The Trusted Computing Base of the CompCert Verified Compiler

David Monniaux, Sylvain Boulmé

The Trusted Computing Base of the CompCert Verified Compiler

David Monniaux Sylvain Boulmé

January 25, 2022

Abstract

CompCert is the first realistic formally verified compiler: it provides a machine-checked mathematical proof that the code it generates matches the source code. Yet, there could be loopholes in this approach. We comprehensively analyze aspects of CompCert where errors could lead to incorrect code being generated. Possible issues range from the modeling of the source and the target languages to some techniques used to call external algorithms from within the compiler.

1 Introduction

CompCert [32, 33, 34] is a formally verified compiler for a large subset of the C99 language (extended with some C11 features): there is a proof, checked by a proof assistant, that if the compiler succeeded in compiling a C program and that program executes with no undefined behavior, then the assembly code produced executes correctly with the same observable behavior. Yet, this impressive claim comes with some caveats; in fact, there have been bugs in CompCert, some of which could result in incorrect code being produced without warning [57]. How is this possible?

The question of the Trusted Computing Base (TCB) of CompCert has been alluded to in general overviews of CompCert [35, 25], but there has been so far no detailed technical discussion of that topic. While our discussion will focus on CompCert and Coq, we expect that much of the general ideas and insights will apply to similar projects and other proof assistants: other verified compilers, verified static analysis tools, verified solvers, etc.

We analyze the TCB of the official releases of CompCert,[1] and two forks: CompCert-KVX,[2] adding various optimizations and a backend for the Kalray KVX VLIW (very large instruction word) core, and CompCert-SSA,[3] adding optimizations based on single static assignment (SSA) form [2, 14]. Versions and changes to these software packages are referred to by git commit hashes. We discuss alternate solutions, some of which already implemented in other projects, their applicability to CompCert, as well as related work.

Sections 2 and 3 analyze the TCB part coming from Coq usage. Section 4 presents the TCB part connecting the Coq specification of CompCert's inputs

[1] `https://github.com/AbsInt/CompCert`
[2] `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx`
[3] `https://gitlab.inria.fr/compcertssa/compcertssa`

(source code) to the user view of these inputs. Sections 5 and 6 analyze the TCB part connecting the Coq specification of CompCert's generated programs to the actual platform running these programs. The conclusion (7) summarizes which TCB parts of CompCert (and its forks) are the most error-prone, and discusses possible improvements.

2 The Coq Proof Assistant

CompCert is mostly implemented in Coq,[4] an interactive proof assistant [52]. Coq is based on a strict functional programming language, *Gallina*, based on the Calculus of Inductive Constructions, a higher-order λ-calculus. This language allows writing executable programs, theorem statements about these programs, and proofs of these theorems. CompCert is not directly executed within Coq. Instead, the Coq code is *extracted* to OCaml code, then linked with some manually written OCaml code. We now discuss how issues in the Coq implementation may impact the correctness of CompCert.

2.1 Issues in Coq Proof Checking

Proofs written directly in Gallina would be extremely tedious and unmaintainable, so proofs are usually built using Coq tactics. While some other proof assistants trust tactics to apply only correct logical steps, this is not the case with Coq: what the tactics build is a λ-term, which could have been typed directly in Gallina if not for the tedium, and this λ-term is checked to be correctly typed by the Coq kernel. This allows tactics to be implemented in arbitrary ways, including calling external tools, without increasing the TCB.

A theorem statement is proved when a λ-term is shown to have the type of that statement (the Curry-Howard correspondence thus identifies statements and types, and proofs and λ-terms). Thus, all logical reasoning in Coq relies on the correctness of the Coq kernel, and some driver routines. In addition to the Coq compiler `coqc` and Coq toplevel `coqtop`, a proof checker `coqchk` provides some level of independent checking.

Coq is a mature development, however "*on average, one critical bug has been found every year in Coq*" [50]. Let us comment on the official list of these bugs.[5] Interestingly, the list classifies their risk according to whether they can be exploited by accident. We can probably assume that the designers of CompCert would not deliberately write code meant to trigger a specific bug in Coq and prove false facts about compiled code: exploiting a Coq bug by mistake in a way sufficiently innocuous to evade inspection of the source code, to accept an incorrect optimization that would be triggered only in very specific cases (to evade being found through testing), seems highly unlikely.

Proofs are checked by Coq's kernel, which is essentially a type-checker for the λ-calculus implemented by Coq (the Calculus of Inductive Constructions with universes). There have been a number of critical bugs involving Coq's kernel, particularly the checking of the guard conditions (whether some inductively defined function truly performs structural induction) and of the universe conditions (Coq has a countable infinity of type universes, all syntactically called

[4] `https://coq.inria.fr/`

[5] `https://github.com/coq/coq/blob/master/dev/doc/critical-bugs`

`Type`, distinguished by arithmetic constraints, which must then be checked for validity). These conditions prevent building some terms having paradoxical types. Furthermore, there are options (in the source code or the command-line) that disable checking guard, universe or positivity conditions. For instance, if one disables the guard condition to build a nonterminating function as though it were a terminating one, it is possible to prove "false":

```
Unset Guard Checking.
Fixpoint loop {A: Type} (n : nat) {struct n}: A := loop n.
Lemma false: False.          Proof. apply loop. exact 0. Qed.
```

`coqchk -o` lists which guard conditions have been disabled—none in CompCert.

The Coq kernel can evaluate terms (reduce them to a normal form), but is rather slow in doing so. For faster evaluation, it has been extended with a virtual machine (`vm_compute`) [20] and a native evaluator (`native_compute`) [6]. Both are complex machinery, and a number of critical bugs have been found in them.[6] In CompCert, there is a few direct calls to `vm_compute`, none to `native_compute`; but there may be indirect calls through tactics calling these evaluators.

2.2 Issues in Coq Extraction

Coq's extractor, as used in CompCert, produces OCaml code from Coq code, which is then compiled and linked together with some other OCaml code. Extraction [36, 37], roughly speaking, corresponds to removing non-computational (proof) content, compensating for some typing issues (see below), renaming some identifiers (due to different reserved words), and of course printing out the result. Coq's extractor and OCaml are in the TCB of CompCert.

OCaml's type safety ensures that, barring the use of certain features that circumvent this type safety (unsafe array accesses, marshaling, calls to external C functions, the `Obj` module allowing unsafe low-level memory accesses...), no type mismatch or memory corruption can happen at runtime within that OCaml code. None of these features are used within CompCert, except for calling C functions implementing the OCaml standard library, and some calls to `Obj.magic`, a universal unsafe cast operator, produced by Coq's extractor.

Calls to `Obj.magic` are used by the extractor to force OCaml to accept constructs (dependent types, arbitrary type polymorphism) that are correctly typed inside Coq but that, when mapped to OCaml types, result in ill-typed programs. The following program is correct in Coq (or in System F) but cannot be typed within OCaml's Hindley-Milner style of polymorphism, so uses `Obj.magic`:[7]

```
Definition m (g : ∀ {T}, list T → list T) : Type :=
  ((g (false :: nil)), (g (0 :: nil))).     Extraction m.
```

The following program, which is similar to some code in the `Builtins0.v` CompCert module, uses dependent types

```
Inductive data := DNat : nat → data | DBool : bool → data.
```

[6] For instance, there used to be a bug with respect to types with more than 255 constructors that allowed proving "false" `https://github.com/clarus/falso`, so ludicrous that it made it into a satirical site `https://inutile.club/estatis/falso/`.

[7] Some System F-like polymorphism was added to OCaml: structure types with polymorphic fields. This is not used by Coq's extractor as of Coq 8.13.2.

```
Definition get_type (d : data) : Type :=
  match d with DNat _ ⇒ nat | DBool _ ⇒ bool end.
Definition extract (d : data) : get_type d :=
  match d with DNat n ⇒ n    | DBool b ⇒ b    end.
Require Extraction.  Extraction extract.
```

Its extraction uses `Obj.magic`:[8]

```
let extract = function DNat n  -> Obj.magic n
                     | DBool b -> Obj.magic b
```

Thus, incorrect behavior in the Coq extractor could, in theory at least, produce OCaml code that would not be type-safe, in addition to producing code not matching the Coq behavior. Is this serious cause for concern? On the one hand, the extraction process is quite syntactic and generic. It seems unlikely that it could produce valid OCaml code that would compile, pass tests, yet occasionally would have subtly incorrect behavior.[9] On the other hand, CompCert is perhaps the only major project using the extractor, which is thus not thoroughly tested. We do not know of any extractor bug that could result in CompCert miscompiling. Another related potential source of bugs comes from the link of OCaml code extracted from Coq and "external" OCaml code. This is discussed in Section 3.2.

Sozeau-et-al [50] study an approach to reduce the TCB of Coq by providing a formally verified (in Coq) implementation of a significant subset of its kernel and paving the road for a formally verified extraction. However, the target language of the extraction (OCaml ?) would still be in the TCB. An alternative solution would be direct generation of assembly code from Gallina, as done by Œuf [40]; however parts of CompCert are currently written in OCaml and would have to be rewritten into Gallina. Œuf extracts Gallina to Cminor, one of the early intermediate languages of CompCert, then produces code using CompCert.[10] CertiCoq[11] [43, 42] also extracts to Clight, which may be compiled with any C compiler.

