CompCert: Formal Verification of a Realistic Compiler

Communications of the ACM

Xavier Leroy

Motivation — Limitations of Program Analysis

Software reliability relies on program analysis techniques:

- **Static Analysis:** scalable, but produces false positives
- **Dynamic Analysis:** precise, but misses behaviors

Goal: obtain mathematical correctness guarantees through **formal verification**.

Motivation — Limitations of Static Analysis

```c
void good_func(char *arg) {
  char tmp[128];
  strncpy(tmp, arg, sizeof(tmp));
}

void bad_func(char *arg) {
  char tmp[128];
  strcpy(tmp, arg);
}

void main(int argc, char **argv) {
  good_func(argv[1]);
}
```

Our oracle labels any program containing `strcpy` as buggy.

However, the vulnerable function is never actually executed in this instance.

Motivation — Limitations of Dynamic Analysis

```c
int main(int argc, char **argv) {
  int x = atoi(argv[1]);
  if (x > 10) {
    ok();
  } else {
    if (x < 5) {
      ok();
    } else {
      bug();
    }
  }
  return 0;
}
```

Hybrid Symbolic Execution

- x is initially symbolic
- Branches are concretized during execution

After taking the first `else` branch: $x \leq 10$

The executor concretizes: $x = 4$

Result: the buggy region $5 \leq x \leq 10$ is missed.

Motivation — The need for Formal Verification

Limitations motivate **formal verification** to provide mathematically rigorous correctness guarantees.

However, verification faces two main challenges:

1. **Computational limits:** Fully automatic formal verification is generally **undecidable**.
2. **Environmental assumptions:** Verifying code alone cannot guarantee safety if trusted components fail.
    - Operating System
    - SMT solver
    - **Compiler**

Motivation — CompCert: A Compiler Verification Solution

CompCert is the first formally verified compiler for *Clight*, a large subset of C.

- Proves in Coq that generated binaries preserve source program semantics.
- Supports classic optimizations like constant propagation and register allocation.

Front End

Clight ⇒ Cminor

Back End

Cminor ⇒ PowerPC

Motivation — CompCert Limitations

Unsupported Features

- `goto`
- `long long` / `long double`
- Duff's device
- `struct` / `union` by value
- Variadic functions

Semantic Simplification

Clight expressions are **side-effect free**.

Expressions cannot modify memory during evaluation.

This greatly simplifies the formal semantics and makes machine-checked proofs practical.

Tradeoff: reduced language complexity $\Rightarrow$ feasible formal verification

Trusted Compilation — Semantic Preservation

Let S be the source program, C the compiled code, and B a behavior.

Notation $L \Downarrow B$ states that language L executes with behavior B.

Ideal Target (Strict Equivalence):

$$\forall B, S \Downarrow B \iff C \Downarrow B \tag{1}$$

Strict equivalence requires deterministic semantics. This is difficult for C due to undefined behaviors like:

- Out-of-bounds memory access
- Null pointer dereference

Trusted Compilation — Semantic Preservation

To make (1) practical, we restrict analysis to safe programs (programs that do not go wrong):

$$S_{\text{safe}} \Rightarrow (\forall B, C \Downarrow B \Rightarrow S \Downarrow B) \qquad (2)$$

Let $\mathbb{W}$ denote the set of wrong behaviors. Under deterministic semantics, (2) simplifies to:

$$\forall B \notin \mathbb{W}, S \Downarrow B \Rightarrow C \Downarrow B \qquad (3)$$

Trusted Compilation — Semantic Preservation

Compiled code must satisfy the program specification.

We write $C \vDash \mathrm{Spec}$ if:

- C cannot go wrong.
- All behaviors of C satisfy Spec.

A correct compiler preserves specifications:

$$S \vDash \mathrm{Spec} \Rightarrow C \vDash \mathrm{Spec} \tag{4}$$

Special case (safety preservation):

$$S_{\mathrm{safe}} \Rightarrow C_{\mathrm{safe}} \tag{5}$$

Trusted Compilation — Verified, validated, certifying compilers

A verified compiler modeled as a function Comp:

$$\mathrm{Comp}(S) = \begin{cases} \text{Error, if compilation fails} \\ \mathrm{OK}(C), \text{ otherwise} \end{cases}$$

It satisfies the correctness property:

$$\forall S, C, \mathrm{Comp}(S) = \mathrm{OK}(C) \Rightarrow S \approx C \quad (6)$$

Core Guarantee: A verified compiler either flags an error or produces semantically correct code.

Trusted Compilation — Verified, validated, certifying compilers

A validator is a boolean function:

$$\text{Validate}(S, C)$$

It returns `true` if $S \approx C$, and `false` otherwise.

