develop CompCertMC, the first extension of CompCert that compiles into a low-level language with flat memory spaces. Based on CompCertMC, we develop Stack-Aware CompCertX, a complete extension of CompCert that supports a notion of compositional compilation that we call *contextual compilation* by exploiting the uniform stack access policy provided by the abstract stack.

CCS Concepts: • **Theory of computation → Program verification**; **Abstraction**; • **Software and its engineering → Software verification**; **Semantics**; **Functionality**;

Additional Key Words and Phrases: memory model, abstract stack, certified compilers, compositional compilation, machine code generation

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

62

Authors' addresses: Yuting Wang, Yale University, USA, yuting.wang@yale.edu; Pierre Wilke, Yale University, USA , CentraleSupélec, France, pierre.wilke@centralesupelec.fr; Zhong Shao, Yale University, USA, zhong.shao@yale.edu.

2475-1421/2019/1-ART62
https://doi.org/10.1145/3290375

1 INTRODUCTION

In the field of software verification using formal methods, verification is usually carried out in the high-level source languages. Verified compilers are then used to transport the correctness properties at the source level to executable code to obtain an end-to-end guarantee which is important for critical software systems such as OS kernels [Gu et al. 2015, 2016].

The state-of-the-art verified compiler for C is CompCert [Leroy 2009a] which has been successfully applied to many software verification projects. However, the vanilla CompCert is restricted in several aspects, making it less flexible for real-world verification projects. First, CompCert does not guarantee the correct compilation to executable machine code. Its verified compilation chain outputs a form of assembly code which is still quite far away from the actual assembly code; in particular, it does not operate over a finite stack. Second, CompCert only supports a limited form of modular verification known as *verified separate compilation* (proposed by Kang et al. [2016]), meaning if the modules are written in the same source language and compiled down to the same target language through the same compilation passes, then the behaviors of the *syntactically linked* modules are preserved by compilation. Verified separate compilation is not applicable to compilation and linking of *heterogeneous modules*, *i.e.*, modules written in different languages that are compiled through different compilation passes, which are common in software systems. For instance, the code for context switch in an OS implementation is usually written in the assembly language and needs to be linked with other C modules compiled to assembly code [Gu et al. 2015, 2016].

We observe that the above restrictions can largely be attributed to the lack of an explicit notion of stack with appropriate permissions for controlling stack access in the memory model of CompCert. CompCert uses a block-based memory model uniformly for all of its languages [Leroy et al. 2012; Leroy and Blazy 2008], where the memory is divided into *blocks* isolated from one another. In particular, the "stack" in the target assembly language is represented as an *unbounded* list of blocks. Further compilation to machine code is difficult because it is not possible to merge this list into a stack with finite space. For verified compilation of heterogeneous modules, a major problem is to protect the private data on the stack belonging to one module from being modified by another module. To solve this problem, it is required to have some mechanism to enforce the said protection of stack data, which is lacking in the vanilla CompCert.

Our goal of this work is to significantly relax the aforementioned restrictions in CompCert by extending it with the following features:

- *Stack-awareness.* We would like to extend the compilation chain of CompCert to be aware of the existence of a finite stack with permissions for protecting private data on the stack. This will provide the basis for the following extensions.
- *Compilation to flat memory spaces.* We would like to further compile CompCert assembly code into a form that closely mirrors the actual machine code. In particular, it should use flat memory spaces for representing the stack, code and data.
- *Completeness.* The extensions should be complete. That is, they should support all the features of CompCert, including *stack-allocated data*—an important concept for the compilation of C programs that characterizes stack data modifiable outside of the current function call—and the full compilation chain containing the non-trivial optimization passes that transform the structure of the stack, such as inlining and tailcall recognition.
- *Verified separate compilation.* The extensions should at least support verified separate compilation, the default approach to modular verification in CompCert.
- *Contextual compilation.* We also would like to support contextual compilation, a notion of modular verification for compilers that is stronger than verified separate compilation,

denoting that any source module starting from an arbitrary context (i.e., arbitrary memory state and an arbitrary entry point) can be correctly compiled into the target language.

Our key idea is to instrument the memory model of CompCert with a built-in abstraction of the stack 1) to keep track of the concrete stack consumption at all levels of compilation, so as to prove the preservation of stacks bounds by compilation, which enables verified compilation to lower-level code with a finite stack, and 2) to enrich the memory blocks with stack permissions and impose a uniform access policy to prevent inadvertent modification of private stack data across modules. In the following paragraphs, we explain our key novelties in the application of this idea to support the above features in CompCert.

Stack-awareness. Our starting point is the definition of an abstract data type in the memory model of CompCert to represent the stack, that we call the *abstract stack*. It consists of a collection of abstract frames that record information about the stack frames allocated by function calls.

By recording the size of the concrete stack frames in the abstract frames and providing methods in the memory model to push and pop abstract frames onto the abstract stack, we are able to instrument the semantics of all the languages of CompCert to keep track of and bound the concrete stack consumption. Then, we prove that all CompCert's passes preserve the stack consumption, *i.e.* the stack consumption of the target program is no larger than that of the source program, by generalizing *memory injections*—the invariant between memory states for proving compilation correct—to relate abstract stacks in the source and target languages.

By recording permissions in the abstract frames for distinguishing public and private regions of stack frames, we are able to tell *stack-allocated data* which are modifiable outside the current function (*e.g.*, local variables whose addresses are passed as arguments to other functions) apart from the private data which are only modifiable by the current function (*e.g.*, callee-save and spilled register values). Furthermore, by restricting in the memory model which accesses are permitted with respect to the abstract stack, we enforce a uniform access policy on the stack. We prove that all CompCert's passes preserve the stack access policy by augmenting memory injections to capture preservation of stack permissions.

Compilation to flat memory spaces. By exploiting the property that the CompCert's passes now preserve stack consumption, we merge the stack blocks in CompCert's assembly language into a single finite stack, and prove that any source program that does not overflow the concrete stack can be correctly compiled to a single-stack assembly program, from which we further compile to a language with data and code collapsed into flat memory spaces.

We still allow for dynamic allocation of an unbounded number of heap blocks besides the static blocks for code, data and stack. Note that most of the existing verification works based on CompCert (e.g. Verified Software Toolchain [Appel 2011] and CertiKOS [Gu et al. 2016, 2018]) actually do not rely on the axiomatised heap blocks as provided in CompCert, but rather allocate a global array that serves as a heap and provide their own heap management functions that operate on that global array. This way, all the memory is statically known and we can merge all parts of the memory into a flat memory space. Of course, we may take a similar approach to modeling a finite heap by introducing an abstract data type for the heap and operations on this data type into CompCert's memory model. We still need to evaluate the benefits and explore the feasibility of this extension, which are left for future work.

Completeness. The above extensions support stack-allocated data through stack permissions, as described above. They also support the full compilation chain of CompCert. For this, the main difficulty lies in proving the preservation of stack consumption. This is relatively straightforward for most compiler passes, where the call-return structure is preserved between the source and the target programs. However, more work is needed for optimizations such as inlining and tailcall

recognition. Because these passes may merge function calls or reorder allocations of frames, we need to generalize the abstract stack to record the *history* of abstract frames and generalize memory injections to capture the merging and reordering of corresponding stack frames. The details are described in Sec. 3 and Sec. 4.

Verified separate compilation. The above extensions also support verified separate compilation by following the same idea described in Kang et al. [2016]. The work amounts to defining the notions of syntactic linking for our new languages with flat memory spaces and proving that the new compilation passes commute with syntactic linking.

Contextual compilation. Based on the previous extensions, we further support contextual compilation. For this, the key is to make sure the memory regions for the stack in different modules are used in a consistent way throughout the compilation. We enforce this consistency by describing the semantics of every module using our stack-aware memory model and exploiting the uniform stack access policy imposed by our abstract stack. The details are described in Sec. 6.

1.1 Comparison with Related Work

There already exists a number of extensions to CompCert that aim to support some of the aforementioned features. We briefly discuss the key differences between the most relevant related work and our work. We will give a detailed comparison later in Sec. 7.

The distinguishing feature of our work is that it is based on the unique idea of enriching CompCert's memory model with an abstraction of the stack for tracking stack consumption and stack permissions. We apply this idea to verify the compilation of C to a language with flat memory spaces, which seems to be a first.

Carbonneaux et al. [2014] have developed *Quantitative CompCert* which supports compilation to a finite stack. Their key idea is to augment the event traces generated by execution with call and return events attached with the sizes of concrete stack frames, which will be used to calculate the stack consumption of a given trace. They then prove that if the stack consumption for all traces are preserved, then the source program can be correctly compiled into an assembly program with a finite stack. However, they have not tackled the problem of compilation to flat memory spaces or any form of compositional compilation. Moreover, they have not yet shown the event-trace based approach is flexible enough to support inlining or tailcall recognition.

Stewart *et al.* have developed *Compositional CompCert* [Stewart 2015; Stewart et al. 2015] which supports a very general notion of modular verification for compilers known as *verified compositional compilation* (VCC). They can prove the correct compilation of whole programs by composing the correctness proofs of the compilation of individual modules. However, they have not solved the problem of stack merging or compilation to flat memory spaces. Similar to Quantitative CompCert, they have not yet shown their approach supports inlining or tailcall recognition.

We briefly compare the different notions of compositional compilation for CompCert. VCC is more powerful than contextual compilation in that it allows for recursive calls between heterogeneous modules while contextual compilation does not. Contextual compilation is much more powerful than verified separate compilation because it allows for interoperation between heterogeneous contexts and programs. It has been proven adequate for verification of non-trivial system software such as OS kernels as manifested in the CertiKOS project [Gu et al. 2015, 2016], where the linking between arbitrary assembly contexts and C modules and the isolation of stack resources between modules are essential to make modular verification possible.

1.2 Contributions and Overview

We summarize our contributions as follows, all of which have been fully formalized in Coq:

(1) We develop an enhancement to CompCert's memory model with an abstract stack which keeps track of the history of stack frames and uses the history to calculate the stack consumption. We further enrich the abstract stack with permissions to enforce a uniform stack access policy through the interfaces of CompCert's memory model.
(2) Using our new memory model, we implement *Stack-Aware CompCert*, an extension of CompCert that bounds the size of the stack and distinguishes between the public data from private data on the stack at all levels of compilation. We reprove all the passes of CompCert, including the challenging optimizations such as inlining and tailcall recognition, by extending CompCert's memory injections to take into account the preservation of stack consumption and stack permissions.
(3) Based on Stack-Aware CompCert, we develop *CompCertMC*, the first extension to the *full* compilation chain of CompCert (v.3.0.1) that supports compilation to a lower-level language called *MC*. In MC the code, data and stack are laid out in finite flat memory spaces like the actual machine code. Programs written in this language can therefore be transparently converted into actual machine code using formal models of machine architectures (We use the RockSalt [Morrisett et al. 2012; Tan and Morrisett 2018] x86 machine language encoder to demonstrate this process, whose verification is left for future work).
(4) Based on CompCertMC, we implement *Stack-Aware CompCertX*, an extension to CompCert that supports *contextual compilation*. The proof makes critical use of the uniform stack access policy provided by the new memory model and the injection between abstract stacks that takes stack permissions into account. Stack-Aware CompCert, CompCertMC and Stack-Aware CompCertX support all the features of the original CompCert, with the exception that Stack-Aware CompCertX supports contextual compilation which is more powerful than verified separate compilation.

The rest of the paper is organized as follows. First, Sec. 2 introduces the concepts of CompCert necessary for our discussion. Then, the next four sections (Sec. 3 to Sec. 6) elaborate on the above contributions in sequence. Finally, we discuss related work in Sec. 7 and conclude in Sec. 8.