3 Use of Axioms in Coq

Coq, as other proof assistants, checks that theorems are properly deduced from a (possibly empty) set of axioms. Axioms are also introduced as a mechanism to link Gallina programs to external OCaml code through extraction. Improper

[8] Variants of this example correspond to general algebratic data types (GADTs), another recent addition to OCaml's type system not yet exploited by the extractor.

[9] Coq's bug tracker lists extractor bugs that, to the best of our knowledge, result in programs that are rejected by OCaml compilers.

[10] Other systems meant to generate code from definitions in a proof assistant, generate code directly rather than reuse an existant compiler. This approach is promoted [29] with the argument that such a process is safer than textual extraction to, say, OCaml. This is not so clear to us. On the one hand, extracting (without proof of correctness) Gallina to a subset of OCaml, printing the result, then running the OCaml compiler, surely adds a lot to the TCB. On the other hand, it is typically difficult to get right in a compiler the modeling of the assembly instructions, the ABI, the foreign function interface, as discussed in Section 5. Bugs at that level are caught by extensive testing. Surely, the OCaml code generator, the many libraries using OCaml's foreign function interface, are more thoroughly tested by usage than a code generator used to extract a few specific projects developed in a proof assistant.

[11] `https://github.com/CertiCoq/certicoq`

use of axioms may lead to two forms of inconsistency: logical inconsistency and inconsistency between the Coq proof and the OCaml external code.

3.1 Logical Inconsistency

Coq is based on type theory, with logical statements seen through the Curry-Howard correspondence: a proof of a logical statement is the same thing as a program having a certain type. In other words, a theorem is proved if and only if there is a λ-term inhabiting the type corresponding to the statement of the theorem. An axiom is thus just the statement that a certain constant, given without definition, inhabits a certain type.

The danger of using axioms is that they may introduce inconsistency, that is, being able to prove a contradiction; from which, through *ex falso quodlibet*, any arbitrary statement is provable. Furthermore, it is possible that several axioms are innocuous individually, but create inconsistency when added together.

There are several common use cases for axioms in Coq. One is being able to use modes of reasoning that are not supported by Coq's default logic: CompCert[12] adds the excluded-middle ($\forall P,\ P \vee \neg P$) for classical logic, functional extensionality ($f = g$ if and only if $\forall x,\ f(x) = g(x)$), and proof irrelevance (one assumes that the precise statement of a proof as a λ-term is irrelevant). Meta-theoretical arguments have shown that these three axioms do not introduce inconsistencies.[13]

Another use case for axioms is to introduce names for types, constants and functions defined in OCaml, with a relationship between these and those of the OCaml types and functions to be specified for Coq's extraction facility. For instance, to call an OCaml function `f: nat -> bool list` one would use

```
Axiom f: nat → list bool. Extract Inlined Constant f⇒"f".
```

This is used extensively in CompCert, to call algorithms implemented in OCaml for efficiency, using machine integers and imperative data structures; see 3.3 Similarly, one can refer to an OCaml constant as follows[14]

```
Axiom size : nat.   Extract Inlined Constant size ⇒ "size".
```

Incorrect use of axioms to be realized through extraction can lead to logical inconsistency. Consider, for instance this variant, where the `size` external defini-

[12] CompCert module `Axioms.v` imports module `FunctionalExtensionality` from the Coq standard library, which both states functional extensionality and states proof irrelevance as axioms. Some CompCert modules import the standard `Classical` module, which states excluded-middle as an axiom. Since proof irrelevance is a consequence of excluded-middle, it should be possible to just import `Classical` in `Axioms.v` and deduce proof irrelevance from it.

[13] There is a model of Coq's core calculus in Zermelo-Fraenkel set theory with the Axiom of Choice and inaccessible cardinals [30, 53]. Such a model is compatible with these axioms. Previously, in times when Coq's Set sort was impredicative (it can still be selected to be so by a command-line option), it became apparent that this was incompatible with excluded-middle and forms of choice suitable for finding representatives of quotient sets [11, 12]. This should be a cause of caution, though we think it unlikely to exploit such paradoxes by accident.

[14] This may allow compiling a Coq development once (Coq compilation may be expensive, certain proofs take a lot of time) and then adjust some constants when compiling and linking the extracted OCaml code, maybe for different use cases. This is not used in CompCert, which, instead for flexibility, allows certain features to be selected at run-time through command-line options.

tion is supposed to be a negative natural number (maybe because we mistakenly typed $n < 0$ instead of $n < 10$); one can easily derive `False` from it:

```
Axiom size : { n : nat | n < 0 }.
```

One approach for avoiding such logical inconsistencies is to avoid axioms that specify types carrying logical specifications, that is, proofs (e.g., here $n < 0$); this is anyway a good idea, because such types may also result in mismatches (see 3.2). No OCaml function in CompCert accessed from Coq has Coq type carrying logical specification, with one exception, in CompCert-KVX:

```
Axiom profiling_id: Type.
Axiom profiling_id_eq: ∀ (x y : profiling_id), {x=y} + {x<>y}.
```

These axioms state that there exists a type called `profiling_id` fitted with a decidable equality, both of which are defined in OCaml. This decidable equality is a technical dependency of the decidable equality over instructions.

In order to avoid logical inconsistencies due to axioms referring to external definitions, one can prove that the type in which the `Axiom` command states that there exists a certain term is actually inhabited; this establishes that the axiom does not introduce inconsistency. For instance, one can specify an OCaml constant $n < 10$, to be resolved at compile-time, and exclude logical inconsistency by showing that such a constant actually exists:

```
Axiom size : { n : nat | n < 10 }.
Lemma size_can_exist: { n : nat | n < 10 }.
Proof. exists 0; lia. Qed.
```

This approach is occasionally used in Coq and CompCert for axiomatizing algebraic structures. For instance, Coq specifies constructive reals axiomatically, then provides an implementation that satisfies that specification; CompCert-KVX's impure monad (discussed in Section 3.3) is specified axiomatically, but the authors provide several implementations satisfying that specification [7]. Similarly, the authors could have provided an implementation of `profiling_id` (e.g., natural numbers) and `profiling_id_eq` to show that these two axioms did not introduce logical inconsistencies.

3.2 Mismatches between Coq and OCaml

Though safe, the extractor can be used inappropriately. We have just seen that adding an axiom standing for an OCaml function can, if that axiom is not realizable in Coq, lead to logical inconsistency. Even if the axiom is logically consistent, extraction to arbitrary OCaml code can lead to undesirable runtime behavior.

An obvious case is when, in addition to an axiom specifying a constant referring, at extraction time, to an OCaml function, one adds an axiom specifying the behavior of that function, and that behavior does not match the specification. For instance, one can specify `f` to be a function returning a natural number greater than or equal to 3, then, through extraction, define it to return 0:

```
Axiom f : nat → nat.  Axiom f_ge_3 : ∀ x, (f x) ≥ 3.
Definition g x := Nat.leb 1 (f x).
Extract Constant f ⇒ "fun x → 0".
```

Unsurprisingly, it is possible to prove in Coq that g always returns true, and yet to run the OCaml code and see that it returns false. It is similarly possible to write Coq code with impossible cases that the extractor will extract to `assert false`, and the extracted code will actually reach this statement and die with an uncaught exception—an after all better outcome than producing output that contradicts theorems that have been proved. In the following code, `False_rec _ _` eliminates on *False*, which is obtained from contradiction with $x \geq 3$, and is extracted to an always failing assertion.

```
Program Definition h x := match f x with
  | O ⇒ False_rec _ _          | S O ⇒ False_rec _ _
  | S (S O) ⇒ False_rec _ _    | S (S (S x)) ⇒ x
  end.
```

Axiomatizing the behavior of externally defined functions circumvents the idea of verified software; nowhere in the CompCert source code is there such axiomatization. An equivalent but perhaps more discreet way of axiomatizing the behavior of OCaml function is through dependent types. Consider, again,

```
Axiom size : { n : nat | n < 10 }.
```

It is possible, through extraction mechanisms, to bind `size` to the OCaml constant 11; this is because the type of `size` is extracted to the same exact OCaml type as `nat`, the proof component is discarded. It is then possible to similarly lead the OCaml code extracted from Coq to cases that should be impossible.

The only case of such axiomatization, in CompCert-KVX, is the previously introduced `profiling_id_eq` axiom, which is bound to the `Digest.equal` function from OCaml's standard library, and defined to be string equality. We can surely assume that OCaml's string equality test to be correct, otherwise many things in Coq and other tools used to build CompCert are likely incorrect as well.

It is also possible to instruct the extractor to extract certain Coq types to specific OCaml types, instead of emitting a normal declaration for them. The main use for this is to extract Coq types such as `list` or `bool` to the corresponding types in the OCaml standard library, as opposed to introducing a second list type, a second Boolean type; this is in fact so common that the standard `Coq.extraction.ExtrOcamlBasic` specifies a number of such specific extractions, and so does CompCert. This is not controversial. The extractor also allows fully specifying how a Coq type maps to OCaml, including the constructor and "match" destructor; the only use of this feature in CompCert is in CompCert-KVX for implementing some forms of hash-consing (Sec. 3.4).

An in-depth discussion of further aspects of Coq/OCaml interfacing may be found in Boulmé's habilitation thesis [7].

