The trusted computing base of the CompCert verified compiler

David Monniaux Sylvain Boulmé

VERIMAG

2022-04-06 / ESOP 2022, Garching

https://hal.archives-ouvertes.fr/hal-03541595

Executive summary

The problems are not where you think they are.

Part 1

Problems that pique theoretical computer scientists but are mostly irrelevant in practice.

Part 2

Actual problems you likely have never heard of.

(Much more in the paper than in the talk!)

Plan

CompCert

Rationale

The only industrial compiler with a **formal proof of correctness** (see also CakeML).

Versions

- **Official** `https://github.com/AbsInt/CompCert`
- Commercial `https://www.absint.com/compcert/index.htm`
- (Ours) For Kalray KV3 & extended optimizations, esp. for Risc-V, AArch64 `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx`
- With SSA `https://compcertssa.gitlabpages.inria.fr/`

Proof of semantic preservation

Semantics

C source and (idealized) assembly source semantics as traces of observable events:

- **calls to external functions** (I/O...)
- special processor instructions (system registers)
- read and writes on volatile variables

Theorem

If compilation succeeds, then any execution (up to undefined behavior) of the C program yields the **same execution trace** for the assembly program.

Questions

"Theorem formally proved within the Coq proof assistant."

- What is actually guaranteed?
- Are there loopholes?
- What does it rely on?
- What assumptions?

Plan

Coq

Coq (as opposed to, e.g., PVS) does not trust its own tactics.
Tactics produce proof terms, going to a typechecker (**small kernel**).
Typechecker reduces terms in typed λ-calculus (Calculus of Inductive Constructions with universes)

Trust in the reduction strategies, including based on virtual machine and native compilation.

`https://github.com/coq/coq/blob/master/dev/doc/critical-bugs`
Some rare bugs could be exploited inadvertently.

Extraction

Coq code in CompCert is extracted to OCaml and linked to OCaml code.

The TCB includes the **extractor**, the OCaml native-code compiler (same for Coq itself).

OCaml code is memory-safe...

- if you don't call external C functions (but the standard library?)
- if you don't use the `Obj` module and `unsafe` array/string functions

Some `Obj.magic` in extracted code (e.g. System F does not map to ML's Hindley-Milner polymorphism), which should be type-safe since CIC is type-safe.

Coq and extractor

We don't think there is an important hazard of Coq, OCaml or extractor bugs

- that could be exploited unintentionally
- that would be undetected (memory corruption from C code / `Obj` typically crashes OCaml programs)

Axioms in Coq

As in any other logical system, introducing contradictory axioms allows proving false theorems.

CompCert uses classical axioms (excluded middle, functional extensionality, proof irrelevance), consistent with CIC.

We did not detect misuse of axioms.

Linking to OCaml code

Coq definitions or axioms may be specified to be extracted to OCaml types, terms:

- for e.g. using OCaml's `list` and `bool` types instead of declaring new (isomorphic) list and Boolean types extracted from Coq
- for calling external OCaml functions (oracles)
- for implementing smart constructors

Oracle calls

Used to run procedures not easily coded in Coq (hash tables, calls to external tools...), e.g. **register allocator**

Axiom in Coq, "extract constant as" OCaml code

Mismatch of OCaml types (e.g., integer vs Boolean) detected by OCaml typechecker.

CompCert avoids specifications such as "function that returns $n \leq 3$", because $n \leq 3$ would be trusted.

Untrusted oracle use

Oracles used

- for picking heuristic choice
- for computing untrusted results

Untrusted results must be fed to trusted verifier (e.g. check that a fixed point is a fixed point, that a register allocation is valid...).

Coq is purely functional, OCaml isn't.

Hash-consing

= create one single copy of each active term
two different solutions in CompCert KVX

Minimal trust solution

An untrusted “factory” returns terms (from impure monad), which are checked for correctness.
The only trusted part: **pointer equality of two OCaml terms implies term equality** (surprisingly controversial)

More trusted solution

Extract the constructor to a “smart constructor” with hash-consing.
Trust a few lines of OCaml

Executive summary so far

Did not see any way to cause miscompilation.