2 AN INTRODUCTION TO COMPCERT

This section presents the CompCert compiler. We first describe the memory model that the semantics of the languages are based upon. It includes relations between memory states known as memory injections that are central to the correctness proof of the individual compiler passes. We then describe the correctness of compilation including the support of separate compilation. We finally discuss the optimizations that can significantly change the structure of the stack, in particular, inlining and tailcall recognition. For a more thorough coverage of CompCert, interested readers can consult Blazy et al. [2006]; Leroy [2009a,b]; Leroy et al. [2012].

2.1 Basics of the Memory Model of CompCert

The memory in CompCert is modeled as a collection of *blocks*, where each block is an array of abstract bytes. A pointer is a pair (b, o) consisting of a block identifier b and an integer offset o within that block. This model captures the important aspects of semantics of pointers in C programs. For instance, in CompCert C every local variable of a function is associated with a distinct block, therefore preventing pointer arithmetic across different variables.

The contents of these memory blocks are values represented by the following discriminated union of undefined values, 32-bit or 64-bit integers, single- or double-precision floating point numbers or pointers:

$$\mathcal{V} \triangleq \mathtt{Vundef} \mid i_{32} \mid i_{64} \mid f_{32} \mid f_{64} \mid (b, o)$$

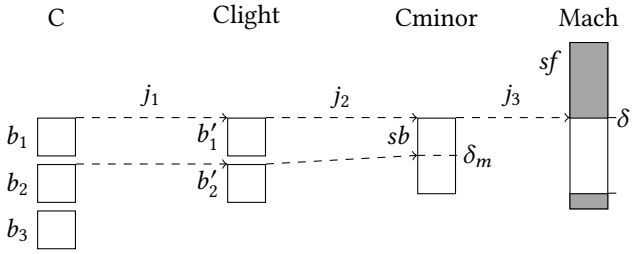


Fig. 1. Evolution of the blocks in a stack frame

Undefined values are used to indicate uninitialized memory and also to represent the result of meaningless operations such as subtraction between two pointers in different blocks. CompCert also associates permissions with each offset of each block, stating whether those locations have no permission (hence no operation is allowed on them), are readable (they can be read from), are writable (they can be read from and written to), or are freeable (they can be read from, written to, and freed).

The main operations on memory states are listed below, where $\mathcal{M}$ is the type of memory states:

$$\begin{aligned}
&\texttt{alloc} : \mathcal{M} \to \texttt{Z} \to \texttt{Z} \to \mathcal{M} \times \texttt{block}\\
&\texttt{free} : \mathcal{M} \to \texttt{block} \to \texttt{Z} \to \texttt{Z} \to \lfloor \mathcal{M} \rfloor\\
&\texttt{load} : \mathcal{M} \to \texttt{chunk} \to \texttt{block} \to \texttt{Z} \to \lfloor \mathcal{V} \rfloor\\
&\texttt{store} : \mathcal{M} \to \texttt{chunk} \to \texttt{block} \to \texttt{Z} \to \mathcal{V} \to \lfloor \mathcal{M} \rfloor
\end{aligned}$$

Most of these operations return an `option` type, meaning they can fail, in particular if the permission requirements are not met. We write $\lfloor \bullet \rfloor$ for both the type itself and the `Some` constructor, and we write $\emptyset$ for the `None` constructor. Only `alloc` always succeeds, therefore modeling an infinite memory. This operation allocates a new block. We say that a block is *valid* if it has been allocated. Operations `load` and `store` are parameterized by a chunk κ which designates the size, type and signedness of the value stored at the given memory location. The chunk κ dictates encoding and decoding of values, according to their size (denoted by $|\kappa|$) and signedness. It is easy to understand the function of each operation from its signature. For example, the operation `store` $m\ \kappa\ b\ o\ v$ stores a value v according to the chunk κ into the locations $(b, o), \ldots, (b, o + |\kappa| - 1)$. It succeeds only if those locations have the writable permission.

2.2 Memory Injections

CompCert compiles C programs into assembly programs, going through 10 intermediate languages and 20 compiler passes (as of CompCert v3.0.1). During this compilation, the structure of the memory is transformed, in particular the memory that corresponds to stack blocks, as shown in Fig. 1. In CompCert C, for a given function, every local variable and function parameter is allocated in a separate fresh block (b_1, b_2 and b_3). An early compilation pass, SimplLocals – at the Clight level – lifts scalar local variables whose address is never taken to temporary variables. One motivation for doing so is that the optimizations later in the compilation chain will be able to operate more aggressively on temporaries or pseudo-registers rather than on the contents of the memory. The memory blocks for such lifted variables are discarded, as is the case for b_3. Later, in the Cminor language, the blocks for the remaining variables are merged into a single stack block *sb* that hold the so-called *stack-allocated data*. Finally, in the Mach language, the structure of the stack frame is completely laid out and the stack block becomes a part of the larger stack frame *sf*, which also contains data such as the return address of the function or the callee-save registers (pictured in gray), introduced by the pass called Stacking.

To verify the correctness of the compilation passes, CompCert introduces so-called *memory injections* (j_1, j_2, j_3 in Fig. 1) to track the relationship between the blocks in the source and target programs. A memory injection is a partial function from blocks to locations, *i.e.* $\texttt{block} \to \lfloor \texttt{block} \times \mathbb{Z} \rfloor$. This is a partial function to account for the fact that some blocks may be pulled out of memory and have no counterpart in the target memory. This is the case for example of block b_3 in Fig. 1, *i.e.* $j_1(b_3) = \emptyset$. In the other cases, the resulting block and offset specifies where the source locations are mapped. For example, we have $j_1(b_1) = \lfloor b_1', 0 \rfloor$, meaning that block b_1 has been renamed into b_1' but the offset is unchanged. We also have $j_3(sb) = \lfloor sf, \delta \rfloor$. This means that the source location (sb, o) is mapped to the target location $(sf, o + \delta)$, for any o.

Value injection. The source and target values are related by the value injection relation, noted $\hookrightarrow_j$, and defined as follows:

$$\frac{v \neq (b, o)}{v \hookrightarrow_j v} \qquad \frac{}{\texttt{Vundef} \hookrightarrow_j v} \qquad \frac{j(b) = \lfloor b', \delta \rfloor}{(b, o) \hookrightarrow_j (b', o + \delta)}$$

The injection is reflexive for regular (non-pointer) values. A pointer value in the source language is relocated according to the memory injection. Undefined values inject into any value in the target language. The purpose of specializing undefined values is to capture for instance the semantics of pointer operations, which *gets more defined* after injection. Consider for instance the pointer subtraction $(b_2', 0) - (b_1', 0)$ at the Clight level, which evaluates to `Vundef`. At the Cminor level, the pointer subtraction becomes $(sb, \delta_m) - (sb, 0)$, which evaluates to δ_m.

Memory injection. The value injection relation is lifted to memory states: given two memory states m_1 and m_2, there is a memory injection between m_1 and m_2 by j (noted $m_1 \hookrightarrow_j m_2$), if (essentially) the permissions of corresponding locations are preserved and the contents of corresponding locations are in value injection. There are also well-formedness properties of these injection functions: only valid blocks are mapped by j, no two distinct source locations with non-empty permissions are injected into the same target location, etc.

CompCert's memory model comes with a set of properties that describe how the various memory operations affect the memory contents and permissions, and how the memory injections and the memory operations interact across a program's execution. The proof of correctness of the entire compiler relies on those properties.

2.3 The Correctness of Compilation

The CompCert compiler takes C programs as input and translates them through a sequence of compilation passes into the CompCert assembly language, as depicted in Fig. 2 (the first pass is an identity transformation that translates C programs with a non-deterministic semantics into those with a deterministic one).

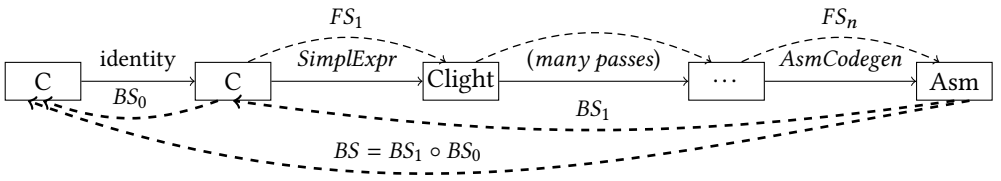


Fig. 2. The simulation relations for the compiler passes of CompCert

The goal is to prove that every observable behavior of the compiled assembly program can be exhibited at the C source level. This is derivable from a backward simulation relation between the

C and assembly programs, *i.e.*, there is an invariant between the source and target program states such that, starting from states related by the invariant, every step of an execution of the target program can be simulated by zero or more steps of an execution of the source program with the invariant being preserved by these steps. For the compiler C consisting of the sequence of passes $C_1, \ldots, C_n$, it suffices to prove that for each C_i there is a backward simulation relation between its source and target programs. Then, by transitivity of simulations, we collapse the sequence of backward simulations into that for C. However, backward simulations are usually more difficult to prove than forward simulation, *i.e.*, their dual notations, because one step of execution may become several steps after compilation and it may be difficult to relate the intermediate states between these steps to the source ones. Fortunately, in many situations, it suffices to prove forward simulation for a compilation pass because it can be flipped into a backward simulation.[1] This idea has been used to prove CompCert correct, as depicted in Fig. 2. First, a backward simulation (BS_0) is established for the first pass. Then, a sequence of forward simulations ($FS_i (1 \leq i \leq n)$) is established for the remaining passes, which is composed into a single forward simulation by transitivity and flipped into a backward simulation (BS_1). Finally, the backward simulations are composed to form the backward simulation (BS) for the whole compiler.

We shall write $[\![P]\!]$ to represent the small-step operational semantics of program P and $[\![T]\!] \sqsubseteq [\![S]\!]$ to represent the backward simulation between the target program T and the source program S. The correctness theorem of CompCert is then stated as follows:

Theorem 2.1. *Given the CompCert compiler C and a complete C program P, $[\![C(P)]\!] \sqsubseteq [\![P]\!]$.*

Its proof follows the above discussion. The details can be found in Leroy [2014]. The critical steps for proving a simulation relation are to find an invariant between the states of the source and target programs and to show it holds throughout their execution. Note that this invariant usually contains a memory injection between the memory states of the source and target programs.

Theorem 2.1 is only for the compilation of *complete* C programs, *i.e.*, programs without undefined external references. CompCert also supports *verified separate compilation* for incomplete C programs via the commutative property between compilation passes and syntactic linking. In every language of CompCert, a program consists of a set of definitions each of which is a mapping from an identifier to a global variable, an internal function, an external function or an undefined object. Programs are linked by taking the union of their definitions such that:

- If there is a single non-external definition for an identifier, then all the external and undefined definitions are resolved to it;
- If there are multiple non-external definitions for an identifier, the linking fails.

We write $P_1 \oplus \ldots \oplus P_k$ for the result of syntactic linking of programs $P_1, \ldots, P_k$. Verified separate compilation relies on the following property for each compiler pass C_i:

$$C_i(P_1 \oplus \ldots \oplus P_k) = C_i(P_1) \oplus \ldots \oplus C_i(P_k)$$

The correctness theorem of separate compilation states that if C modules are compiled to assembly using the CompCert compiler, then the syntactically linked target modules refine the syntactically linked source ones, as follows.

Theorem 2.2. *Given the CompCert compiler C and modules $P_1, \ldots, P_k$ written in the C language,*

$$[\![C(P_1) \oplus \ldots \oplus C(P_k)]\!] \sqsubseteq [\![P_1 \oplus \ldots \oplus P_k]\!].$$

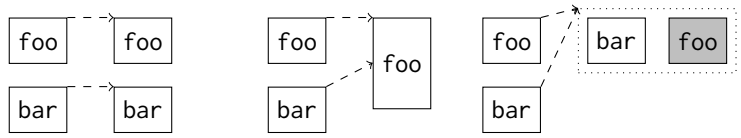
Proof. By commutativity between compilation and syntactic linking, $C(P_1) \oplus \ldots \oplus C(P_k) = C(P_1 \oplus \ldots \oplus P_k)$. We conclude by using Theorem 2.1. □

[1] The requirement is that the target language of this pass is *determinate* and the source language is *receptive*. Interested readers can find the details in Sevcík et al. [2011].