3.3 Interfacing External Code as Pure Functions

Coq is based on a pure functional programming language; as in mathematics, if the same function gets called twice with the same arguments, it returns the same value. OCaml is an impure language, and the same function called with the same arguments may return different values over time, whether it depends on mutable state internal to the program or on external calls (user input, etc.). By binding Coq axioms to impure functions, we can, again, lead OCaml code extracted from Coq to places it should not go.

For instance, the `z` Boolean expression extracted from this Coq program is `false` though it is proved to be `true`: it calls the same function twice with the same argument and compares the result[15]; but since that function is impure and returns the value of a counter incremented at each call, two successive calls always return unequal values.

```
Axiom f: unit → nat.
Extract Constant f ⇒
  "let count = ref 0 in fun () → count := S (!count); !count".
Definition z: bool := Nat.eqb (f tt) (f tt).
Lemma ztrue: z = true.
  unfold z; rewrite Nat.eqb_refl; congruence.
Qed.
```

CompCert calls a number of OCaml auxiliary functions as pure functions, most notably the register allocator. These functions are "oracles", in the sense that they are not trusted to return correct results; their results are used to guide compilation choices, and may be submitted to checks. Both CompCert-SSA and CompCert-KVX add further oracles.

Could impure program constructs, in particular mutable state, in these oracles, lead to runtime inconsistencies? The code of some of these oracles is simple enough that it can be checked to behave overall functionally: mutable state, if any, is created locally within the function and does not persist across function calls. In the register allocator, there are a few global mutable variables (e.g., `max_age`, `max_num_eqs`), and perhaps it is possible to obtain different register allocations for the same function by running the allocator several times. It seems unlikely that some CompCert code would intentionally call a (possibly computationally expensive) oracle twice with same inputs, then go to an incorrect answer if the two returned values differ. Yet, it is not obvious that this cannot happen.

To avoid such uncertainties, the CompCert-KVX authors encapsulated some of their oracles, in particular oracles used within simulation checkers by symbolic execution [47, 46, 48], inside the *may-return monad* of [7]. The monad models nondeterministic behavior: the same function may return different values when called with the same argument without leading into inconsistent cases. Beyond soundness, a major feature of this approach is to provide "theorems for free" about polymorphic higher-order foreign OCaml code. In other words, this approach ensures for free (i.e., by the OCaml typechecker) that some invariants proved on the Coq side are preserved by untrusted OCaml code [7]. While this technique has been intensively applied within the *Verified Polyhedron Library* [8], it is only marginally used within the current CompCert-KVX, only for a linear-time inclusion test between lists.

This approach however has two drawbacks. Firstly, despite the introduction of tactics based on weakest liberal precondition calculus, the proof effort is heavier than for code written with pure functions without a monadic style. Secondly, all the code calling impure functions modeled within the may-return monad also becomes impure code modeled within that monad, meaning that a significant part of the rest of CompCert (at least the code calling the sequence

[15] This result is computed by the "`Nat.eqb`" Boolean equality over naturals (in contrast, the Coq propositional equality, written "`=`", is only logical).

of optimization phases and their proofs) would have to be rewritten using that monad.[16]

CompCert's Coq code accesses mutable variables storing command-line options through helper functions. This supposes that these variables stay constant once the command line has been parsed, which is the case.

In Coq, all functions must be shown to be terminating (because nonterminating terms can be used to establish inconsistencies). Arguments for the termination of a function are sometimes more intricate and painful to write in Coq than those for its partial correctness, and termination is not really useful in practice: from the point of view of the end-user there is no difference between a terminating function that takes prohibitively long time to terminate, and a nonterminating function. For this reason, some procedures in CompCert and forks that search for a solution to a problem (e.g., a fixpoint of an operator) are defined by induction on a positive number, and return a default or error value if the base case of the induction is reached before the solution is found. `Iteration.PrimIter`, used for instance in the implementation of Kildall's fixpoint solving algorithm for dataflow analysis, thus uses a large positive constant `num_iterations`$=10^{12}$. Such numbers are often informally known as *fuel*.

CompCert-SSA takes an even more radical view: a natural number `fuel` is left undefined, as an axiom, inside the Coq source code, and is extracted to OCaml code `let rec fuel = S fuel`, meaning that `fuel` is circularly defined as its own successor, and in practice acts as an infinite stream of successors. Why that choice? `num_iterations` is a huge constant belonging to the `positive` type, which models positive integers in binary notation; there is a custom induction scheme for this type that implements the usual well-founded ordering on positive integers. In contrast, `fuel` is a natural number in unary notation, on which inductive functions may be defined by structural induction, which is a bit easier than with a custom induction scheme; but it is impossible to define a huge constant in unary notation. The `num_iterations` scheme is cleaner, but we have not identified any actual problem with the `fuel` scheme. The OCaml code extracted from Coq has no way to distinguish `fuel` from a large constant.

The `fuel` trick however breaks if pointer equality is exposed on the natural number type [7]. The following program, defined using a "may return" monad, where `phys_eq_nat` is pointer equality on natural numbers, can be proved not to return true; yet, it does return true at runtime.

```
Definition fuel_eq_pred :=
  match fuel with
  | O ⇒ Impure.ret false
  | S x ⇒ phys_eq_nat fuel x
  end.
```

[16] Much of CompCert is already written in an error monad, with respect to which, the may-return monad is a straightforward generalization. It thus seems feasible to rewrite CompCert with the may-return monad instead of the existing error monad. In practice, this represents a lot of reengineering work. For example, currently, the may-return monad provides a tactic in backward reasoning, based a weakest-precondition calculus. In contrast, CompCert provides a tactic for forward reasoning on the error monad. Thus, defining a tactic on the may-return monad that behaves like the one of the error monad would help in reducing the amount of changes in CompCert proofs.

3.4 Pointer Equality and Hash-Consing

The normal way in Coq to decide the equality of two tree-like data structures is to traverse them recursively. The worst-case of this approach is reached when the structures are equal, in which case they will be traversed completely. Unfortunately this case is frequent in many applications for verified compilation, verified static analysis, etc.: when the data structures represent abstract sets of states (in abstract interpretation), equality signals the equality of these abstract sets, which indicates that a fixed point is reached; equality between symbolic expressions is used for translation validation through symbolic execution [47]. Furthermore, there are many algorithms that traverse pairs of tree-like structures for which there are shortcuts if two substructures are equal: for instance, if this algorithm computes the union of two sets, then if these sets are equal, then the union is the same [39, §5]; being able to exploit such cases has long been known to be important for the speed of static analyzers [4, §6.1.2].

If we were programming in OCaml, we could simply use pointer equality (==) for a quick check that two objects are equal: if they are at the same memory location, then they are necessarily structurally equal (the converse is not true in general). In Coq, a naive formalization of this approach could be:

```
Parameter A: Type.
Axiom phys_eq: A → A → bool.
Axiom phys_eq_implies_eq: ∀ x y, phys_eq x y = true → x = y.
```

This approach is however unsound.[17] We prove that `x_eq_x` and `x_eq_y` are equal; yet in the extracted code, the former evaluates to true, the second to false.

```
Definition x := S O. (* 1 *) Definition y := S O. (* 1 *)
Definition x_eq_x := phys_eq x x. Definition x_eq_y := phys_eq x y.

Extract Inlined Constant phys_eq ⇒ "(==)".
Recursive Extraction x_eq_x x_eq_y.
Lemma same : x_eq_x = x_eq_y.          Proof. reflexivity. Qed.
```

To summarize, OCaml pointer equality can distinguish two structurally equal objects, whereas this is provably impossible for Coq functions: for Coq, `x` and `y` are the same, so they are interchangeable as arguments to `phys_eq`. This is the functionality issue of Section 3.3 in another guise: the same OCaml function must be allowed to return different values when called with the same argument.

The solution used in CompCert-KVX for checking that symbolic values are equal was thus to model pointer equality as a nondeterministic function in a "may return" monad. In this model [7], pointer equality nondeterministically discovers some structural equalities.[18] This solution has one drawback: the whole of the symbolic execution checker is defined within this monad, and the

[17] We saw in the preceding section another possible cause of unsoundness: if circular data structures are defined in OCaml inside inductive types, pointer equality can be used to establish that a term is equal to one of its strict subterms, which is normally impossible, thus leads to an absurd case at execution time. To avoid this, either completely disallow linking to circular terms constructed in OCaml, or restrict pointer equality test to types where such circular terms are not constructed.

[18] In this model, a given Coq term is not necessarily equal to "itself" for pointer equality, because, in a Coq proposition, "itself" implicitly means a structural copy of "itself".

authors unsafely exit from that monad to avoid running much of CompCert through it. It is uncontroversial that pointer equality implies equality of the pointed objects. The only cause for unsoundness in such an approach could be the unsafe exit. Yet, again, why would CompCert-KVX call twice the symbolic execution engine with the same arguments to reach an absurd case for different outcomes?