Plan

C semantic issues

What if CompCert's notion of C is not the programmer's?

We have conducted extensive testing by comparison with gcc (applications and randomly generated code). No discrepancy from compiled code...

A little fun question

```
#include <stdio.h>

struct toto { unsigned fx : 3; };

int main() {
  struct toto m;
  m.fx = 2;
  if (m.fx == 2) {
    printf(”coincoin\n”);
  }
  return 0;
}
```

Related

```c
#include <stdio.h>
#include <stdlib.h>

struct toto { unsigned fx : 3; };

int main() {
  struct toto *m = malloc(sizeof(struct toto));
  m->fx = 2;
  int r = (m->fx != 2);
  free(m);
  return r;
}
```

Malloc is OK

```
#include <stdio.h>
#include <stdlib.h>

struct toto { unsigned fx; };

int main() {
  struct toto *m = malloc(sizeof(struct toto));
  m->fx = 2;
  int r = (m->fx != 2);
  free(m);
  return r;
}
```

Global ?

```
#include <stdio.h>

struct toto { unsigned fx : 3; } m;

int main() {
  m.fx = 2;
  return (m.fx != 2);
}
```

Explanation

Bitfields are internally converted into bit shifts / masks on integers.

If a word is uninitialized (“value undefined” or Vundef), operations all yield Vundef, thus undefined behavior.

Why is this a problem

The compiler may substitute anything for undefined behavior.

This is why CompCert is cautious with optimizations and is generally designed **not to optimize based on undefined behavior**.

- Many programs rely on undefined behavior.
- CompCert's source semantics wrongly introduces undefined behavior on bitfields (and possibly other topics).
- Would it really gain speed?

Target semantics

CompCert (`Asm.v` module) has an **idealized** semantics of assembly code.

It reflects not only the actual assembly instructions (ISA) but also the application binary interface (ABI) of the operating system (linking, etc.).
e.g. “how do I get the address of a global symbol?”

Idealized version: talks of “pointer”, “32-bit integer”, “64-bit integer” and not about their bit representation.
Is a 32-bit value 0-extended? sign-extended? higher bits irrelevant?

ABI not specified in one clean place but spread over.
May be different from gcc/clang (documented).

Pseudo-instructions

Some instructions are not real CPU instructions but are expanded by **unverified OCaml code**.

- "Too magical", impossible to express/prove in idealized semantics (e.g., memcpy builtin, stack frame allocation/deallocation).
- ABI issues ("get the address of a thread-local variable")

The OCaml code and the Coq specification of the instructions must match.

What we found

- On multiple architectures, **some pseudo-instructions were incorrectly specified** (missing clobbered registers).
- Some of this was found when developing optimizations, proved correct and **generating incorrect code** (because of incorrect preexisting specification).
- **Some instructions were incorrectly printed** but resulted in syntactically correct assembly code (order of arguments of fused multiply-add on x86).

What would be needed

Systematic testing of the specified semantics of the instructions and pseudo-instructions with respect to generated assembly code?

Conclusion

- Computer science researchers worry about explicit axioms such as “pointer quality implies equality”, and pure functionality, which have not resulted in any miscompilation.
- Source semantics have some odd corners (bitfields).
- Target semantics, trusted transformation and printing have been insufficiently tested (order of arguments printed incorrectly) and need validation.

`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx`

Functionality

Coq is a pure functional language. $f(x)$ always has the same value. OCaml is an impure language. $f(x)$ can change values if references are used.

It is possible to reach "unreachable" cases in Coq code by calling an OCaml impure function that changes value.

Seems difficult to exploit by mistake (call the same OCaml function twice with same arguments intentionally, exploit "unreachable" case).

Impure monad

The register allocator is trusted to be functional despite using mutable state.

A more elegant solution: an **impure "may return" monad**.

Used for some external calls in CompCert KVX (hash-consing engine for symbolic execution).