2.4 Modification of the Stack by Tailcall Recognition and Inlining

Most of CompCert's passes preserve the call-return structure of programs and therefore preserve the shape of the stack. However, the function inlining and tailcall recognition optimizations of CompCert modify this structure and multiple source frames may inject into the same target frame (at those stages a stack frame is represented by a block containing stack-allocated data). This situation is illustrated by Fig. 3. Consider a regular function call from function `foo` to `bar`, and 3 possible compilations or optimizations of that function call. First, the call is kept as a regular call and the structure of the stack is unchanged. This is the most common case in CompCert passes. The resulting memory injection is depicted in Fig. 3b. Second, the call to `bar` is inlined. As a consequence, the target stack frame is now the concatenation of the two source frames. At the entry point of `bar` in the target, both the source frames for `foo` and `bar` inject into this single frame, as shown in Fig. 3c. Third, the call to `bar` is transformed into a tail call. In CompCert, this can happen only if the caller's frame is of size 0. In our case, the size of stack-allocated data for `foo` must be 0. At the tailcall of `bar`, the frame for `foo` is deallocated. Then, at the entry point of `bar`, the frame for `bar` is allocated and both source frames for `foo` and `bar` inject into the target frame for `bar`, as shown in Fig. 3d (where the frame with a gray background is deallocated). The frame of `foo` can inject into that of `bar` because it is a block of size 0, hence has no permission.

```
int bar(){ BAR; }
int foo(){
  FOO;
  return bar();
}
```



(a) Example functions (b) Structure unchanged (c) Inlining of bar (d) Tail call of bar

Fig. 3. Stack injection for inlining and tailcall

A major challenge to the implementation of our extensions is to characterize the transformations of the stack incurred by inlining and tailcall recognition optimizations, as we shall see in the following sections.

3 AN ABSTRACT STACK IN COMPCERT

This section introduces the heart of our approach: the formalization of an abstract stack within the memory model of CompCert. This abstract stack will serve two purposes: ensuring that the stack fits in a finite memory region of predetermined size; and enforcing a stack access policy to protect private regions while still allowing modifications to public regions.

First, we describe our formalization of the abstract stack. Then we define the size of such abstract stacks, and we express a stack access policy based on the abstract stack.

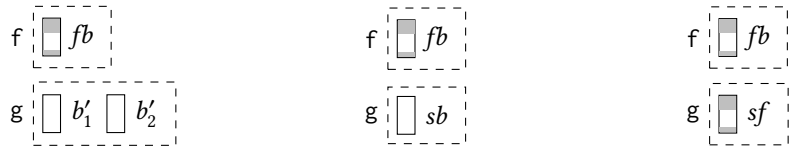
3.1 The Abstract Stack

$$
\begin{array}{lcl}
\mathcal{P} & \triangleq & \texttt{Public} \mid \texttt{Private} \\
\mathcal{I} & \triangleq & \{\texttt{bsize} : \mathbb{Z}; \texttt{bperm} : \mathbb{Z} \to \mathcal{P}\} \\
\mathcal{B} & \triangleq & \mathit{block} \times \mathcal{I} \\
\mathcal{F} & \triangleq & \{\texttt{fblocks} : \overrightarrow{\mathcal{B}}; \texttt{fsize} : \mathbb{Z}\} \\
\mathcal{T} & \triangleq & \overrightarrow{\mathcal{F}} \\
\mathcal{S} & \triangleq & \overrightarrow{\mathcal{T}}
\end{array}
$$

Notations

$$
\begin{array}{lcl}
\mathbb{P}(bi, o) & \triangleq & bi.\texttt{bperm}\ o \\
|bi| & \triangleq & bi.\texttt{bsize} \\
(b, bi) \in s & \triangleq & \exists\ f\ t,\ (b, bi) \in \texttt{fblocks}(f) \wedge f \in t \wedge t \in s \\
b \in s & \triangleq & \exists\ bi,\ (b, bi) \in s
\end{array}
$$

Fig. 4. Definition of abstract frames



(a) Clight (b) Cminor (c) Mach

Fig. 5. The abstract stack across compilation

We model the stack as a collection of *abstract frames* which are defined in Fig. 4. An abstract frame $f : \mathcal{F}$ is the abstract counterpart of an activation record for a given function. It is formally encoded as a record with one field `fblocks` recording a list of *abstract blocks* which abstract over the memory blocks constituting the activation record, and one field `fsize` recording the size that this frame will occupy in the *concrete* stack. Note that we use a list of abstract blocks instead of a single abstract block because for some languages of CompCert such as C and Clight the activation record is a list of memory blocks (as shown in Fig. 1). Note also that, for computation of actual stack consumption, `fsize` records the size of the concrete stack frame allocated at the level of machine code. It has no relation to and should not be confused with the sizes of memory blocks (`bsize`). An abstract block $b : \mathcal{B}$ consists of a block identifier together with some *block information*. A block information $bi : \mathcal{I}$ is a record with a field `bsize` recording the size of the corresponding stack block and a field `bperm` which associates a stack permission $p : \mathcal{P}$ with each offset of this stack block which is either `Public` or `Private`.

Based on this definition of abstract frames, a first natural attempt to define abstract stacks $\mathcal{S}$ is as a list of abstract frames, *i.e.* $\mathcal{S} \triangleq \overrightarrow{\mathcal{F}}$. However, this definition fails to properly account for the stack consumption of tailcalls. As described in Sec. 2.4, a tailcall first deletes the frame of its caller then allocates the frame of its callee. As a consequence, the maximum stack space consumed by a sequence of tailcalls is the maximum size of the frames of the called functions, instead of the sum of the sizes of these frames in the case of regular calls. To account for this effect in later developments, we organize an abstract stack $s : \mathcal{S}$ into a list of stages, where a stage $t : \mathcal{T}$ is a list of abstract frames allocated by a sequence of tailcalls, whose head is the active frame for the most recent tailcall in the sequence and whose tail contains the frames historically allocated by the remaining tailcalls and now deallocated. This is formally defined in Fig. 4.

Fig. 5 shows the evolution of an abstract stack at different points during the compilation. Note that in this figure and in the following discussions we will depict the stack as growing "downwards" to be consistent with the convention at the machine code level that the stack grows from a higher address to lower ones. In this example, we consider an assembly function `f` calling a function `g`, compiled from Clight through Cminor to Mach. The assembly function `f` is not compiled, therefore its stack frame stays the same in all cases: its stack block fb has distinguished private (gray) and public (white) regions. On the other hand, the stack of function `g` evolves as described in Fig. 1. In Clight and Cminor (Figures 5a and 5b), the abstract frame for `g` only contains all-public block information. However, starting from Mach (Fig. 5c), the abstract frame for `g` gets a non-trivial block information, which marks some regions as private (gray). The `bperm` field of this block information may be defined as the following function where $[\delta, \delta + |sb|)$ is the range of sf that sb injects into:

$$\texttt{bperm}\ o \triangleq\ \texttt{if}\ \delta \le o < \delta + |sb|\ \texttt{then Public else Private}.$$

3.2 Stack Consumption and Its Upper Bound

We define the size of abstract frames, stages and stacks. The size of an abstract frame $f : \mathcal{F}$ is defined as $\texttt{size\_frame}(f) \triangleq f.\texttt{fsize}$.

The size of a stage $t : \mathcal{T}$ should denote the amount of stack space at the level of machine code that is needed for the execution of the entire sequence of tailcalls that allocate frames in t. It is therefore defined as the maximum size of all the frames $f \in t$:

$$\texttt{size\_stage}(t) \triangleq \max_{f \in t}(\texttt{size\_frame}(f))$$

Finally, the size of a stack $s \in \mathcal{S}$ is defined as the sum of the sizes of its stages, *i.e.*

$$\texttt{size\_stack}(s) \triangleq \sum_{t \in s} \texttt{size\_stage}(t)$$

We write $|f|$, $|t|$ and $|s|$ for the size of a frame f, of a stage t and of a stack s, respectively. To ensure the stack fits into a finite memory, we parameterize our abstract stack with the maximum size of the concrete stack `MAX_STACK`. Later in this paper (Sec. 4.1), we shall enforce that every memory operation maintains the invariant that the size of the stack is below this threshold.

3.3 Stack Access Policy

We consider the following stack access policy: a memory store at a location l is allowed by our policy either if location l is declared as a public location in the stack, or if l is *owned* by the function currently executing, *i.e.* l belongs to the abstract frame that is at the top of the abstract stack. We formally define in the following the notions of public locations and being at the top of the stack. Then we define our stack access policy as the `visible` predicate.

Definition 3.1 (Public locations). A location (b, o) is public in stack s, written $\texttt{public}(s, b, o)$, if for any bi such that $(b, bi) \in s$, $\mathbb{P}(bi, o) = \texttt{Public}$.

Note that this definition defines as public the locations (b, o) where b is not part of the stack ($b \notin s$), *i.e.*, it is not recorded in any abstract frame of the stack. This includes for instance global variables and heap-allocated memory blocks.

Definition 3.2 (Top of the stack). A block b is at the top of the stack s, written $\texttt{is\_stack\_top}(s, b)$, if s can be written as $(t :: s')$ and $(b, \_) \in \texttt{fblocks}(f)$ for some $f \in t$.

Definition 3.3 (Visible locations). A location (b, o) is visible in the stack s, written $\texttt{visible}(s, b, o)$, if b is at the top of the stack s or if (b, o) is a public location. Formally,

$$\texttt{visible}(s, b, o) \triangleq \texttt{is\_stack\_top}(s, b) \vee \texttt{public}(s, b, o).$$

4 STACK-AWARE COMPCERT

We introduce Stack-Aware CompCert in this section. We first enrich CompCert's memory model with an abstract stack and adapt the semantics of the languages of CompCert to introduce stack operations. With these extensions, we reprove all the compiler passes of CompCert by extending the invariants for simulation proofs to include stack injections and invariants of stack consumption, and showing how we maintain these invariants in the compilation passes.

4.1 Stack-Aware Memory Model

We define stack-aware memory states $\mathcal{MS}$ as the extension of an ordinary CompCert memory state m with an abstract stack s. The abstract stack of a memory state m is accessed as ($\texttt{stack}\ m$). We modify the memory operations that update the memory contents (namely `store` and `storebytes`) so that only visible locations may be written to. The new definition of the `store` operation is now as follows, where the $\texttt{range\_visible}\ s\ b\ lo\ hi$ predicate asserts that all locations in the range $[lo, hi)$ are visible in stack s.

```
Definition store' κ m b o v : option mem :=
  if range_visible (stack m) b o (o + |κ|) then store κ m b o v else None.
```

We also enforce the stack access policy on external calls, asserting that the locations marked as private are not modified by external calls.

We provide the following stack manipulation primitives:

push_stage: $\mathcal{MS} \to \mathcal{MS}$; record : $\mathcal{MS} \to \mathcal{F} \to \lfloor \mathcal{MS} \rfloor$; pop_stage : $\mathcal{MS} \to \lfloor \mathcal{MS} \rfloor$

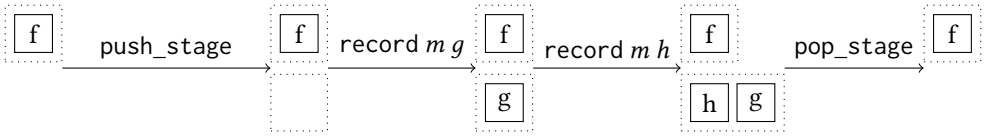


Fig. 6. Effects of the stack manipulation primitives

Fig. 6 illustrates the effect of those primitive stack operations. The push_stage method adds a new empty stage on top of the current stack. The record method prepends a frame onto the topmost stage of the stack. It fails if there is no topmost stage, *i.e.* if the stack is empty, or if the size of the stack after recording that frame would be greater than the predetermined bound on the stack size MAX_STACK. As record is the only operation that can make the stack size increase, it is therefore an invariant of the stack that its size is below MAX_STACK. The pop_stage method pops the top stage (if any) from the stack.