Opportunistic detection of identical substructures through pointer equality was implemented for instance in Astrée [4]. This approach takes advantage of the fact that many algorithms operating on functional data structures simply copy pointers to parts of structures that are left intact: The opportunistic approach detects that some parts of structures have been left untouched, skipping costly traversals. It however does not work if a structure is reconstructed from scratch, for instance as the result of a symbolic execution algorithms: if two symbolic executions yield the same result, these results are defined by isomorphic data structures but the pointers are different. What is needed then is *hash-consing*: when constructing a new node, search a hash-table containing all currently existing nodes for an identical node and return it if it exists, otherwise create a new node and insert it into the table. Hash-consing is widely used in symbolic computation, SMT-solvers etc.; there exist libraries making it easy in OCaml [15], and the OCaml standard library contains a weak hash-table module, one of the main uses of which is being a basic block for hash-consing.

The difficulty is that, though overall the construction of new objects behaves functionally (it returns objects that are structurally identical to what a direct application of a constructor would produce), it internally keeps a global state inside the hash-table. Several solutions have been proposed to that problem [10]; one is to keep that global state explicitly inside a state monad, which amounts to threading the current state of the hash table through all computations. In the original version from [10], this implied implementing the hash-table by emulating an array using functional data structures, which was very inefficient. Coq 8.13 introduced primitive 63-bit integers and arrays (with a functional interface), optimized for cases where the old version of an updated array is never used anymore [13, §2.3], which, through special extraction directives, may be extracted to OCaml native integers and arrays. That solution was not adopted for CompCert-KVX, only because Coq 8.13 had not yet been released when the project started. Instead, CompCert-KVX has experimented with two alternative approaches for hash-consing.

The first approach used in CompCert-KVX introduces an untrusted OCaml function (modeled as a nondeterministic function within the may-return monad) that constructs terms through the hash-consing mechanism (searching in the hash-table etc.); these terms are then quickly checked for equivalence with the desired terms, using a provably correct checker. For instance, if a term $c(a_1, \ldots, a_n)$ is to be constructed, and the function returns a term t, then the root constructor of t is checked to be c, then the arguments to that constructor are checked to be equal to $a_1, \ldots, a_n$ by pointer equality.[19] This solution does not add anything to the trusted computing base, apart from pointer equality. A may-return monad is used because the OCaml code is untrusted, and in particular is not trusted to behave functionally. The drawback is that, though

[19] A unique identifier is added as an extra field to each object, for reasons including efficient hashing. Structural equality is thus modulo differences in unique identifiers.

the OCaml code will always make sure that there are never two identical terms in memory at different pointer addresses, this is not reflected from the point of view of proofs: in the Coq model (discussed above) of pointer equality within the may-return monad, pointer equality implies structural equality, but structural equality does not imply pointer equality. However, only the former is needed for a symbolic execution engine that checks that two executions are indeed equivalent by structural equality of terms, as in the scheduler in CompCert-KVX [47].

Having to thread a whole computation through a monad, further adding to proof complexity, for actions that are expected to behave functionally overall, is onerous. One solution is to add hash-consing natively inside the runtime system; for instance, the GimML language,[20] from the ML family [19, 17, 18], automatically performs hash-consing on datatypes on which it is safe to do so, which is for instance used to implement efficient finite sets and maps. This can be emulated by a "smart constructor" approach [10], replacing, through the extraction mechanism, calls to the term constructor, term pattern matching, and term equality by calls to appropriate OCaml procedures: the constructor performs hash-consing, the pattern matcher performs pattern matching ignoring the internal-use "unique identifier" field used for hash-consing, and term equality is defined to be pointer equality; appropriate OCaml encapsulation prevents manipulation of these terms except through these three functions, and in particular prevent them from being constructed by other methods than the smart constructor. Assuming that this OCaml code is correct, this is indeed sound, due to the global invariant that there never exist two distinct yet structurally identical terms of the hash-consed type currently reachable inside memory. Because terms can only be built using the smart constructor, and that hash-consing ensures that pointer equality is equivalent to structural equality, pointer equality can indeed be treated as a deterministic function, without need for a monad. This approach has the benefit of an easy-to-understand interface and simple proofs; this was the second approach experimented within CompCert-KVX and was used for the `HashedSet` module [39].

This second approach adds significantly more OCaml code to the trusted computing base than just assuming that pointer equality implies structural equality. Yet, this OCaml code is small, with few execution paths, and can be easily tested and audited. It assumes the correctness of OCaml's weak hash-tables; however, Coq's kernel includes a module (`Hashset`) that is also implemented using these weak hash-tables, so one already assumes that correctness when using Coq.

4 Front-end and semantic issues

CompCert parses C and assigns a formal semantics to it. As such, it depends on a formal model of the C syntax and a formal semantics for it, supposed to reflect the English specification given in the international standard [22]. CompCert supports an extensive subset of C99 [21] (notable missing items are variable-length arrays and some forms of unstructured branching, à la Duff's device) and some C11 features (note that in C11, support for variable-length arrays is optional).[21]

[20] `https://projects.lsv.fr/agreg/?page_id=258` Formerly HimML.

[21] The CH_2O project (`https://robbertkrebbers.nl/research/ch2o/`) aims at formalizing

The formal semantics of C supported by CompCert is called "CompCert C". Converting the source program, given in a text file, to the CompCert C AST (abstract syntax tree) on which the formal semantics is defined, relies on many non-trivial transformations: preprocessing, lexing (lexical analysis), parsing (AST building) and typechecking. Most of them are unverified, but trusted. There are two important exceptions: significant parts of the parser and the typechecker of CompCert C are formally verified. The formally verified parser is implemented using the `Menhir` parser generator, and there is a formal verification of its correctness with respect to an attribute LR(1) grammar [23]. It relies on an unverified "pre-parser" to distinguish identifier types introduced by `typedef` from other identifiers (a well-known issue of context-free parsing of C programs). It produces an AST which is then simplified and annotated with types, by another unverified pass, called "*elaboration*". Finally, the resulting CompCert C program is typechecked, by the formally verified typechecker. This is where the fully verified frontend of CompCert really starts.

Obviously, a divergence between the semantics of C as understood by CompCert and that semantics as commonly understood by programmers to be compiled may lead to problems. Validating such semantics is an important issue [5]. The standard has evolved over time for taking into account common programming practices or for solving some contradictions.[22] CompCert semantics has also evolved to get closer to the standard, see [28]. In the last years, a few minor divergences have been spotted. For instance, there was a minor misimplementation of scoping rules (commit 99918e4) that led the following program to allocate `s` of size 3 (`sizeof(t)` being interpreted with `t` the global variable, whereas the standard mandates it should refer to the `t` variable declared before it on the same line) instead of 4:

```
char t[]={1,2,3};
int main() { char t[]={1,2,3,4}, s[sizeof(t)];
  return sizeof(s); }
```

Another example: CompCert and other compilers accepted some extension to the syntax of C99 (anonymous fields in structures and unions) but assigned slightly different meanings to it (different behavior during initialization, issue 411).

The C standard leaves many behaviors *undefined*—anything can happen if the program exercises such a behavior (the compiler may refuse the program, the program may compile and run but halt abruptly when encountering the message, or may continue running with arbitrary behavior). Some undefined behaviors, such as array access out of bounds, are exploited in malicious attacks. The C standard also leaves many behaviors *unspecified*, meaning the compiler may choose to implement them arbitrarily within a certain range of possibilities—e.g., the order of evaluation of parts of certain expressions with respect to side effects.[23] Actually, distinguishing between *unspecified* and *undefined* behavior in the evaluation order is rather complex: see [27] for a formal semantics. Furthermore, many compilers implement extensions to the standard.

the ISO C11 standard in Coq. This development is unrelated to the formalization inside CompCert.

[22] See an example on `http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm`.

[23] This should not be confused with syntactic associativity, which is fully defined by the standard.

Some deviate from the standard's mandated behavior in some respects.[24]

Many programs, be them applications, libraries or system libraries, rely on the behavior of the default compiler on their platform (e.g., gcc on Linux, clang on MacOS, Microsoft Visual Studio for Windows).[25] If compilation just fails, then issues are relatively easy (though maintaining support for multiple compilers, often through conditional compilation and preprocessor definitions, is error-prone); subtler problems may be encountered when software compiles but has different behavior with different compilers.[26] It may be difficult to narrow differences in outcomes to a bug (including reliance on undefined behavior) or to a difference in valid implementations of unspecified behavior.

The only semantic issue that we know of regarding CompCert's forthcoming version 3.10 is with respect to bitfields. A write to a bitfield is implemented using bitshift and bitwise Boolean operations, and these operations produced the "undefined" value if one of their operands is "undefined". Writing to a bitfield originally stored in an uninitialized machine word or long word, which is the case for local variables, thus results in an "undefined" value, whereas the bits written to are actually defined. Reading from that bitfield will then produce the "undefined" value, as can be witnessed by running the program in CompCert's reference interpreter, which stops complaining of undefined behavior. Fixing this issue would entail using a bit-wise memory model (issue 418).[27] It may be possible to write and prove correct a phase that would replace this "undefined" value by an arbitrary value and thus result in miscompilation. We do not know, however, of any phase that would produce this in CompCert or variants.

CompCert-KVX's test suite includes calling compiler fuzzers CSmith[28] and YarpGen:[29] random programs are generated, compiled with gcc and CompCert-KVX and run on a simulated target—an error is flagged if final checksums diverge.