Because $S \approx C$ is generally undecidable, validators are incomplete and may return `false` even if correctness holds.

A validator is verified if:

$$\forall S, C, \text{Validate}(S, C) = \text{true} \Rightarrow S \approx C \quad (7)$$

Trusted Compilation — Verified, validated, certifying compilers

Combining an unverified compiler with a verified validator yields the same guarantees as a verified compiler:

$\mathrm{Comp}'(S) =$
match $\mathrm{Comp}(S)$ with
| Error $\rightarrow$ Error
| $\mathrm{OK}(C) \rightarrow$ if $\mathrm{Validate}(S, C)$ then $\mathrm{OK}(C)$ else Error

Trusted Compilation — Verified, validated, certifying compilers

Proof-Carrying Code (PCC)

The compiler generates:

- Compiled code C
- A certificate proof π showing $C \vDash \mathrm{Spec}$

The Spec defines behavioral targets like type or memory safety.

Challenge: Extending this approach across complex Intermediate Representations (IR).

Trusted Compilation — Composition of compilation passes & Summary

Modern compilers leverage multiple intermediate representations.

A compiler pipeline with n intermediate languages requires multiple verified translation passes.

$$\begin{aligned}&\mathrm{Comp}(L_1) = \text{match } \mathrm{Comp}_1(L_1) \text{ with}\\ &\quad \mid \text{Error} \rightarrow \text{Error}\\ &\quad \mid \mathrm{OK}(L_2) \rightarrow \mathrm{Comp}_2(L_2)\end{aligned}$$

The multi-pass semantic preservation theorem combines formulas (3) and (6):

$$\forall S, C, B \notin \mathbb{W}, \mathrm{Comp}(S) = \mathrm{OK}(C) \wedge S \Downarrow B \Rightarrow C \Downarrow B$$

CompCert Overview

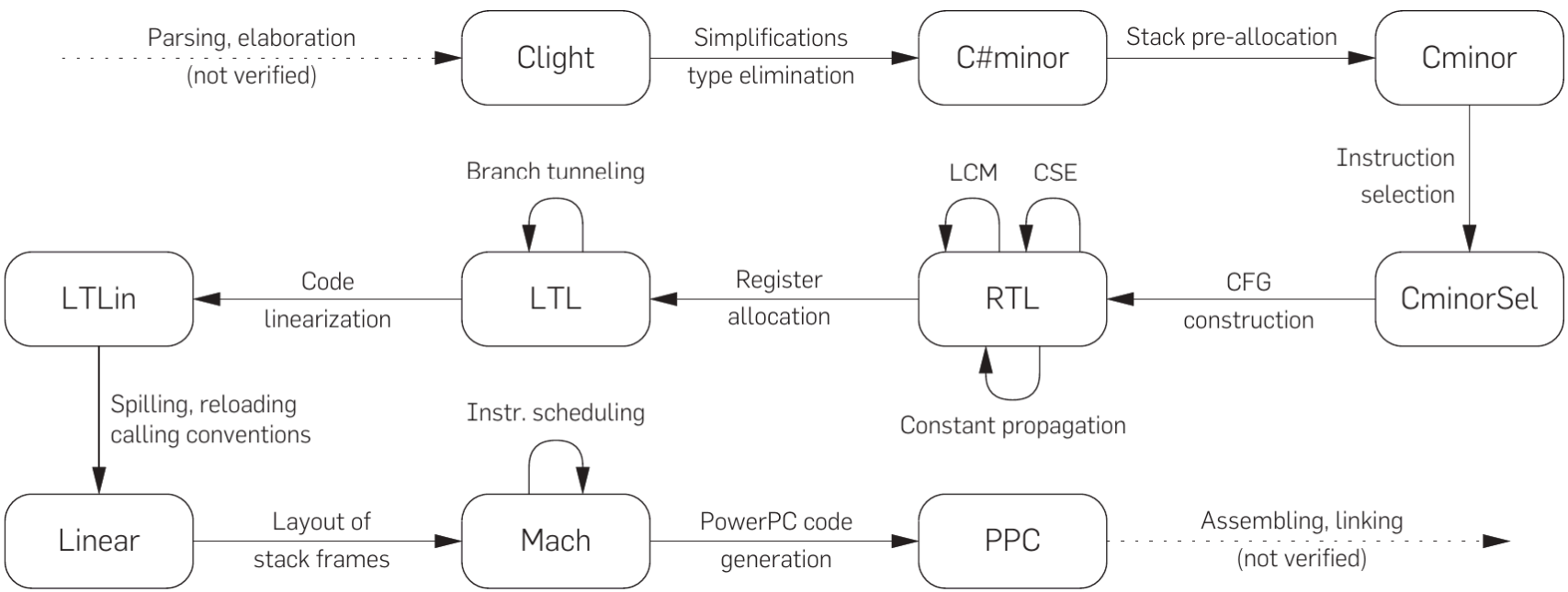

Figure 1: Compilation passes and intermediate languages.