4.2 Stack-Aware Semantics

The semantics of all CompCert languages are expressed as state transition systems, where states follow a similar shape for all languages (except assembly): they are the sum of *call states* which model the state of the program immediately before calling a function; *return states* which model a state where a function has returned, just before restoring the execution of its caller; and *regular states* which model any state not related to function calls.

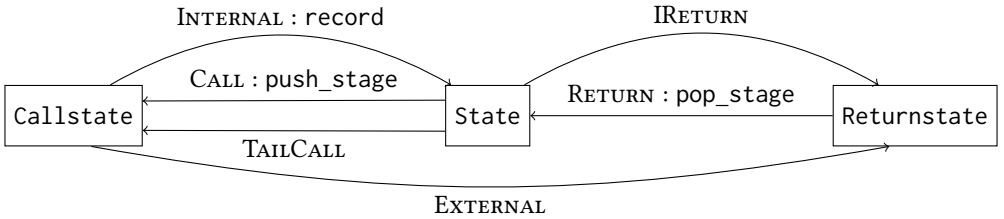


Fig. 7. State transitions with stack primitives

We focus on transitions that have an effect on the stack. Fig. 7 summarizes these transitions, which have the same shape in every language. Transitions are represented with arrows, labeled with the name of the transition and the stack primitive used in this rule, if any.

There are two possible steps from a regular state into a call state: either by performing a regular call, in which case a new stage is pushed on top of the stack; or by performing a tail call, in which case the stack is not modified. There is only one possible step from a regular state to a return state, *i.e.* when a return instruction is found or the end of the function has been reached, and the stack is not modified.

From a call state, two steps are possible. If the function being called is an external function, we immediately step to a return state. If the function being called is an internal function, then we

allocate its stack blocks (using the memory `alloc` operation), and we record an abstract frame (using the `record` operation) on the top of the stack, and we step into a regular state. The abstract frame we record is computed from the stack blocks we just allocated. The `fsize` field of the abstract frame represents the concrete size that this frame will use once the function is compiled to Mach and assembly. This size cannot be computed solely based on the body of the considered function; indeed depending on which optimizations it undergoes, this size may vary arbitrarily. Rather, by following the idea in Carbonneaux et al. [2014], we parameterize the semantics of all languages from C to Mach (excluded) with an oracle `stackspace` that gives, for every function identifier, the amount of concrete stack space that will be needed. This is the size we use for the abstract frame argument of `record` operation. In Mach and assembly, we do not need this oracle anymore because the layout of the stack frame is completely fixed and the size to consider is exactly the size of the stack block that is allocated at function entry. This oracle is instantiated as a byproduct of the compiler: for a given C program, the compiler produces both an output assembly program and a mapping `stackspace` from function identifiers to their stack consumption.

From a return state, the only possible step is to a regular state, after the execution of the `pop_stage` primitive to remove the stack frame of the function being returned from. This step does not depend on whether the function being returned from is internal or external.

4.3 The General Structure of the Proofs

Given the intermediate languages with stack-aware semantics, we reprove all the compiler passes of CompCert following the conventional approach. To prove a forward or backward simulation, we maintain an invariant between the states in the source and target programs, and show that the matching execution steps in the source and target program preserve this invariant. This invariant is usually parameterized by a memory injection. Because we augment the memory with an abstract stack, we need to extend the memory injection to take into account injections between the abstract stacks. Besides memory and stack injections, we also need to maintain another invariant to prove preservation of stack consumption. The following sections discuss the design of stack injections and stack consumption invariants and show how they are used to reprove the compiler passes of CompCert, especially the optimization passes such as inlining and tailcall recognition.

4.4 Frame and Stack Injections

In order to prove the semantic preservation of each compiler pass, CompCert's memory model comes with a set of properties about the interactions of memory operations with memory transformations such as memory injections. For example, the following property asserts that the `store` operation is preserved by injection:

```
Lemma store_inject:
  ∀ j κ m1 b1 ofs v1 n1 m2 b2 δ v2, m1 ↪_j m2 → v1 ↪_j v2 → j b1 = ⌊b2, δ⌋ →
  store κ m1 b1 ofs v1 = ⌊n1⌋ → ∃ n2, store κ m2 b2 (ofs + δ) v2 = ⌊n2⌋ ∧ n1 ↪_j n2.
```

To prove the same property in our enhanced memory model with abstract stack, we need to ensure that whenever the `store` in the source memory m_1 succeeds, it also succeeds in the target memory m_2 at the corresponding location. In particular, we need to ensure that the stack access policy is preserved under injection. To that end, we define an injection relation over abstract frames, then we lift it to stages and stacks.

4.4.1 Stack Permission Refinement. A stack permission p_1 is refined by a stack permission p_2, written $p_1 \sqsubseteq p_2$, when p_2 gives at least as much stack access as p_1. The refinement relation is

determined by the two following inference rules:

$$\frac{}{\texttt{Private} \sqsubseteq \texttt{Public}} \qquad \frac{}{p \sqsubseteq p}$$

4.4.2 Frame Injection. We define $f_1 \hookrightarrow_j f_2$, the injection from the abstract frame f_1 to f_2 by the injection function j. The injection holds if for every abstract block (b_1, bi_1) in f_1 such that b_1 is injected by j into b_2 at offset δ ($j\ b_1 = \lfloor b_2, \delta \rfloor$), there exists some block information bi_2 such that:

- (b_2, bi_2) is in f_2 and;
- for every offset o lower than $|bi_1|$ the stack permission of o in bi_1 is refined by the stack permission of $o + \delta$ in bi_2: $\forall\ o,\ 0 \le o < |bi_1| \rightarrow \mathbb{P}(bi_1, o) \sqsubseteq \mathbb{P}(bi_2, o + \delta)$.

Going back to Fig. 1, we can check that the different frames satisfy this frame injection relation. For the first two transformations, the block permissions are always public, making the frame injection trivial. For the injection from Cminor to Mach, we need to check that the stack permissions are refined. In this example, this amounts to showing that:

$$\forall\ o, 0 \le o < |sb| \rightarrow \texttt{Public} \sqsubseteq (\ \texttt{if}\ \delta \le o + \delta < \delta + |sb|\ \texttt{then Public else Private})$$

which in turns reduces to $\texttt{Public} \sqsubseteq \texttt{Public}$, which holds trivially.

4.4.3 Stage Injection. Now we define the injection of stages t_1 and t_2, parameterized by a memory injection j and a memory state m_1, written $t_1 \hookrightarrow_j t_2$. For readability, the memory is omitted from the notation. If a stage t_1 has no permission in m_1 ($\texttt{noperm}\ m_1\ t_1$), *i.e.* all blocks in all frames of that stage have empty permissions in m_1, then it injects into any stage t_2. If f_1 is the head of the source stage, then there must exist a frame f_2 which is the head of the target stage such that $f_1 \hookrightarrow_j f_2$. Note that we do not require anything about source frames in t_1 in the second rule, because those frames correspond to blocks de-allocated by historical tailcalls and have no permissions.

$$\frac{\texttt{noperm}\ m_1\ t_1}{t_1 \hookrightarrow_j t_2} \qquad \frac{f_1 \hookrightarrow_j f_2}{f_1 :: t_1 \hookrightarrow_j f_2 :: t_2}$$

The intuition behind this definition of stage injection, in particular the first rule that enables to inject a stage with no permissions into any stage, comes mostly from the function inlining and tail call recognition optimizations, where such cases occur, as we will see in Sec. 4.6.

4.4.4 Stack Injection. For most of CompCert's passes, the stack injection that we will consider is a simple pairwise relation, as shown in the following rules.

$$\frac{}{[] \hookrightarrow_j []} \qquad \frac{t_1 \hookrightarrow_j t_2 \qquad s_1 \hookrightarrow_j s_2}{t_1 :: s_1 \hookrightarrow_j t_2 :: s_2}$$

That is mostly satisfactory because most passes preserve the call-return structure of programs. However, for inlining and tailcall recognition, we need to generalize this notion to capture these cases described in Fig. 3c and Fig. 3d. We parameterize the relation by a *stack injection descriptor* g, which records the number of source stages that inject into each target stage, as a list of natural numbers. For example, the stack injection descriptor for the situation depicted in Fig. 3b is $[1; 1]$, whereas the stack injection descriptor for Fig. 3c and Fig. 3d is $[2]$.

The stack injection between stacks s_1 and s_2 parameterized by the memory injection j and the stack injection descriptor g, written $s_1 \hookrightarrow_j^g s_2$, is defined inductively as follows (where $\|l\|$ gives the length of the list l):

$$\frac{}{[] \hookrightarrow_j^{[]} []} \qquad \frac{\|\overrightarrow{t_1}\| = n \qquad 0 < n \qquad \forall\ t_1 \in \overrightarrow{t_1},\ t_1 \hookrightarrow_j t_2 \qquad s_1 \hookrightarrow_j^g s_2}{\overrightarrow{t_1} +\!\!+ s_1 \hookrightarrow_j^{n::g} t_2 :: s_2}$$

The inductive rule asserts that for every target stage t_2, there is a non-empty list of source stages $\overrightarrow{t_1}$, all of which inject into t_2.

4.4.5 Preservation of Memory Operations. We can now reprove the preservation of `store` operations under injections, *i.e.* the `store_inject` theorem. This follows immediately from the preservation of the stack access policy under injection, shown in the following lemma.

Lemma 4.1 (Preservation of the stack access policy). *Given two stacks s_1 and s_2 in injection by the memory injection j and the stack injection descriptor g, for any blocks b_1 and b_2 and offset δ such that $j(b_1) = \lfloor b_2, \delta \rfloor$, for every location (b_1, o) visible in s_1, location $(b_2, o + \delta)$ is visible in s_2.*

Proof. The proof of this lemma follows immediately from the preservation of the "stack-top" property and the preservation of public locations, which are omitted here. □

As a result, we recover the preservation of `store` operations by memory injections, as shown in the lemma `store_inject` presented at the beginning of this section.

4.5 Subtle Compilation Scenarios for Optimization Passes

For most passes of CompCert, the stack operations in the source exactly match the same stack operations in the target and the stack injection and the invariant of stack consumption are therefore trivially maintained. However, in the tailcall recognition and function inlining optimizations, maintaining these invariants is a non-trivial task. In order to better understand this point, we show in Fig. 8 four subtle scenarios that happen in the tailcall recognition and function inlining optimizations. For each of these scenarios, we show a simple snippet of code before and after the transformation, together with the sequence of stack operations that each program executes.

The first scenario (Fig. 8a) corresponds to the case of an inlined function call: a function call to `g` is replaced by the code of `g`. There is no more function call in the target to match the function call and the associated stack operations in the source. The second scenario (Fig. 8b) corresponds to the case of a tail call inside the body of an inlined function which is transformed into a regular call: once `g` is inlined into `f`, the call to `g` is no longer in tailcall position, and needs to be turned into a regular call, hence introducing a `push_stage` operation in the target. The third scenario (Fig. 8c) corresponds to the inlining of a tail call: in that case a `record` operation in the source has no matching equivalent in the target. Finally, the fourth scenario (Fig. 8d) corresponds to the tail call recognition pass where a regular function call is transformed into a tail call: the `push_stage` and `pop_stage` operations corresponding to function `g` in the source are not matched in the target. However the `pop_stage` for `f` does match in both programs. Each of these scenarios consists of a sequence of stack injections, together with the name of the stack operations performed in the source and target programs. Frames with a gray background have no permissions, as required by tailcall recognition optimization as discussed in Sec. 2.4.