Due to possible semantic differences for the subset of the C language between the tools that they use for their formal proofs and CompCert, Gernot Heiser, lead designer of the seL4 verified kernel, argues that translation validation of the results of black-box compilation by gcc is a safer route:

> [...] using CompCert would not give us a complete proof chain. It uses a different logic to our Isabelle proofs, and we cannot be certain that its assumptions on C semantics are the same as of our Isabelle proofs.

Another option, for C code produced from a higher-level language by code

[24] For instance, Intel's compiler, at least at some point, deliberately deviated from standard floating-point behavior to produce more efficient code. An option was needed to get standard compliance. In contrast, `gcc` would by default comply with the standard, and enable optimizations similar to Intel's when passed options such as `-ffast-math` or the aptly-named `-funsafe-math-optimizations` [38].

[25] On Linux, compiling software with `gcc -std=c99`, which disables some GNU-specific extensions, often fails. On the KVX, CompCert-KVX includes a kludge for defining a `__int128` type suitable enough for processing system header files.

[26] As an example, C compilers are allowed to replace `a*b+c` by a fused multiply-add `fma(a, b, c)`, which may produce slightly different results. Such replacements may be disabled by a command-line option or a pragma.

[27] Questions of "undefined" and "poison" values are notoriously difficult to get right in semantics; see [31] for a discussion of intricate bugs in LLVM.

[28] `https://github.com/csmith-project/csmith` and [57]

[29] `https://github.com/intel/yarpgen`

generators, is to replace CompCert's frontend by a verified a code generator for that language, directly targeting one of CompCert's intermediate representations (e.g., `Clight`) and semantics, as done for instance for Velus [9] for a subset of the Lustre synchronous programming language.

Some features of the C programming language are not supported by CompCert's formally verified core, but can be supported through optional unverified preprocessing, chosen by common line options: `-fstruct-passing` allows passing structures (and unions) as value as parameters to functions, as well as returning them from a function;[30] `-fbitfields` allows bit fields in structures.[31] Preprocessing implements these operations using lower-level constructs (memory copy builtin, bit shift operators), sometimes in ways incompatible with other compilers—CompCert's manual details such incompatibilities.

In addition, option `-finline-asm` allows inline assembly code with parameter passing, in a way compatible with `gcc` (implementing a subset of `gcc`'s parameter specification). The semantics of inline assembly code is defined as clobbering registers and memory as specified, and emitting an externally observable event. Option `-fall` activates structure passing, bitfields, and inline assembly, for maximal compatibility with other compilers.

Because inline assembly is difficult to use,[32] and because its semantics involves emitting an event, preventing many optimizations, CompCert also provides builtin functions that call specific processor instructions. If a builtin has been given an arithmetic semantics, then it can be compiled into arithmetic operators suitable for optimization; this is the case, for instance, of the "fused multiply add" operator on the KVX.In contrast, instructions that change special processor registers are defined to emit observable events.

5 Assembly back-end issues

The verified parts of CompCert do not output machine code, let alone textual assembly code. Instead, they construct a data structure describing a set of global definitions: variables and functions; a function contains a sequence of instructions and labels. The instructions at that level may be actual processor instructions, or pseudo-instructions, which are expanded by unverified OCaml into a sequence of actual processor instructions. The resulting program is printed to textual assembly code by the `TargetPrinter` module; most of it consists in printing the appropriate assembly mnemonic for each instruction, together with calling functions for printing addressing modes and register names correctly, but there is some arcane code dealing with proper loading of pointers to global symbols, printing of constant pools, etc. Some of this code depends on linking peculiarities and on the target operating system, not only on the target

[30] In C, passing *pointers* to structures that container parameters or are meant to container return values is a common idiom. The language however also allows passing or returning the structures themselves, and this is implement in various ways by compilers, including passing pointers to temporary structures or, for structures small enough to fit within a (long) machine word, directly as an integer register. How to do so on a given platform is specified by the ABI.Parameter passing, with all particular cases, may be a quite delicate and convoluted part of the ABI.

[31] Recently, direct verified handling of bitfields was added to CompCert (commit d2595e3). This should be available in release 3.10.

[32] Inline assembly is so error-prone that specialized tools have been designed to check that pieces of assembly code match their read/write/clobber specification [44].

processor.

5.1 Printing Issues

An obvious source of potential problems is the huge "match" statement with one case per instruction, each mapping to a "print" statement. If the "print" statement is incorrect, then the instruction printed will not correspond to the one in the data structure. Printing an ill-formed instruction is not a serious problem, as the assembler will refuse it and compilation will fail. There have however been recent cases where CompCert printed well-formed text assembly instructions that did not correspond to the instruction in the data structure. The reason why such bugs were not caught earlier is that these instructions are rarely used. Commit 2ce5e496 fixed a bug resulting in some fused multiply-add instructions being printed with arguments in the wrong order; these instructions are selected only if the source code contains an explicit fused multiply-add builtin call, which is rare. In CompCert-KVX, commit e2618b31 fixed a bug—"nand" instructions would be printed as "and"; "nand" is selected only for the rare `~(a & b)` pattern. The bug was found by compiling randomly generated programs.

In some early versions of CompCert there used to be a code generation bug [57, §3.1] that resulted in an exceedingly large offset being used in relative addressing on the PowerPC architecture; this offset was rejected by the assembler. Similar issues surfaced later in CakeML on the MIPS-64 architecture [16] and in CompCert on AArch64 (commit c8ccecc). This is a sign that constraints on immediate operand sizes are easily forgotten or mishandled,[33] and a caution: incorrect value sizes could result in situations not resulting in assembler errors.

5.2 Pseudo-Instructions

In addition to instructions corresponding to actual assembly instructions, the assembler abstract syntax in CompCert features pseudo-instructions, or macro-instructions, most notably: allocation and deallocation of a stack frame; copying a memory block of a statically known size; jumping through a table. The reasons why these are expanded in unverified OCaml code are twofold. First, the correspondence between the semantics of such operations and their decomposition cannot be easily expressed within CompCert's framework for assembly-level small-step semantics, especially the memory model. CompCert models memory as a set of distinct blocks, and pointers as pairs (block identifier, offset within the block); [34] stack allocation and deallocation create or remove memory blocks by moving the stack pointer, which is just a positive integer. Jump tables (used for compiling certain `switch` statements) are arrays of pointers to instructions within the current function, whereas CompCert only knows about function pointers. Second, their expansion may use special instructions (load/store of multiple

[33] For instance, CompCert-KVX generates loads and stores of register pairs on AArch64, with special care: their offset range is smaller than for ordinary loads and stores.

[34] This reflects the C standard's view that variables and blocks live each in their own separate memory space. For instance, in C, comparisons between pointers to distinct variables have undefined behavior [22, §6.5.8]. Some CompCert versions in which pointers truly are considered to be integers have been proposed [3, 41].

registers, hardware loops...) not normally selected, the behavior of which may be difficult to express in the semantics[35] or the memory model. This is typically the case for memory copy; see below.

Stack Frame (De)Allocation Stack (de)allocation pseudo-instructions address the gap between the abstract representation of the memory as a set of blocks completely separated from each other and the flat addressing space implemented by most processors, call frames laid out consecutively, allocation and deallocation amounting to subtracting or adding to the stack pointer. A refined view, with a correctness proof going to the flat addressing level, was proposed for the x86 target [55] but not merged into mainline CompCert.

Loading Constants Certain instructions may need some expansion and case analysis, and possibly auxiliary tables. For instance, on the ARM architecture, long constants must be loaded from constant pools addressed relatively to the program counter; thus emitting a constant load instruction entails emitting a load and populating the constant pool, which must be flushed regularly since the range of adressing offsets is small. Getting the address of a global or local symbol (global or `static`) variable may also entail multiple instructions, and perhaps a case analysis depending on whether the code is to be position-independent, and, in CompCert-KVX, whether the symbol resides in a thread-local program section.[36] The low-level workings of the implementation of these pseudo-instructions rely on the linker performing relocations, on the application binary interface specifying that certain registers point to certain memory sections, etc.

Builtins CompCert allows the user to call special "builtins", dealing mainly with special machine registers and instructions (memory barriers, etc.). These builtins are expanded in `Asmexpand` or `TargetPrinter` into actual assembly instructions.

As an example, consider the memory copy builtin, which may both be used by the user (with `__builtin_memcpy_aligned()`) to request copying a memory block of known size, and is also issued by the compiler for copying structures. Expanding that builtin may go through a case analysis on block size and alignment: smaller blocks will be copied by a sequence of loads and stores, larger blocks using a loop. The scratch registers may be different in each case, and this case analysis must be replicated in the specification; alternatively, the specification may contain a upper-bound on the set of clobbered registers, but in

[35] Hardware loops, on processors such as the KVX, involve special registers. When the program counter equals the "loop exit" register, and there remain loop iterations to be done, control is transferred to the location specified by the "loop start" register. In all existant CompCert assembly language semantics, non-branching instructions go to the next instruction. Modeling hardware loops would thus involve changing all instruction semantics to transfer control according to whether the loop exit is reached, proving invariants regarding the hardware loop registers, etc. This could be worth it if the hardware loops could be selected for regular code, not just builtins, but this itself would entail considerable changes in previous compiler phases.