CompCert Overview

From CminorSel to RTL, code is translated into a Control Flow Graph (CFG). Nodes represent machine-level instructions operating over pseudo-registers.

The RTL representation simplifies data-flow optimizations:

- Constant Propagation
- Common Subexpression Elimination
- Lazy Code Motion

Next, register allocation maps pseudo-registers to LTL (hardware registers or abstract stack locations) via graph coloring.

Register Allocation

Register allocation is a critical optimization. Other optimizations primarily serve to maximize its efficiency.

The core goal is keeping hot variables in registers, spilling remaining ones to the stack.

Allocation is modeled as a **graph coloring** problem:

> Interfering variables must not share the same register

Because graph coloring is NP-hard, optimal register allocation is also NP-hard.

Register Allocation — RTL Intermediate Language

Instructions

$$
\begin{aligned}
i ::= \ & \mathrm{nop}(l) \\
& \mid \mathrm{op}(\texttt{op}, \vec{r}, r, l) \\
& \mid \mathrm{load}(\kappa, \texttt{mode}, \vec{r}, r, l) \\
& \mid \mathrm{store}(\kappa, \texttt{mode}, \vec{r}, r, l) \\
& \mid \mathrm{call}(\texttt{sig}, (r \mid \mathrm{id}), \vec{r}, r, l) \\
& \mid \mathrm{tailcall}(\texttt{sig}, (r \mid \mathrm{id}), \vec{r}) \\
& \mid \mathrm{cond}(\texttt{cond}, \vec{r}, l_{\mathrm{true}}, l_{\mathrm{false}})
\end{aligned}
$$

Control-flow graphs

$$g ::= l \to i$$

Internal functions

$$
\begin{aligned}
F ::= \{ & \ \texttt{name} = \mathrm{id}; \ \texttt{sig} = \mathrm{sig}; \\
& \ \texttt{params} = \vec{r}; \ \texttt{stacksize} = n; \\
& \ \texttt{entrypoint} = l; \ \texttt{code} = g \\
\}
\end{aligned}
$$

External functions

$$\mathrm{Fe} ::= \{\texttt{name} = \mathrm{id}; \ \texttt{sig} = \mathrm{sig}\}$$

Register Allocation — State Representation

The intermediate language supports an infinite supply of **pseudo-registers**. To prove semantic preservation, the execution state is formalized as a 6-tuple:

$$S(\Sigma, g, \sigma, l, R, M)$$

- Σ: Call stack model
- g: Current CFG
- σ: Memory block for activation record
- l: Current program point
- R: Pseudo-registers map ($R : r \to v$)
- M: Global memory map

Register Allocation — Lock-Step Simulation

During allocation, every original transition maps to exactly one transformed transition, producing a **lock-step simulation diagram**:

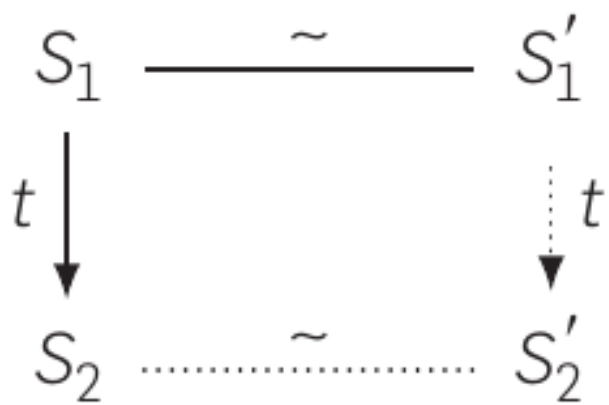

Figure 2: Solid lines represent hypotheses; dotted lines represent conclusions.

The relation ($\sim$) ensures structural identity for memory layouts and observable behaviors.

Register Allocation — Semantic Invariants

Let the transformed state be $S(\Sigma', g', \sigma', l', R', M')$. The mapping function $\varphi(r)$ colors pseudo-registers to physical registers or stack slots.

Semantics are preserved if states satisfy these invariants:

- **Control Flow:** $l' = l$ and $g' = \varphi(g)$
- **Memory Space:** $M' = M$ and $\sigma' = \sigma$
- **Register Agreement:** For all pseudo-registers r **live** at point l: $R(r) = R'(\varphi(r))$

Implementation Details

CompCert is developed entirely within the **Coq proof assistant**, serving as both the programming and verification framework.