4.6 Preservation of Stack Injection under Stack Operations

We now show how the stack injection is preserved throughout the executions of related programs. The lemmas that show the preservation of stack injections across stack operations are shown in Fig. 9. Most compiler passes maintain the same stack structure at every point in the execution of the source and target programs. For these passes, the stack injection descriptor is a list of 1s, and every stack operation (`push_stage`, `record`, `pop_stage`) in the source program has a counterpart in the target program. We use PushStageInject at the call site, RecordInject at function entry and PopStageInject at function exit to maintain the stack injection.

We also need versions of these lemmas where the stack operations happen only in the source program. Rule PushStageInjectLeft allows a `push_stage` operation to be matched with no operation

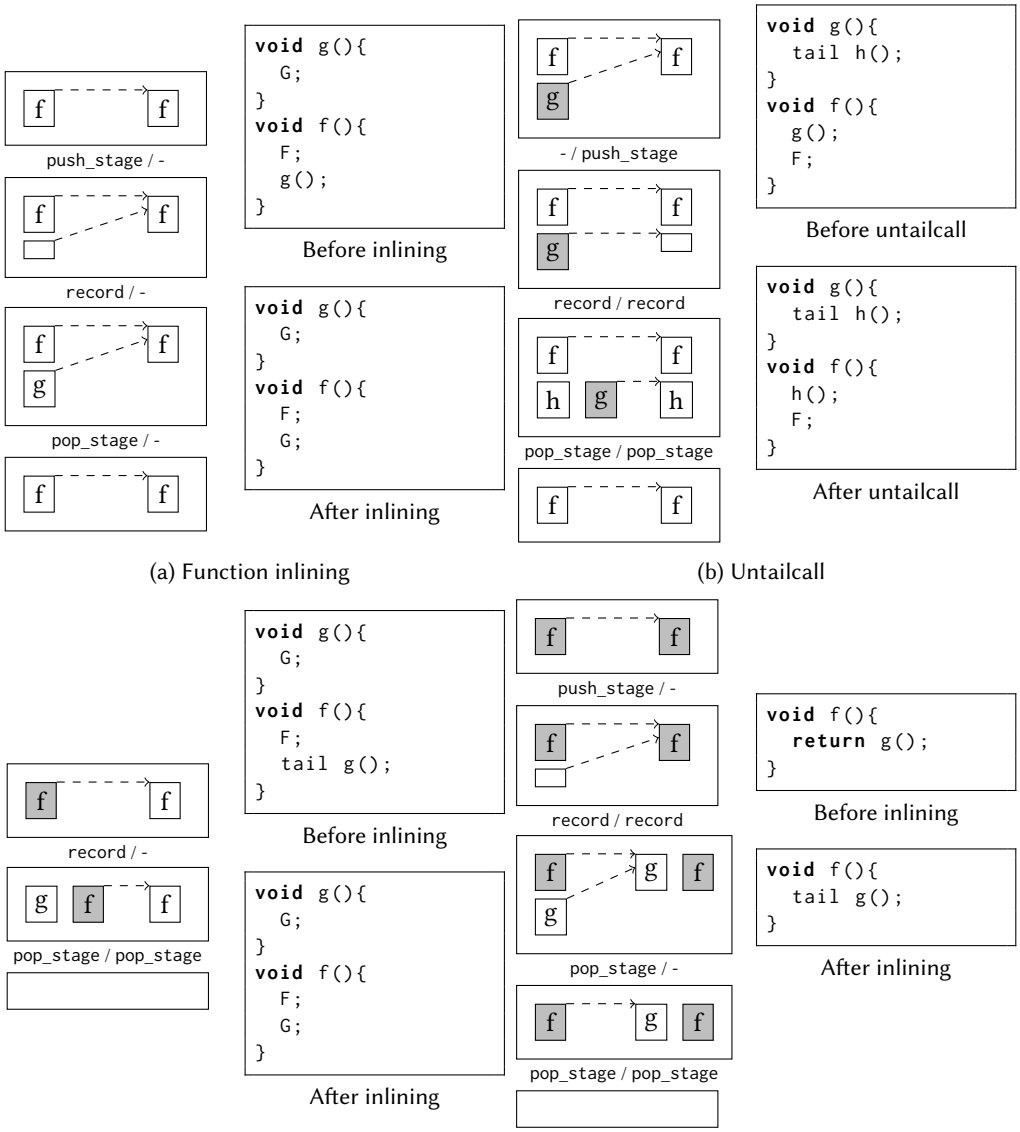


Fig. 8. Evolution of the stack injection in different scenarios

in the target, and increases the number of source stages that inject into the top target stage. It is used in the scenarios in Fig. 8a and Fig. 8d. Rule RecordInjectLeft allows to record a frame only in the source stack, provided that the frame to be recorded injects into the topmost target stage. It is used in the scenarios in Fig. 8a and Fig. 8c. Rule PopStageInjectLeft allows to pop a stage only from the source, provided that at least one stage remains in the source after the `pop_stage` operation. It is used in the scenarios Fig. 8a and Fig. 8d.

Finally, the "Untailcall" scenario in Fig. 8b requires to match no operation in the source with a `push_stage` operation in the target, because a tail call in the source (not generating any `push_stage` operation) is transformed into a regular call in the target. Rule PushStageInjectRight achieves

$$\textsc{PushStageInject}\ \frac{m_1 \hookrightarrow_j^g m_2}{\texttt{push\_stage}\ m_1 \hookrightarrow_j^{1::g} \texttt{push\_stage}\ m_2} \qquad \textsc{PushStageInjectRight}\ \frac{\texttt{noperm}\ m_1\ (\texttt{hd}(\texttt{stack}\ m_1)) \qquad m_1 \hookrightarrow_j^{(n+2)::g} m_2}{m_1 \hookrightarrow_j^{1::(n+1)::g} \texttt{push\_stage}\ m_2}$$

$$\textsc{RecordInject}\ \frac{m_1 \hookrightarrow_j^g m_2 \qquad f_1 \hookrightarrow_j f_2}{\texttt{record}\ m_1\ f_1 \hookrightarrow_j^g \texttt{record}\ m_2\ f_2} \qquad \textsc{RecordInjectLeft}\ \frac{m_1 \hookrightarrow_j^g m_2 \qquad [f_1] \hookrightarrow_j \texttt{hd}(\texttt{stack}\ m_2)}{\texttt{record}\ m_1\ f_1 \hookrightarrow_j^g m_2}$$

$$\textsc{PopStageInject}\ \frac{m_1 \hookrightarrow_j^{1::g} m_2}{\texttt{pop\_stage}\ m_1 \hookrightarrow_j^g \texttt{pop\_stage}\ m_2} \qquad \textsc{PopStageInjectLeft}\ \frac{m_1 \hookrightarrow_j^{(n+2)::g} m_2}{\texttt{pop\_stage}\ m_1 \hookrightarrow_j^{(n+1)::g} m_2} \qquad \textsc{PushStageInjectLeft}\ \frac{m_1 \hookrightarrow_j^{n::g} m_2}{\texttt{push\_stage}\ m_1 \hookrightarrow_j^{(n+1)::g} m_2}$$

Fig. 9. Preservation of stack operations

this by requiring: *1)* that at least two source stages inject into the topmost target stage; *2)* that the topmost source stage has no permissions (in the memory model). We can split the topmost group of stages into two groups: one with only the topmost stage, that will inject into the newly pushed target stage thanks to the absence of permissions in that stage; the rest of the stages inject into the second-to-top target stage, just like before.

4.7 Stack Consumption Invariant

Most passes of CompCert preserve the stack structure. For those passes, the stack consumption invariant is represented as the proposition $s_1 \geq_g s_2$ inductively defined by the following rules, where s_1 and s_2 are respectively the source and target abstract stacks and g is a stack injection descriptor.

$$\frac{|t_1| = |t_2| \qquad s_1 \geq_g s_2}{(t_1 :: s_1) \geq_{(1::g)} (t_2 :: s_2)} \qquad \frac{}{[] \geq_{[]} []}$$

The tailcall recognition and inlining optimization passes change the structure of the stack. For tailcall, we define the following rules to capture the invariant of stack consumption.

$$\frac{\|\overrightarrow{t_1}\| = n \qquad n > 0 \qquad \exists t_1 \in \overrightarrow{t_1}, |t_1| = |t_2| \qquad s_1 \geq_g^{\texttt{tc}} s_2}{(\overrightarrow{t_1} +\!\!+ s_1) \geq_{(n::g)}^{\texttt{tc}} (t_2 :: s_2)} \qquad \frac{}{[] \geq_{[]}^{\texttt{tc}} []}$$

The inductive case of $\geq^{\texttt{tc}}$ captures the fact that tailcall optimization may collapse a list of stages $\overrightarrow{t_1}$ allocated by a sequence of regular calls into a single stage t_2 allocated by a sequence of tailcalls. It requires that the size of some stage t_1 in $\overrightarrow{t_1}$ (more specifically, t_1 is the stage with the maximum size in $\overrightarrow{t_1}$) is equal to the size of the stage t_2. To see how this invariant is used to prove preservation of stack consumption for tailcall optimizations, consider the situation after the record operation in Fig. 8d as an example. At that point, $\overrightarrow{t_1} = [[g], [f]]$ and the two stages $[f]$ and $[g]$ inject into the single stage $t_2 = [g, f]$. Assume the frame size of f ($|f|$) is greater than that of g ($|g|$), then we pick $t_1 = [f]$ and get $|t_1| = |t_2| = |f|$.

For inlining, we define the following rules:

$$\frac{\|\overrightarrow{t_1}\| = n \qquad n > 0 \qquad \overrightarrow{t_1} = t_1' +\!\!+ [t] \qquad |t| = |t_2| \qquad s_1 \geq_g^{\texttt{il}} s_2}{(\overrightarrow{t_1} +\!\!+ s_1) \geq_{(n::g)}^{\texttt{il}} (t_2 :: s_2)} \qquad \frac{}{[] \geq_{[]}^{\texttt{il}} []}$$

Again, the inductive case of $\geq^{\mathrm{i1}}$ captures the fact that inlining may inject a list of stages $\overrightarrow{t_1}$ into a single stage t_2. It requires that the last stage in $\overrightarrow{t_1}$ which is allocated at the starting point of inlining must have a size equal to the size of t_2. To see how this invariant is used to prove preservation of stack consumption for inlining, consider the situation after the record operation in Fig. 8a as an example. At this point, $\overrightarrow{t_1} = [[g], [f]]$. Both $[g]$ and $[f]$ inject into $t_2 = [f]$. Note that the starting point of inlining in the source is f. Thus we have $t = [f] = t_2$ and $|t| = |t_2|$.

It is important to emphasize that the size of the frame for each function is the size of the concrete stack frame at machine code level (given by an oracle whose purpose and instantiation have been described in Sec. 4.2). They are therefore the same throughout the compilation passes. For the example of function inlining, it might seem counter-intuitive at first sight that the stack consumption can be preserved, when one knows that there might be more local variables in the function body after inlining. However, this is irrelevant because the frame sizes we consider in every language are fixed to the sizes of the concrete frames, including the languages before the inlining phase.

The three propositions above imply that the size of the target stack is smaller than or equal to the size of the source stack. In the regular case, where each target stage has exactly one corresponding source stage, this is trivial as the stacks have the same size. In the two other cases (tailcall and inlining), for each target stage, there is one source stage that is larger or equal, hence each group of source stages is larger than or equal to the corresponding target stage, hence the size of the source stack is larger than or equal to the size of the target stack.