[36] In C11 [22], the `_Thread_local` storage class specifies that one separate copy of the variable exists for each thread. Typically, a processor register points to the thread-local memory area and these variables are accessed by offsets from that register. CompCert has no notion of concurrency, but on the KVX, some system variables are thread-local and must be accessed as such even from single-threaded programs.

any case no clobbered register should be forgotten. There may also be a complicated distinction of cases regarding which source register is alias to which other source register, or which scratch one. A bug in that builtin, which did not check alignment and generated improper offsets for load instructions, was found in CompCert on AArch64; the assembler would reject the generated code (commit c8ccecc). Another bug in the same builtin, on four architectures (ARM, AArch64, PowerPC, RISC-V), due to an incorrect test about register aliasing, resulted in successful compilation, assembly and linking with incorrect code being emitted (commit c2c871c).

One bug was found in the CompCert-KVX stack frame allocation code, which had no adverse consequence unless a very large stack frame or many parameters were used, which explains why it was not detected earlier (commit fccfa9).

Clobbered Registers Expansions of pseudo-instructions and builtins often use scratch registers. The registers that are clobbered by each pseudo-instruction and builtin are defined in the Coq file (`Asm.v`) giving the semantics of the abstract assembly language. Thus, changes to expansions must affect coherently both the `Asm.v` specification and the `AsmExpand` and/or `TargetPrinter` OCaml module.

In the last few years, several specification bugs about registers clobbered by pseudo-instructions and builtins were found in CompCert, on several architectures. Commit 0df99dc4 fixes several wrong specifications of clobbered registers on AArch64; commit a4cfb9c2 on ARM;commit 39710f78 on RISC-V. It seems that none of these bugs could result in the generation of incorrect code, for the registers that were wrongly specified not to be clobbered were not used by the CompCert code generator to store persistent data. The problem is that it was possible to modify the code generator with full correctness proof, and have CompCert generate incorrect code. For instance, some pseudo-instructions would use the return address register as a scratch register, not specified as clobbered. Some compilers perform leaf function optimization: the prologues and epilogues of functions that never call other functions do not save and restore the return address. CompCert applies this optimization only on the PowerPC architecture, and even then only partially; if one had added this optimization to AArch64 or RISC-V, incorrect code would be generated in leaf functions using the wrongly specified pseudo-instructions, though all proofs would go through.

Bugs in expansion of builtins due to incorrect specification of clobbered registers (or memory), and those related to outcome depending on compiler choices (e.g., register aliases), eerily resemble those due to improper use of inline assembly in C programs [44]. Perhaps similar methods of validation could be used.

As an alternative, we propose moving the parts that deal with case distinctions (register aliasing, sizes, alignments...) out of the untrusted code base into the trusted code base, possibly one pseudo-assembly instruction for each case. For instance, there could be one "memory copy" pseudo-assembly instruction for each different code sequence to be generated, with fixed "clobbered" registers and explicit constraints on alignment, size etc. in the specification of the instruction. Verified Coq code would select the proper pseudo-instruction to use. This would likely avoid bugs due to case distinctions in trusted code, alleviate difficulties in properly specifying the pseudo-instructions and keeping this

specification synchronized with their expansion, and make it easier to perform unit testing on the expansions.

5.3 Microarchitectural Concerns

CompCert-KVX introduced instruction scheduling to CompCert.[37] Instruction scheduling reorders instructions while preserving semantics so as to minimize execution time. Current high-performance processors dynamically reorder instructions, but this is complex and consumes extra energy; *in-order* processors need the compiler to schedule instructions for good performance, taking into account latencies (the number of clock cycles between the operands of an instruction being read and the results being produced) and resource constraints (the number of instructions that can be simultaneously executed; e.g., a processor may be able to execute two instructions at a time, but only one of them may be a memory access, and only one of them may be floating-point).

Tables of resource uses and latencies are cumbersome to build, and often involve access to private documentation and/or reverse engineering; there are thus likely incorrect.[38] Fortunately, all targets of CompCert-KVX have *interlocked* pipelines, meaning that, if a value is read from a register that awaits a write, the instruction is stalled; thus sequential semantics are preserved: the worst that can happen if incorrect latencies are used is that the pipeline stalls for some cycles, which is a performance, not a correctness, issue. In contrast, on processors with non-interlocked pipelines the latencies belong to the semantic definition of the assembly code: a read from a register that awaits a write yields the previous value held in that register. Regarding resource constraints, on a very large instruction word (VLIW) processor, bundles of instructions that exceed resource constraints will be refused by the assembler; on a conventional multiple-issue processor, successive instructions that cannot be issued at the same cycle for lack of resources will be issued sequentially, which is equivalent since the processor preserves sequential semantics even when issuing several instructions. We conclude that pipeline modeling issues have no impact on the correctness of the generated code of CompCert-KVX, but solely on its performance.[39]

5.4 Assembling and Linking

CompCert produces assembly code in textual form, which must then be assembled and linked using another toolchain, such as `gcc` (the GNU Compiler Collection) or `clang` (LLVM). This toolchain is thus within the TCB. Absint GmbH, which sells the commercial releases of CompCert, also sells for certain architectures the Valex tool which matches the CompCert code to the binary code [35, 25]. An alternative is direct generation of machine code, as in CakeML [29]; CompCertELF extends CompCert with a verified assembler for the x86 target [56].

[37] Tristan & Leroy [54] had developed scheduling for CompCert but their developments were not made publicly available, let alone integrated into CompCert releases.

[38] The CompCert-KVX team had private documentation on the KVX; despite that, due to the tedium of building tables, they had a few bugs, as shown by commit logs. Their tables for AArch64 and RISC-V are based on the source code of other compilers.

[39] The situation would of course be very different in the case of a tool bounding worst case execution time through precise processor modeling.

Finally, CompCert's correctness proof was originally meant for a "closed world": a program wholly compiled with it as a single module. In reality, most large C projects are compiled from multiple files which are then linked. The correctness proof was later extended, in version 2.7, to account for separate compilation and linking, following [24]. There have been proposals for more ambitious formalizations of the linking process [49], even implementing a verified linker for a subset of ELF on the x86-32 architecture [56]; [40] Specifying and proving correct a general ELF linker is itself a fairly ambitious project [26].

6 Modeling and Application Binary Interface Issues

The semantics of assembly instructions is defined, for each architecture, in the official manuals from the architecture designers. The *application binary interface* (ABI), specific to each combination of architecture and operating system (or execution environment), defines how parameters are to be passed (in which registers, etc.), what kind of different global symbols exist and how they are accessed, what registers are reserved for system use, how the execution stack is to be laid out, what values the high-order bits of long registers may contain if the register contains a shorter value, etc. In contrast, CompCert's vision of values is somewhat abstract, even at the assembly level, which may pose problems especially when interfacing to other parts of the runtime system.

6.1 Modeling of Values

CompCert considers that a value, e.g., stored in a register, is either a 32-bit integer; a 64-bit integer; a 32-bit single precision floating-point number; a 64-bit double precision floating-point number; a pointer, consisting in a block identifier and an offset; or "undefined", a value that can be refined into any other value, modeling undefined behavior that does not stop program execution (because not yet externally observed). This is, however, an abstraction of reality. Pointers, in reality, are not a pair (block, offset) but a single 32-bit or 64-bit integer. How is a 32-bit value stored in a 64-bit register? Are the higher-order bits indifferent, supposed to be 0 (0-extension) or equal to the sign bit (sign-extension)?

These modeling issues have subtle consequences on the implementation of certain instructions. If the application binary interface specifies that 32-bit values stored in 64-bit processor registers are 0-extended, then the 0-extension operation as defined in CompCert (taking a 32-bit unsigned value and returning the same value as a 64-bit unsigned integer) can be implemented as a no-operation at assembly level (with the special annotation, for the register allocator, that the target register should be the same as the source register).[41] Similarly, if the application binary interface specifies that 32-bit values stored in 64-bit processor registers are sign-extended, then the sign-extension operation as defined

[40] ELF is a standard file format for object code.

[41] This also explains why on some platforms, the code produced by CompCert contains useless moves. If a 32-bit value needs to be extended to 64 bits in a way that both the 32-bit and 64-bit version are live after extension, then these two values, even if they are implemented by the same bit-string, will have to reside in two different registers, since CompCert value semantics distinguishes 32-bit from 64-bit values.

in CompCert can be implemented as a no-operation at assembly level. Finally, the application binary interface may specify that the higher 32 bits of a 64-bit register containing a 32-bit value are arbitrary.

Since none of the CompCert semantics specifies register contents at the bit level, it is up to the backend designer to be consistent in what instructions assume and ensure, and this consistency is never formally verified. Consistency must extend to the foreign function interface: for instance, if a CompCert function is called from a function compiled with another compiler that considers that the higher order 32 bits contain arbitrary values, but CompCert assumes that values are 0-extended, then incorrect behavior may ensue.