- Scale: 42,000 lines of Coq code.
- Distribution:
  - 76% Proof scripts
  - 14% Compiler implementation
  - 10% Language semantics / specifications
- Extraction: Coq code is automatically extracted into pure functional OCaml.

Evaluation

Experimental Setup

- **Hardware:** 2 GHz PowerPC 970 "G5" processor.
- **Benchmarks:** Small test suite matching the subset of C supported by CompCert.
- **Baseline:** Compared against `gcc 4.0.1` at optimization levels `-O0`, `-O1`, and `-O2`.

Performance Outcomes

- Generated code is **2x faster** than unoptimized GCC (`gcc -O0`).
- Competitive with standard optimizations:
  - Only **7% slower** than `gcc -O1`
  - Only **12% slower** than `gcc -O2`

Key Result: Verified compilation remains competitive with optimized GCC.

Evaluation

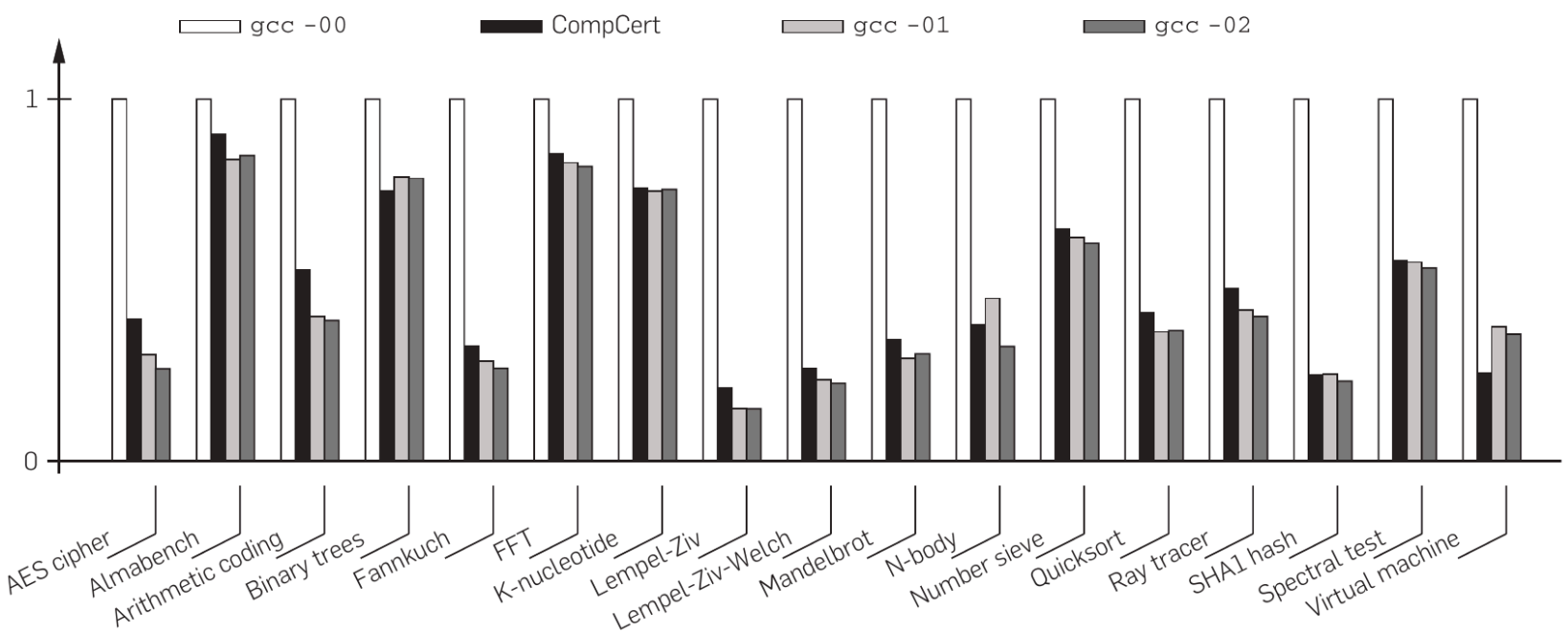

Figure 3: Relative execution times of compiled code.

DEMO TIME!

Conclusion

CompCert demonstrated that a **realistic optimizing compiler** can be formally verified end-to-end.

Key contributions:

- Formal semantic preservation proofs for each compilation pass
- Verified implementations of classic optimizations
- Practical performance competitive with GCC
- Machine-checked proofs developed entirely in Coq

Main Takeaway:

Formal verification can provide strong guarantees for critical software systems without sacrificing practicality.

Thank you!
Questions?