4.8 Preservation of Stack Consumption under Stack Operations

For 2 out of the 4 scenarios in Fig. 8 (Fig. 8a and Fig. 8b), the size invariant could be preserved even if we did not store the *tailcalled* inactive frames in our abstract stack. In these cases indeed, all the frames present in the target stack are also in the source stack, which may contain even more. Therefore, the size of the source stack is necessarily larger than or equal to the size of the target stack.

For tailcall inlining (Fig. 8c), we actually need to maintain the inactive frame for `f` in the source so that we can establish that the size of the source stack after the `record` operation ($max(\mathtt{g}, \mathtt{f})$) is not smaller than the size of the target stack (`f`). For tailcall recognition (Fig. 8d), we cannot preserve the size invariant (this would imply $\mathtt{f} \geq max(\mathtt{g}, \mathtt{f})$) after the `pop_stage` operation on the source. However, we know that at this point in the source program, the next instruction is going to be a `return` instruction, resulting in a `pop_stage` which will be matched in the target program with another `pop_stage`, after which the size invariant will be recovered. Note that we can have these intermediate steps where the size invariant temporarily does not hold because we only use this invariant to prove that `record` operations in the target succeed.

4.9 The Final Theorem of Stack-Aware CompCert

Using the devices presented above, we reprove the correctness of the CompCert compiler. We obtain the same theorem as CompCert, *i.e.* a backward simulation between the C program and its compiled assembly version. The main difference with CompCert's theorem is that the compiler produces an oracle `stackspace` that is fed into the semantics of C.

Theorem 4.2 (Stack-Aware CompCert's semantic preservation). *For every C program p without undefined behavior, if CompCert generates an assembly program tp and an oracle* `stackspace`*, then every behavior of tp in the assembly semantics is an improvement over some behavior of p in the C semantics parameterized with oracle* `stackspace`*.*

All in all, the changes we made to CompCert to obtain Stack-Aware CompCert amount to about 21 thousand lines of Coq code. This includes our formalism of stack abstraction in the memory model (implementation and proofs related to the stack operations), modifications to the semantics of each intermediate language to include stack operations at function call and return, plus modifications to each compiler pass' correctness theorem, where proofs about stack permissions and stack consumption had to be added.

5 COMPCERTMC: GENERATION OF MACHINE CODE IN COMPCERT

In this section, we describe CompCertMC, an extension of Stack-Aware CompCert that compiles towards machine code. By exploiting our abstract stack, we are able to merge the list of stack frames into a single finite stack, and to eventually generate low-level code close to machine code. We also describe the implementation of verified separate compilation by generalizing the syntactic linking to the new languages introduced by CompCertMC.

5.1 Overview of CompCertMC

The structure of CompCertMC is depicted in Fig. 10. CompCertMC is Stack-Aware CompCert with an extended compilation chain to machine code. Starting from CompCert assembly, programs are compiled into lower-level intermediate code, going through the Single-Stack assembly language (`SSAsm`), the *real* assembly language (`RealAsm`), the *flat* assembly language (`FlatAsm`) and machine-code like language (MC). In `SSAsm`, the stack is represented as a single memory block, *i.e.* the list of stack blocks is merged into a single stack block of finite size. In `RealAsm`, the responsibility for dealing with the return address is shifted from the callee (in CompCert) to the caller (as in conventional x86 assembly). Moreover, pseudo-instructions `Pallocframe` and `Pfreeframe` are compiled into actual assembly instructions. The compiler pass responsible for this is called `PseudoInstr`. In `FlatAsm`, the functions and global variables are merged into disjoint flat memory spaces that we call *segments* (similar to sections in ELF files). The next phase (Id Resolution) resolves the references to code labels to concrete addresses, resulting in programs written in the language MC. The MC programs closely mirror the actual machine code (like the ELF object files). We then use an instruction encoder provided by formal machine models (*e.g.*, the RockSalt x86 [Morrisett et al. 2012; Tan and Morrisett 2018]) to generate the actual machine code (*e.g.*, ELF files). The last step is not verified (as indicated by the dashed line in Fig. 10) as it would require a formal model of ELF formats; we left it for future work.

This sequence of additional languages depends on the target architecture and would need to be adapted for other architectures. For this paper, we only target x86 as a proof of concept. We believe our approach is generally applicable to other architectures like ARM, PowerPC or RISC-V and do not foresee any immediate problem in porting these addition compiler passes to them.

Like the original CompCert, CompCertMC supports verified separate compilation. We have generalized the notion of syntactic linking to the new intermediate languages and proved the

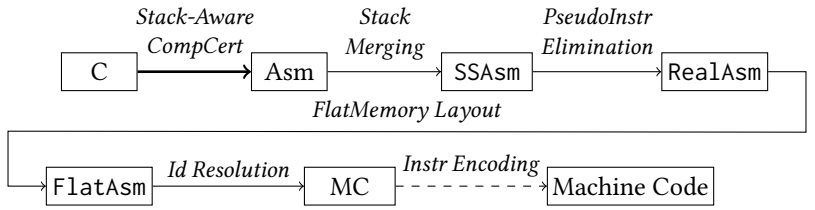


Fig. 10. CompCertMC

```
Pallocframe fi ora olink :=
 sp ← alloc 0 (fsize fi);
 [sp + olink] ← RSP;
 [sp + ora] ← RA;

 RAX ← RSP;
 RSP ← sp.

Pfreeframe sz ora olink :=
 ra ← [RSP + ora];
 Vptr b o ← RSP;
 RSP ← [RSP + olink];
 RA ← ra;
 free b 0 sz.
```

(a) CompCert Asm

```
Pallocframe fi ora :=
 sp ← alloc 0 (fsize fi);

 [sp + ora] ← RA;
 record (sp,fi);
 RAX ← RSP;
 RSP ← sp.

Pfreeframe sz ora :=
 ra ← [RSP + ora];
 Vptr b o ← RSP;
 RSP ← parent_sp;
 RA ← ra;
 free b 0 sz;
 pop_stage.
```

(b) Stack-Aware CompCert Asm

```
Pallocframe fi o :=
 sp ← RSP - fsize fi;

 [sp + ora] ← RA;

 RAX ← RSP;
 RSP ← sp.

Pfreeframe sz ora :=
 ra ← [RSP + ora];

 RSP ← RSP + sz;
 RA ← ra.
```

(c) Single-Stack Asm

Fig. 11. Semantics of `Pallocframe` and `Pfreeframe`

commutative property between the new transformations and syntactic linking. Verified separate compilation easily follows from these generalizations.

In the rest of this section, we discuss the implementation and verification of the first three passes of the extension to Stack-Aware CompCert which contain most of the technical novelties presented above. We elide a discussion of the final correctness theorem of CompCertMC, which is similar to that of Stack-Aware CompCert.

5.2 Compilation to the Single-Stack Assembly Language

The Stack Merging phase is not a program transformation but a semantic reinterpretation: the source and target programs are the same; the semantics of `SSAsm` differs from the semantics of CompCert assembly for the instructions that manipulate the stack, so that they operate over a single stack block rather than separate blocks for each function. Our goal is to show that the `SSAsm` semantics refine the Stack-Aware CompCert assembly semantics for any program. In the rest of this section, we explain the differences between the semantics of the original CompCert assembly, Stack-Aware CompCert assembly and `SSAsm` and discuss the key ideas of the refinement proof.

CompCert's assembly semantics. CompCert assembly programs are lists of instructions that operate over a set of registers and a memory state. Instructions include common assembly instructions together with pseudo-instructions `Pallocframe` and `Pfreeframe` that are called at function entry and exit, respectively. The semantics of the `Pallocframe` instruction, shown in Fig. 11a, is to allocate a new block for the stack frame of the current function, save the stack pointer and the return address in this new block, and update `RSP`. The semantics of `Pfreeframe` is symmetric: it recovers the return address and parent's stack pointer, and frees the stack block. The memory state of assembly programs is still a collection of blocks, rather than a *flat* memory space. Because of this abstract view of the memory, it is needed to store a pointer to the frame of the caller (at offset `olink` in the function's frame), in order to access the function arguments for example, whereas in traditional assembly this is achieved through pointer arithmetic. Note also that it is the responsibility of the called function to write the return address (stored by the `call` instruction in the pseudo-register `RA`) at offset `ora` in the stack frame.

Stack-Aware CompCert's assembly semantics. With our stack-aware memory model, we do not need to store a link to the caller's frame anymore: this pointer can be retrieved in the abstract stack,

through the parent_sp operation, which returns the block associated to the second-to-top stage in the abstract stack. We replace CompCert's assembly semantics with a new assembly semantics that now includes stack operations (record and pop_stage). The semantics of the pseudo-instructions Pallocframe and Pfreeframe in this semantics is given in Fig. 11b.

Single-Stack assembly semantics. We give in Fig. 11c an alternative semantics of CompCert assembly, called Single-Stack assembly (SSAsm), where the Pallocframe pseudo-instruction no longer allocates a block but simply subtracts an offset from the stack pointer register RSP. Similarly, the parent_sp operation in the semantics of Pfreeframe is transformed into a simple addition to RSP. At this point, we can also get rid of the abstract-stack related operations, which do not have a counterpart in actual assembly code.

Correctness of the single-stack semantics. The correctness of this semantics reinterpretation of assembly programs in the single-stack semantics is stated as a forward simulation. That is, every behavior of an assembly program p in Stack-Aware CompCert assembly is also a behavior of p in SSAsm. This proof is based on a memory injection, which maps all the stack blocks of Stack-Aware CompCert assembly functions into a single pre-allocated stack block for SSAsm. To show that if any allocation of stack frame in the source program succeeds then so does it in the target program, it makes use of the fact that the source program with defined semantics never overflows the stack, which is ensured by our stack-aware semantics. A key difference with other passes in CompCert that are based on memory injections is that several source locations may inject to the same target locations at different times during the execution of the program.

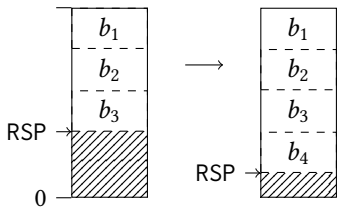


Fig. 12. Single-Stack invariants

We maintain the invariant that the RSP register always points to the top of the abstract stack, such that no source block injects in the target stack block below RSP. This property is fundamental so that we can establish a new injection when a frame is pushed. Fig. 12 illustrates the situation. The hatched regions are those where no source block with permissions injects. Initially, we have three stack blocks b_1, b_2 and b_3 in the stack, and nothing injects below the RSP pointer. Because of that, we know we can record a new block b_4 onto the stack, and moves the RSP pointer so that nothing injects after it.

5.3 Elimination of Pseudo Instructions

After we merge the stack blocks into a unique stack block, there are still some discrepancies in our assembly language. First, the writing of the return address on the stack is performed by the callee, as part of the Pallocframe instruction, whereas in conventional x86 assembly this is a side-effect of the call instruction. Second, we still have these pseudo-instructions (Pallocframe and Pfreeframe), whereas we should only have actual assembly instructions that operate on the stack pointer RSP.

We design a new assembly language, RealAsm, where the semantics of Pallocframe (resp. Pfreeframe) only perform pointer arithmetic on RSP, and the space reserved (resp. freed) excludes space for the return address; and the semantics of call (resp. ret) pushes (resp. pops) the return address on the stack.

We first prove a refinement between any program p in SSAsm and the same program in RealAsm. This is stated as a backward simulation, specifically we show that provided that the source program is safe, then for every target step, there is a corresponding source step. We need to identify the possible sequences of states transitions that involve call, ret, Pallocframe and Pfreeframe. Such

sequences are *1)* `call` followed by a `Pallocframe` instruction (internal function call); *2)* `call` followed by an external function (external function call); *3)* `Pfreeframe` followed by a `ret` instruction (regular function return); *4)* `Pfreeframe` followed by a `jmp` instruction (tailcall). In the last case, the `jmp` is either to an internal function (in which case the next instruction will be a `Pallocframe`) or an external function, but not to an arbitrary instruction. Fig. 13 illustrates the simplest case of internal function calls.