The modeling of certain instructions is delicate. The KVX processor supports, in addition to normal loads from memory, *speculative* loads, otherwise known as *non-trapping* or *dismissible* loads. A normal load from an incorrect memory address will trap; on the KVX, a speculative load from an incorrect address returns 0 instead of trapping. Here, "incorrect" is meant with respect to the page tables of the processor. In the intermediate representations of CompCert-KVX, speculative loads from incorrect memory locations return the special value "undefined", whereas a normal load would terminate execution. "Undefined" is a form of "poison value" propagating through operations, e.g., adding it to an integer yields "undefined". The assembly-level semantics, however, defined the value returned by a speculative load from an incorrect memory location as 0, as per the processor documentation. 0 is a valid refinement of "undefined", and the proofs go through. This is however incorrect modeling, because it conflates two different notions: memory accesses invalid with respect to CompCert semantics, and memory accesses invalid with respect to the processor memory management unit:[42] the former are strictly included in the latter:[43], a valid CompCert memory block may occupy a portion of a valid memory page, but the processor will allow accesses to the whole page. Using this incorrect semantics, one could perform a speculative load from a location known to be incorrect with respect to CompCert semantics (for instance, just past the end of a block allocated on the stack) and assume that this load would return 0, whereas this location, when read, would return another value. Commit 5798f56b replaced this default value by "undefined", which is correct: any value is a valid refinement of "undefined".

6.2 Foreign Function Interface

CompCert's application binary interface (ABI) is not specified in a single point in CompCert: it comprises the calling convention, the value conventions implicit in the choice of instructions, etc. The correctness theorem of CompCert relates the execution of a C program, started from the main function, to the execution of the assembly program produced by its compilation, also started from the main function. It does not discuss functions compiled with other compilers calling a function compiled using CompCert. It also assumes that functions called from

[42] Or, rather, the association of the processor memory management unit and the virtual memory subsystem of the operating system.

[43] In the case of memory over-commit by the OS, a valid memory access with respect to CompCert semantics may result in a segmentation violation. We do not consider this issue here, since it is a case of the OS promising resources to the program then reneging on its promises, and thus not supplying a stable execution environment.

CompCert use the same calling convention. As explained in CompCert's manual

> CompCert attempts to generate object code that respects the Application Binary Interface of the target platform and that can, therefore, be linked with object code and libraries compiled by other C compilers.

The manual then describes areas where CompCert's ABI differs from those of other compilers on the targets that it supports. Again, none of these other ABIs were formalized, so the statement of differences in the manual is not based on formal analysis of compatibility, but rather on human analysis.

6.3 Runtime System

The runtime system for C is rather limited compared to other languages. It uses the C standard library supplied by the target platform. CompCert makes no assumption about it—calls to the standard library are just calls to external functions, and the sequence of these calls, as observable events, in the source semantics is reflected in the assembly code—except for the heap memory allocation and deallocation functions `malloc()` and `free()`, which have special treatment and are given specific semantics (creation and destruction of memory blocks in the CompCert memory model). CompCert assumes that this allocator is correct with respect to CompCert's *infinite* memory model. In particular, CompCert assumes that `malloc` always succeeds and never returns the null pointer, which seems unsound: in theory, some formally verified optimizations may incorrectly remove defensive checks against heap overflow. In practice, we do not know of any optimization in CompCert exploiting this model of `malloc`. This assumption of infinite memory has been removed in CompCertS[3], at the price of a large extension of CompCert.

In CompCert, basic floating-point operations have a semantics defined according to IEEE-754 in round-to-nearest mode. This assumes no change to the rounding mode through a library call or direct access to special CPU registers.

Some processors do not support some expensive arithmetic operations (e.g. floating-point operations, division) in hardware. These are replaced by calls to functions in the runtime system, which are axiomatized to perform the required operation by a combination of elementary instructions. This creates a somewhat paradoxical situation where, for the same operation (say, 32-bit integer division): (i) if the operation is implemented in hardware, then it is trusted; (ii) if implemented in software through a call to the runtime system, then it is trusted; (iii) if implemented in software through expansion inside CompCert, then one has to provide a full proof that this expansion implements the operation: its execution coincides with that of the operator on argument values on which this operator has defined behavior. One argument is that the hardware is likely to have been designed from existant floating-point designs and thoroughly tested with many test vectors,[44] Software emulation is likely to be from a well-tested established library,[45] whereas expansion in CompCert probably has not been tested so well.

[44] E.g. the Berkeley hard float library (`https://github.com/ucb-bar/berkeley-hardfloat`) is used in certain RISC-V designs. Yet, they remind potential users that "These units are works in progress. They may not be yet completely free of bugs [...]".

[45] E.g. the Berkeley soft float library (`http://www.jhauser.us/arithmetic/SoftFloat.html`); but, again "Releases 3 through 3c of Berkeley SoftFloat contain bugs in the square root functions

7 Insights and Conclusion

Some natural questions about "verified" software is: how truly safe is it? What kind of constructs should we be considered as suspicious? As more designs come with some formal proofs of correctness, even regulatory agencies have had to provide guidelines [45]. It is of course perilous to draw general conclusions from the analysis of one single project; here are some insights.

None of the problems found were in the verified parts of CompCert: chances seem slim to stumble into a proof checker bug by accident, not notice something is amiss, and think to have proved a theorem that actually does not hold. This explains why the number of bugs found in CompCert releases is many orders of magnitude below usual compilers [51]. By construction, the bugs of CompCert are located in a limited subpart of the software, called its TCB, which may however not be as small as we may naively expect.

Two bugs were found in the front-end elaboration rules, "corner cases" that should be rarely found in real programs (thus their late discovery). A few subtle semantic bugs were also found in some back-ends. However, most bugs were found in the very last part of the back-end, which expands and prints assembly instructions. The causes of these bugs are: (i) the tedium of writing correct printers for each instruction with appropriate operand ordering, and the lack of systematic unit testing of the printers; (ii) the number of different cases, especially in the choice of register arguments, in the expansion of pseudo-instructions, and again the lack of systematic testing that all cases are correct; (iii) the difficulties in keeping synchronized the specification of the pseudo-assembly instructions (in Coq) and the code performing their expansion, in two different files. All these seem to be common software engineering issues, amenable to standard software engineering solutions such as systematic testing of all cases.

All these issues pertain to the specification and trusted (but unverified) parts of the CompCert back-end, which echoes the results of early experiments that found bugs in these parts [57]. In contrast, no bugs due to the use of axioms for interfacing untrusted code, or the use of the extractor to OCaml, were found. In academic circles, however, much attention is often given to doing away with such axioms and the extractor; this may not reflect the most pressing needs. There seems to be a chasm between, on the one hand, what feels relevant and interesting for experts in proof assistants or type theoreticians, on the other hand what would actually increase reliability in verified compilers or similar tools.

In our opinion, the primary focus for increasing trust in CompCert (and removing possible further bugs) should be a validation mechanism of its assembly and ABI specification with respect to the actual execution platform. For example, SAIL provides a formal ISA semantics for ARMv8 that has been tested against the ARM Architecture Validation Suite [1]. However, CompCert cannot be directly plugged on SAIL, because of its more abstract view of the ISA. And this would not solve the issues related to the runtime environment and the ABI.

that may be of concern for some uses. Those bugs are believed to be repaired in Release 3d and later."

Acknowledgements

We wish to thank A. Miquel for helpful references on the metatheory of Coq, as well as L. Gourdin, X. Leroy and C. Six for discussions about CompCert.

References

[1] Alasdair Armstrong et al. "ISA Semantics for ARMv8-a, RISC-V, and CHERI-MIPS". In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290384. URL: https://doi.org/10.1145/3290384.

[2] Gilles Barthe, Delphine Demange, and David Pichardie. "A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert". In: *Programming Languages and Systems (ESOP)*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 47–66. DOI: 10.1007/978-3-642-28869-2_3.

[3] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "CompCertS: a Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics". In: *J. Autom. Reason.* 63.2 (2019), pp. 369–392. DOI: 10.1007/s10817-018-9496-y.

[4] Bruno Blanchet et al. "A Static Analyzer for Large Safety-Critical Software". In: *ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*. ACM. 2003, pp. 196–207. DOI: 10.1145/781131.781153. arXiv: cs/0701193.

[5] Sandrine Blazy. "Experiments in validating formal semantics for C". In: *C/C++ Verification Workshop*. Oxford, United Kingdom, 2007, pp. 95–102. URL: https://hal.inria.fr/inria-00292043.

[6] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full Reduction at Full Throttle". In: *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 362–377. DOI: 10.1007/978-3-642-25379-9_26.

[7] Sylvain Boulmé. "Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)". See also http://www-verimag.imag.fr/~boulme/hdr.html. Habilitation à diriger des recherches. Université Grenoble-Alpes, Sept. 2021. URL: https://hal.archives-ouvertes.fr/tel-03356701.

[8] Sylvain Boulmé et al. "The Verified Polyhedron Library: an Overview". In: *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2018, Timisoara, Romania, September 20-23, 2018*. IEEE Computer Society, 2018, pp. 9–17. DOI: 10.1109/SYNASC.2018.00014. URL: https://hal.archives-ouvertes.fr/hal-02100006.

[9] Timothy Bourke et al. "A formally verified compiler for Lustre". In: *PLDI 2017: Programming Language Design and Implementation*. ACM Press, 2017, pp. 586–601. URL: http://xavierleroy.org/publi/velus-pldi17.pdf.

[10] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. "Implementing and Reasoning About Hash-consed Data Structures in Coq". In: *Journal of Automated Reasoning* (June 2014), pp. 1–34. ISSN: 0168-7433. DOI: 10.1007/s10817-014-9306-0. HAL: hal-00816672. URL: https://hal.archives-ouvertes.fr/hal-00816672.