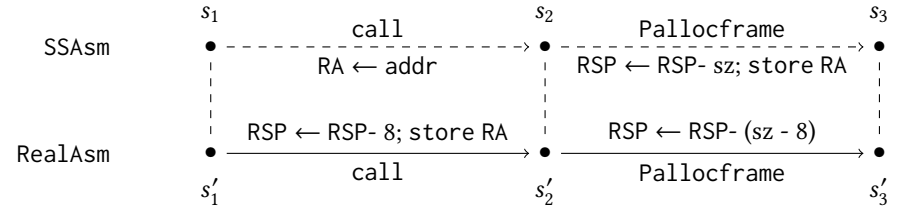


Fig. 13. Backward simulation between internal function calls in `SSAsm` and `RealAsm`

The simulation relation used for this proof has different cases depending on the nature of the next instruction. If the next instruction is not an intermediate instruction (one that appears only in the middle of the sequences of transitions we identified earlier, *i.e.* neither `Pallocframe` nor `ret` nor `jmp`), then the relation is the extensional equality of the states. If the next instruction is `Pallocframe` however, the relation (between s_2 and s'_2) records that the states differ only for the shift of `RSP` and the memory store for the return address in `RealAsm`. After the execution of `Pallocframe`, we recover the extensional equality between states s_3 and s'_3. Similar cases are defined for the other transition sequences described earlier.

Then, we design a simple transformation, `PseudoInstr`, that transforms pseudo-instructions `Pallocframe` and `Pfreeframe` into pointer arithmetic (`sub` and `add`) on `RSP` that reflects exactly their `RealAsm` semantics. The correctness proof of this transformation is stated as a forward simulation, and is quite straightforward. This forward simulation is transformed into a backward simulation using standard tools in CompCert and concatenated with the previous backward simulation to obtain a backward simulation between the `SSAsm` semantics of p and the `RealAsm` semantics of p with pseudo-instructions eliminated.

5.4 Compilation to Flat Memory Spaces

With a source language close to real assembly, the goal of the FlatMemory Layout pass is to layout the data and code into continuous segments. We first introduce its target language `FlatAsm`, then the compilation pass and finally the correctness proof of this pass.

An assembly language with flat memory spaces. The defining feature of `FlatAsm` is that data and functions are stored in continuous data or code segments, instead of separate memory blocks like in the previous assembly languages. The syntax of `FlatAsm` is exactly the same as `RealAsm`. A `FlatAsm` program contains the following components:

- A list of *segments* whose elements are represented by a record data type containing the id and size of the segment. For now, a program only has two segments. One holds code and the other holds data. We use *segment labels* to identify locations in segments, which are pairs of segment identifiers and offsets into the segments.
- A list of global definitions with information of their locations in corresponding segments.
- A mapping from global identifiers to segment labels, denoted by `gmap`. It is used to locate the positions of global definitions in the code or data segment.

- A mapping from pairs of function identifiers and labels to segment labels, denoted by `lmap`. It is used to locate the positions of labels in functions in the code segment.

When a `FlatAsm` program is initialized, for each segment we allocate a memory block of its size. The memory block for the data segment is initialized with the initial values of global data objects. The memory block for the code segment is assigned appropriate permissions. Segment labels are then interpreted as memory addresses or pointers at runtime. The program counter increases using the size information of instructions (provided by an oracle later instantiated by a binary encoder of instructions) in sequential execution and is assigned appropriate addresses in case of function call or branching by consulting `gmap` or `lmap`. The references to data are interpreted into pointers by consulting `gmap` at runtime.

The compilation of `RealAsm` *to* `FlatAsm`. The compilation pass translates a `RealAsm` program into a `FlatAsm` program by calculating `gmap` that maps the internal source functions into the code segment and the global variables into the data segment, `lmap` that maps labels in internal source functions into the code segment, and the sizes of the code and data segments. The calculation is done by scanning the list of global definitions in the source program in sequence and concatenating internal data and function definitions.

Correctness of the translation. The proof follows a conventional pattern for establishing forward simulations in CompCert. The key steps include defining an invariant that asserts `lmap` and `gmap` agree with the memory injection from the source to target program and showing this invariant is maintained by lock-step execution.

Because data and functions are collapsed into continuous segments, we need a new notion of syntactic linking for `FlatAsm` programs for proving separate compilation. To syntactically link modules $M_1, \ldots, M_n$ in `FlatAsm`, besides merging the global definitions as described in Sec. 2.3, we also need to perform the following relocation process:

- Data segments in $M_1, \ldots, M_n$ are merged into a single data segment. A "repositioning" table is created which records the starting offsets of global variables in M_i in the resulting data segment. The same process is applied for merging the code segments.
- The information of locations for global definitions are updated using the repositioning tables for data and code segments.
- The `gmaps` (`lmaps`) in $M_1, \ldots, M_n$ are merged into a single `gmap` (`lmap`) and their values are updated using the repositioning tables.

Then, it is straightforward to show that syntactic linking commutes with the translation from `RealAsm` to `FlatAsm`. From this we derive the correctness of separate compilation.

6 STACK-AWARE COMPCERTX: CONTEXTUAL COMPILATION IN COMPCERT

This section introduces Stack-Aware CompCertX, an improved version of CompCertX [Gu et al. 2015] which is an extension of CompCert used in the context of the CertiKOS project, a formally verified operating system kernel. The main difference between CompCertX and CompCert is that it allows for some degree of compositional compilation that we call *contextual compilation*, because it relates the executions of a source C module and its compiled assembly version in a generalized context, instead of an empty context in the original CompCert compiler.

To generalize CompCert's theorem to the setting of CompCertX, some assumptions must hold on the memory regions of the context that the module under compilation may update. CompCertX enforces these assumptions by disallowing modification to *any* stack data belonging to the context. This results in an incomplete extension to CompCert which does not support compilation of programs with stack-allocated data.

In this section, we remove the restriction on stack-allocated data by basing Stack-Aware CompCertX on top of CompCertMC. We show that the assumptions on the stack can be stated in a straightforward manner by exploiting the stack permissions provided by our abstract stack. In the end, we get Stack-Aware CompCertX that supports contextual compilation and all the features of the original CompCert.

6.1 Overview of CompCertX

CompCertX is a *contextual* compiler, that establishes the correctness of the compilation of a module which may be executed by invoking an arbitrary function f in it from an arbitrary *context*, *i.e.* any starting memory state, and returning a value of the type dictated by `f`'s signature. In particular, the context can be obtained by the execution of a hand-written assembly program (not necessarily the result of compiling a C function with CompCert), calling the function `f`.

The initial context must however satisfy some properties to account for the change in calling conventions between the C version of `f` and its compiled assembly version. Indeed, the former receives its arguments as a list of values, whereas the latter fetches them from registers and stack memory, at locations dictated by `f`'s signature. Hence, for the correctness of the compilation of `f` with arguments *args*, a necessary condition on the initial memory is that it contains the values *args* at the appropriate stack locations.

The final theorem of CompCertX is stated as a simulation argument.

Theorem 6.1. *Let p be a C program, tp be the result of compiling p into CompCert assembly, and f be a function identifier with signature sg called with arguments args. Let* `init_rs` *be a register state and* `init_m` *a memory state such that the arguments args are encoded in* `init_rs` *and* `init_m` *appropriately. Then the execution of f in p starting from initial memory* `init_m` *is refined by the execution of f in tp starting from initial memory* `init_m` *and initial register state* `init_rs`*.*

$$\begin{array}{ccc} \texttt{init\_m} & \xrightarrow{\quad p(f)(args) \quad} & m_C \\ & & \curvearrowright \; \vdots \; j \\ (\texttt{init\_m}, \texttt{init\_rs}) & \dashrightarrow^{tp(f)} & (m_{Asm}, rs) \end{array}$$

In particular, the resulting memory states of both executions, m_c and m_{Asm} are related by a memory injection j, introduced by the compilation passes.

Based on this generalized compiler, Gu et al. [2015] build a library for certified abstraction layers which is used as a foundation for composing large systems out of small independent components. The compositionality achieved is more flexible than what is attainable by separate compilation [Kang et al. 2016] since it is possible to mix programs written in different languages.

6.2 Subtleties in the Proofs of CompCertX

A major issue that has to be dealt with in the correctness proof of CompCertX is that the arguments of the considered function call are duplicated: at the C level, they appear both as a list of values *args* and encoded in the memory `init_m`. As a result, the consistency between those two representations of the arguments has to be maintained.

In CompCert, there are three ways arguments are passed to function calls. In the front-end and the few first languages of the back-end (from C to RTL which is an intermediate language with infinitely many pseudo-registers), arguments as passed as a list of values. Then, in LTL (which represents programs as control flow graphs operating on finitely many physical registers) and Linear (which is like LTL, but with control flow graphs replaced by a linear list of instructions and

explicit branchings), arguments are fetched from a *location set*, which associates *locations* (abstract stack slots and registers) to values. Finally, in Mach and assembly, arguments are fetched directly from the stack (in memory) and registers.

Let us focus in particular on the Stacking compiler pass in CompCertX, from the Linear language to the Mach language. Because of the duplication of arguments, the arguments are both in the memory and in the location set in Linear, while the arguments are only in memory in Mach. For this compiler pass to be proved correct, the consistency of the two copies of the arguments in Linear must be maintained. To that end, modification to the memory region holding duplicated arguments must be forbidden, as it would induce an inconsistency between the memory and the location set, hence break the preservation of semantics by compilation. Fig. 14 illustrates the duplication of the arguments and shows that both the location set and the memory in Linear inject into the memory in Mach, hence the need for the consistency between the Linear memory and location set.

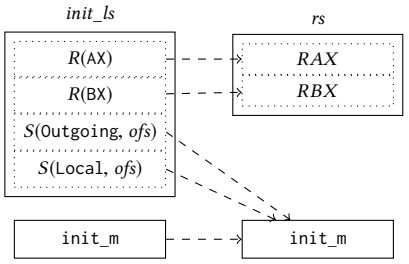


Fig. 14. Injection for the Stacking pass

In the past, there have been several strategies used to overcome this situation (see Gu et al. [2014] for details). For instance, one solution was to tag the blocks to distinguish global blocks from stack blocks, and forbid writing to stack blocks. This effectively enforces the desired consistency invariant on programs. However, this totally prevents writing to stack blocks, including stack-allocated data that we might pass pointers to. In another more involved solution, this consistency was maintained by adding a hypothesis to the final theorem, stating that the semantics of the source program was defined, even when the permissions of the arguments region were removed. This led to complex reasoning to relate two executions of the source program (one with the permissions for the arguments, and the other without) with one execution of the target.

6.3 Enforcing Assumptions for CompCertX Using the Abstract Stack

Using our stack-aware memory model, we prevent programs from modifying the stack of their caller, except in their public locations (*e.g.* stack-allocated data), in a more elegant way. This is true under the assumption that the initial memory `init_m` has its abstract stack set up in such a way that the block that contains the initial caller's stack frame has private permissions for its private regions, in particular the arguments, as per the following definition of `initial_caller_protected`, where `init_sg` is the signature of the function called by the initial caller, from which we compute the offsets that contain the arguments (`arg_locs`).

$$\begin{aligned}&\texttt{initial\_caller\_protected}(m, \texttt{init\_sg}) \triangleq \\ &\quad \exists\, \texttt{init\_sp}\ bi, \quad \texttt{hd}(\texttt{stack}\ m) = ([\texttt{init\_sp}], bi) :: \_ \wedge \\ &\qquad\qquad\qquad\quad \forall\, o \in \texttt{arg\_locs}(\texttt{init\_sg}),\ \mathbb{P}(bi, o) = \texttt{Private}\end{aligned}$$

This solution is more elegant than the previous solutions, and ensures the preservation of more than just the arguments section: the callee-save registers, the spilled locations, the return address can also be preserved across the execution of the C function, if we strengthen the definition of `initial_caller_protected`. This is much stronger than the previous statement and is a milestone towards compositional compilation. Indeed, when composing different modules together, we need to ensure that the execution of the called module does not invalidate the private regions of its caller *e.g.* by overwriting its return address. As a sanity check, the `initial_caller_protected` predicate holds for contexts generated as the result of compiling programs with CompCert.

7 RELATED WORK

Verified compilation is not a new research topic. In the area of verified compilation of functional programs, compilation does not involve complicated manipulations of runtime stack like that of imperative programs (*e.g.* CakeML [Kumar et al. 2014]). As a result, the difficulties discussed in this paper do not show up. We compare our work with related work on verified compilation of imperative programs. Most notable related work is about extending CompCert to support merging of stack blocks, machine code generation or compositional compilation. We perform the comparison both quantitatively based on the development time (person-months) and the lines of codes for each extension, and qualitatively based on the features we support as discussed in the introduction (excluding stack-awareness which is a unique feature of our work), *i.e.*, what language is targeted, the completeness of the implementation, and the degree of compositionality achieved, if any.

Table 1. Comparison with the related projects

	Target	Completeness	Compositionality	Time	LOC
CompCert(3.0.1)	CompCert Asm	complete	separate	-	135k
SACC	CompCert Asm	complete	separate	7.5	+21k
CCMC	MC	complete	separate	2	+19k
SACCX	MC	complete	contextual	1	+8k
QCC	SingleStack Asm	w.o. some opts.	N/A	-	100k
CCC	CompCert Asm	w.o. some opts.	general	10	200k
SCC	CompCert Asm	complete	separate	2	+3k
CCX	CompCert Asm	no s.a. data	contextual	-	+8k
CC-TSO	x86-TSO	w.o. some opts.	concurrency	45	85k
CompCertS	CompCert Asm	w.o. some opts.	N/A	25	220k

Table 1 summarizes the result of the comparisons. The first column lists the names of the extensions of CompCert that we consider. The first one is the original CompCert. The following three are Stack-Aware CompCert (SACC), CompCertMC (CCMC) and Stack-Aware CompCertX (SACCX) described in this paper. The rest are existing projects that we describe below. In column 3, "complete" means the compiler implements the full compilation chain of CompCert and supports all the features in the vanilla CompCert such as stack-allocated data; "w.o. some opts" means some optimization passes of CompCert are not implemented; and "no s.a. data" means the compiler does not support programs with stack-allocated data. The last two columns contains the comparison of development time (in person-months) and lines of Coq code. The number of lines is the result of running `coqwc` on the associated developments and adding the columns "spec" and "proof". A number following the + sign indicates the change in LOC between that extension and the version of CompCert it is based upon. Specifically, SACC is based upon CompCert(v3.0.1); CCMC is based upon SACC; and SACCX further extends CCMC. SCC and CCX are based on CompCert(v2.4) and CompCert(v2.3), respectively.

Quantitative CompCert (QCC). Carbonneaux et al. [2014] design an analysis of stack bounds and use their results to merge stack blocks into a single stack. Their key ideas are the following: *1)* they augment the event traces with call and return events, which will be used to calculate the stack consumption of a given trace; *2)* they compute an oracle as a byproduct of the compiler, which associates every function identifier with its stack usage. They prove that if the stack consumption for all traces at the source level does not exceed the size of the actual stack, the source program can be correctly compiled into an assembly program with a finite stack.

We have borrowed from their work the idea of using an oracle generated as a byproduct of compilation to determine the sizes of concrete frames. The main differences between our work and

theirs are listed as follows. First, instead of using a linear event trace to calculate stack consumption, we divide the linear trace into "stages" each of which is a list of frames allocated by a sequence of tailcalls. With this more structured representation of historical events, we are able to maintain the invariant and prove the preservation of stack consumption for optimization passes that change the call and return events, such as tailcall and inlining optimizations. By contrast, they only prove preservation of stack consumption for compilation passes that do not change the augmented event traces. The extended version of their paper suggests new refinement relations for event traces that would allow them to deal with tailcall recognition and function inlining. However, these relations deviate from the conventional one used in CompCert, and it is not clear how feasible their suggested approach is. As such, these optimizations are still not verified in Quantitative CompCert.

Second, our compiler correctness theorem is applicable in compositional compilation, whereas their analysis of stack consumption requires knowledge of *full* event traces. Therefore, their stack merging is possible only for compilation of full programs.

Third, because they only deal with compiler passes that do not change the call and return events, they are able to get an accurate bound of stack consumption at the source level. However, because the tailcall optimization changes the call-return structure, the stack consumption we compute at the source level may be an over-approximation. Consider the following example:

```
int sum(int n, int acc) {
  if(n <= 0) return acc;
  else return sum(n - 1, n + acc)
}
```

The tailcall optimization may change the recursive call to `sum` into a tailcall. Then, the stack consumption at the source level is linear in `n`, while it is constant in the target program. One way to solve this problem would be to parameterize the semantics of languages before tailcall optimization with an oracle which predicts which function calls will be compiled into tailcalls, and compute a tighter bound for source programs. The implementation of this idea is currently underway.

Last, our correctness theorem states that every behavior of the compiled program is an improvement over a behavior of the source program provided the source program does not have undefined behavior. In order to discharge this "defined semantics" hypothesis, we need to analyze the stack usage of source programs. This is orthogonal to the work in this paper and left for future work.

Compositional CompCert (CCC). The state-of-the-art verified compositional compiler for C programs is *Compositional CompCert* [Stewart 2015; Stewart et al. 2015]. In CCC, source modules may be written in any intermediate language whose semantics can be described in the framework of *interaction semantics*. The compilation of heterogeneous modules is proved individually and the results are combined to prove contextual equivalence between the full source and target programs linked through interaction semantics. In order to formalize the non-interference properties of memory operations across (heterogeneous) modules, Stewart *et al.* designed *structured memory injections*, which generalize memory injections to keep track of the ownership of blocks by modules.

Although CCC has a very general notion of compositionality, several optimizations including tailcall recognition and function inlining have not been ported and the target language is CompCert assembly. It is unclear whether their work can be extended to support all optimizations; or to support lower-level assembly languages. Last, although structured memory injections are a general mechanism for describing invariants over memory states, they also incur a lot of complexity in verification, as manifested by their Coq development.

In contrast, both CompCertMC and Stack-Aware CompCertX compile into a language that closely mirrors the actual machine code. They both implement the full compilation chain of CompCert but support a less general notion of compositionality.

Separate CompCert (SCC). Kang et al. [2016] have developed a lightweight approach to separate compilation in CompCert. In their approach, all the source modules are written in C and are simultaneously compiled down to CompCert assembly (with the exception that the compilation paths for different modules can vary in the optimizations being applied). Their work has been officially integrated into CompCert as *verified separate compilation* since version 2.7. As such, support of verified separate compilation becomes a de facto requirement for any extension that claims support of the full CompCert. All three compilers introduced in this paper support it. The compositionality provided by Separate CompCert is weaker than Compositional CompCert: it works only for homogeneous programs. On the other hand, the effort for developing SCC is also significantly smaller.

CompCertX (CCX). CompCertX [Gu et al. 2015] extends CompCert to support the compilation of abstraction layers for deep specifications. Like Stack-Aware CompCertX, it supports contextual compilation. The biggest issue with CompCertX is that it does not support compilation of programs with stack-allocated data. Moreover, the non-interference property of memory operations is formalized in an ad hoc way as discussed in Sec. 6.2. In contrast, all three compilers introduced in this paper support stack-allocated data and enforce the non-interference property through the uniform access policy provided by the abstract stack.

CompCertTSO (CC-TSO). CompCert-TSO [Sevcík et al. 2011, 2013] is an extension of CompCert that compiles programs with concurrency based on the relaxed shared-memory model TSO. It is designed to allow most of the proofs of the compiler passes to be done by threadwise simulation arguments (although some passes still need to be proved on full programs). CompCertTSO targets x86 machine code supporting TSO concurrency and with a finite memory. Optimizations that are unsound under the TSO semantics are not implemented in CompCertTSO, such as common subexpression elimination. The implementation also involves deep changes to CompCert.

CompCertS. CompCertS [Besson et al. 2017] is an extension of CompCert that aims to support low-level manipulations such as bitwise operations on pointer values. This work also features a finite memory space and also borrows from Quantitative CompCert the idea of parameterizing the semantics with an oracle for the stack usage of functions. The main difference lies in the way that stack memory is tracked. Our work relies on the abstract stack structure while CompCertS is more *ad hoc*. Because we have an abstract stack, we are able to prove the function inlining and tailcall recognition optimization, while CompCertS excludes these optimizations. Moreover, unlike the present work, they only support whole programs and do not merge the stack blocks into a finite stack block.

The Cerco C Compiler. Finally, we briefly discuss the Cerco (Certified Complexity) compiler [Amadio et al. 2014] that is loosely based on CompCert and formally verified in the Matita proof assistant [Asperti et al. 2011]. The Cerco compiler targets at 8-bit machine code for Intel 8051/8052 microprocessors. It takes a C source program as input and outputs an annotated version of it in addition to generating object code. The annotations reflect the costs in time and space for compiled instructions at the machine code level back to the source level, including the stack consumption at function calls and returns. They are used to analyze and verify the time and space complexity of generated machine code at the source level. In this sense, the approach taken by Cerco is similar to QCC where the usage of annotations can be thought as a generalization of the oracle for stack space. The front-end of Cerco is based on CompCert, while its backend is adapted from a course project which does not perform any advanced optimization such as inlining or tailcall recognition.

8 CONCLUSION

We proposed a lightweight approach to verified compositional compilation to machine code in CompCert. By enriching CompCert's memory model with an abstract stack that keeps track of stack consumption and stack permissions, we developed Stack-Aware CompCert, a complete extension of CompCert with a notion of finite stack supporting a uniform stack access policy. Based on Stack-Aware CompCert, we developed CompCertMC, the first complete extension of CompCert that compiles C programs into low-level code with flat memory spaces, and Stack-Aware CompCertX which supports contextual compilation by exploiting the enrichment of the abstract stack with stack permissions.

ACKNOWLEDGMENTS

We would like to thank anonymous referees for helpful feedbacks that improved this paper significantly. This research is based on work supported in part by NSF grants 1521523, 1715154, and 1763399 and DARPA grant FA8750-15-C-0082. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

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.

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.

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2011. The Matita Interactive Theorem Prover. In *Proc. 23rd International Conference on Automated Deduction (CADE'11)*. Springer-Verlag, Berlin, Heidelberg, 64–69.

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017. CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics. In *Interactive Theorem Proving*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer International Publishing, Cham, 81–97.

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-end. In *Proceedings of the 14th International Conference on Formal Methods (FM'06)*. Springer-Verlag, Berlin, Heidelberg, 460–475.

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-End Verification of Stack-Space Bounds for C Programs. In *Proc. 2014 ACM Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, 270–281.

Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2014. Deep Specifications and Certified Abstraction Layers. Yale Univ. Technical Report YALEU/DCS/TR-1500; http://flint.cs.yale.edu/publications/dscal.html.

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.

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.

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.

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.

Xavier Leroy. 2005–2014. The CompCert verified compiler. http://compcert.inria.fr/.

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

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.

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.

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.

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.

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.

Gang Tan and Greg Morrisett. 2018. Bidirectional Grammars for Machine-Code Decoding and Encoding. *Journal of Automated Reasoning* 60, 3 (2018), 257–277.