[11] Laurent Chicli, Loic Pottier, and Carlos Simpson. "Mathematical Quotients and Quotient Types in Coq". In: *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*. Ed. by Herman Geuvers and Freek Wiedijk. Vol. 2646. Lecture Notes in Computer Science. Springer, 2002, pp. 95–107. DOI: 10.1007/3-540-39185-1_6.

[12] Laurent Igal Chicli. "Sur la formalisation des mathématiques dans le Calcul des Constructions Inductives". PhD thesis. Université de Nice, 2003. URL: http://www-sop.inria.fr/lemme/Laurent.Chicli/these_chicli.ps.

[13] Sylvain Conchon and Jean-Christophe Filliâtre. "A persistent union-find data structure". In: *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*. Ed. by Claudio V. Russo and Derek Dreyer. ACM, 2007, pp. 37–46. DOI: 10.1145/1292535.1292541.

[14] Delphine Demange. "Semantic foundations of intermediate program representations. (Fondements sémantiques des représentations intermédiaires de programmes)". PhD thesis. École normale supérieure de Cachan, France, 2012. URL: https://tel.archives-ouvertes.fr/tel-00905442.

[15] Jean-Christophe Filliâtre and Sylvain Conchon. "Type-safe modular hash-consing". In: *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. Ed. by Andrew Kennedy and François Pottier. ACM, 2006, pp. 12–19. DOI: 10.1145/1159876.1159880.

[16] Anthony C. J. Fox et al. "Verified compilation of CakeML to multiple machine-code targets". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Ed. by Yves Bertot and Viktor Vafeiadis. ACM, 2017, pp. 125–137. DOI: 10.1145/3018610.3018621. URL: https://doi.org/10.1145/3018610.3018621.

[17] Jean Goubault. "HimML: Standard ML with Fast Sets and Maps". In: *In 5th ACM SIGPLAN Workshop on ML and its Applications*. Also INRIA RR-2265. ACM Press, 1994. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.4967&rep=rep1&type=pdf.

[18] Jean Goubault. *Implementing Functional Languages with Fast Equality, Sets and Maps: an Exercise in Hash Consing*. Tech. rep. May 1994 version also available. Bull S.A. Corporate Research Center, 1992. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.1757&rep=rep1&type=pdf.

[19] Jean Goubault-Larrecq. *The GimML reference manual*. version 1.0. July 2021. URL: http://www.lsv.fr/~goubault/GimML/refman.pdf.

[20] Benjamin Grégoire and Xavier Leroy. "A compiled implementation of strong reduction". In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*. Ed. by Mitchell Wand and Simon L. Peyton Jones. ACM, 2002, pp. 235–246. DOI: 10.1145/581478.581501. URL: https://doi.org/10.1145/581478.581501.

[21] *International standard—Programming languages—C*. Tech. rep. ISO/IEC, 9899:1999.

[22] *International standard—Programming languages—C*. Tech. rep. ISO/IEC, 9899:2011.

[23] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. "Validating LR(1) Parsers". In: *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012*. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 397–416. URL: http://xavierleroy.org/publi/validated-parser.pdf.

[24] Jeehoon Kang et al. "Lightweight Verification of Separate Compilation". In: *SIGPLAN Not.* 51.1 (Jan. 2016), 178–190. ISSN: 0362-1340. DOI: 10.1145/2914770.2837642.

[25] Daniel Kästner et al. "Closing the gap – The formally verified optimizing compiler CompCert". In: *SSS'17: Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium*. CreateSpace, 2017, pp. 163–180.

[26] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. "The missing link: explaining ELF static linking, semantically". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. Ed. by Eelco Visser and Yannis Smaragdakis. ACM, 2016, pp. 607–623. DOI: 10.1145/2983990.2983996.

[27] Robbert Krebbers. "A Formal C Memory Model for Separation Logic". In: *J. Autom. Reason.* 57.4 (2016), pp. 319–387. DOI: 10.1007/s10817-016-9369-1.

[28] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. "Formal C semantics: CompCert and the C standard". In: *ITP 2014: Interactive Theorem Proving*. LNCS 8558. Springer, 2014, pp. 543–548. DOI: 10.1007/978-3-319-08970-6_36.

[29] Ramana Kumar et al. "Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB - (Short Paper)". In: *Interactive Theorem Proving (ITP)*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 362–369. DOI: 10.1007/978-3-319-94821-8_21.

[30] Gyesik Lee and Benjamin Werner. "Proof-irrelevant model of CC with predicative induction and judgmental equality". In: *Log. Methods Comput. Sci.* 7.4 (2011). DOI: 10.2168/LMCS-7(4:5)2011. URL: https://doi.org/10.2168/LMCS-7(4:5)2011.

[31] Juneyoung Lee et al. "Taming undefined behavior in LLVM". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 633–647. DOI: 10.1145/3062341.3062343.

[32] Xavier Leroy. "A formally verified compiler back-end". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446. URL: http://xavierleroy.org/publi/compcert-backend.pdf.

[33] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009). DOI: 10.1145/1538788.1538814. HAL: inria-00415861.

[34] Xavier Leroy. *The CompCert C verified compiler*. 3.9. an up-to-date version is at https://compcert.org/man/. May 2021.

[35] Xavier Leroy et al. "CompCert – A formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 2016.

[36] Pierre Letouzey. "Extraction in Coq: An Overview". In: *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*. Vol. 5028. Lecture Notes in Computer Science. Springer, 2008, pp. 359–369.

[37] Pierre Letouzey. "Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant)". PhD thesis. University of Paris-Sud, Orsay, France, 2004. URL: https://tel.archives-ouvertes.fr/tel-00150912.

[38] David Monniaux. "The pitfalls of verifying floating-point computations". In: *TOPLAS* 30.3 (May 2008), p. 12. ISSN: 0164-0925. DOI: 10.1145/1353445.1353446. arXiv: cs/0701192. URL: http://hal.archives-ouvertes.fr/hal-00128124/en/.

[39] David Monniaux and Cyril Six. "Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion". In: *LCTES '21: 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, Virtual Event, Canada, 22 June, 2021*. Ed. by Jörg Henkel and Xu Liu. ACM, 2021, pp. 85–96. DOI: 10.1145/3461648.3463850.

[40] Eric Mullen et al. "Œuf: Minimizing the Coq Extraction TCB". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, 172–185. ISBN: 9781450355865. DOI: 10.1145/3167089.

[41] Eric Mullen et al. "Verified Peephole Optimizations for CompCert". In: *SIGPLAN Not.* 51.6 (June 2016), 448–461. ISSN: 0362-1340. DOI: 10.1145/2980983.2908109.

[42] Zoe Paraskevopoulou. "Verified Optimizations for Functional Languages". PhD thesis. Princeton University, Nov. 2020. URL: http://zoep.github.io/thesis_final.pdf.

[43] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. "Compositional optimizations for CertiCoq". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473591.

[44] Frédéric Recoules et al. "Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine". In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1236–1247. DOI: 10.1109/ICSE43902.2021.00113. arXiv: 1903.06407.

[45] *Requirements on the Use of Coq in the Context of Common Criteria Evaluations*. Tech. rep. French National Cybersecurity Agency (ANSSI) and INRIA, Sept. 2020. URL: https://www.ssi.gouv.fr/uploads/2014/11/anssi-requirements-on-the-use-of-coq-in-the-context-of-common-criteria-evaluations-v1.0-en.pdf.

[46] Cyril Six. "Optimized and formally-verified compilation for a VLIW processor". PhD thesis. Université Grenoble Alpes, France, July 2021. URL: https://hal.archives-ouvertes.fr/tel-03326923.

[47] Cyril Six, Sylvain Boulmé, and David Monniaux. "Certified and efficient instruction scheduling: application to interlocked VLIW processors". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 129:1–129:29. DOI: 10.1145/3428197. HAL: hal-02185883.

[48] Cyril Six et al. "Formally Verified Superblock Scheduling". In: *Certified Programs and Proofs (CPP '22)*. Philadelphia, United States, Jan. 2022. DOI: 10.1145/3497775.3503679.

[49] Youngju Song et al. "CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371091.

[50] Matthieu Sozeau et al. "Coq Coq correct! verification of type checking and erasure for Coq, in Coq". In: *Proc. ACM Program. Lang.* 4.POPL'20 (2020), 8:1–8:28. DOI: 10.1145/3371076.

[51] Chengnian Sun et al. "Toward Understanding Compiler Bugs in GCC and LLVM". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, 294–305. ISBN: 9781450343909. DOI: 10.1145/2931037.2931074.

[52] *The Coq Reference Manual*. 8.13.2. Apr. 2021. URL: https://github.com/coq/coq/releases/download/V8.13.2/coq-8.13.2-reference-manual.pdf.

[53] Amin Timany and Matthieu Sozeau. *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*. Research Report RR-9105. KU Leuven, Belgium ; Inria Paris, Oct. 2017, p. 32. URL: https://hal.inria.fr/hal-01615123.

[54] Jean-Baptiste Tristan and Xavier Leroy. "Formal verification of translation validators: A case study on instruction scheduling optimizations". In: *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, Jan. 2008, pp. 17–27. URL: http://xavierleroy.org/publi/validation-scheduling.pdf.

[55] Yuting Wang, Pierre Wilke, and Zhong Shao. "An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290375.

[56] Yuting Wang et al. "CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428265.

[57] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.