🔍 Can We Trust Compilers? A 10-Year Journey Into Making Them Smarter and Safer

Anand Kumar Keshavan, July 2025

Wait, What's a Compiler Again?

Imagine you write a message in English and need it translated into Morse code so your robot friend can understand and follow your instructions. A compiler is like that translator—it takes your code (written in languages like C, Python, or Java) and turns it into a form the computer can actually run.

But what if the translator messes up?

That's where things get serious. If the compiler introduces even a tiny mistake while optimizing or translating your code, it could cause your program to behave strangely—or dangerously—especially if it's part of an airplane, medical device, or self-driving car.

That's why, over the last 10 years, researchers have been trying to **prove** that compilers do their job correctly. Not just test them—prove them, mathematically.

What Can Go Wrong?

Let's say you wrote code to safely control the brakes of a car. You double-checked it, tested it, even had experts look at it.

Then, the compiler comes along and tries to make your code faster (a process called **optimization**). In doing so, it makes a tiny change that causes the brakes to fail under rare conditions. Your source code was perfect, but the compiled version isn't. Yikes.

This isn't just theory—it happens in real life. Big-name compilers like GCC and LLVM have been found to quietly introduce bugs during optimization. That's scary.

The Science of Proving Compilers Correct

There are two main ways researchers have tackled the problem of buggy compilers:

1. Whole-Compiler Verification

This method tries to **prove the entire compiler is correct**—from start to finish. Imagine creating a perfect recipe that always gives you the same cake, no matter how many times you bake it. That's what projects like **CompCert** do.

With this method:

- Every part of the compiler is verified using mathematical logic.
- The team uses special software (like Coq) to write both the compiler and the proof.
- It's super reliable—but it takes **a lot** of effort.

Example: **CompCert**, a verified C compiler used in planes and cars, took years to build but has never produced a wrong program in tests.

2. Translation Validation (TV)

This one's more like spell-checking the finished essay. Instead of proving the entire compiler is perfect, TV checks **each compilation** after it happens. If something looks wrong, it raises a flag.

It's more practical for big systems like LLVM, which are too massive to verify entirely. TV uses smart software to compare the original and translated programs and ask: "Did this change what the code was supposed to do?"

Tools like **Alive** and **Vellvm** are used to do this automatically.

How Does It Work in Practice?

Let's look at two examples:

CompCert: The Gold Standard

- Breaks down compilation into small, simple steps.
- Uses about 20 "mini-languages" to gradually lower your code from C to machine language.
- Each step is verified using Coq.

- Ensures that what you wrote is **exactly** what the machine runs.
- Used in real industries—like avionics.

LLVM: The Real-World Giant

- Super complex, written in C++, and widely used.
- Instead of verifying the whole thing, tools validate its output step-by-step.
- Uses “SMT solvers” (math robots) to check whether optimized code still does the same thing as the original.

Types of Compiler Optimizations (and Their Verification Status)

Think of optimizations like smart tricks a compiler uses to make your code faster. These include:

Optimization Type	How Verified?	Status
Peephole tweaks (tiny rewrites)	Checked with SMT tools (Alive)	✅ Mature
Loop changes (speed up loops)	Verified with math (Polyhedral model)	🛠️ Ongoing
Register Allocation (manage variables)	Very hard—needs special tools	🧪 In progress
Concurrency (multi-threading)	Still very tricky	🔬 Research ongoing
Floating-point math (like 1.0 + 2.0)	Needs special models	🔍 Still being figured out

---

The New Challenge: Machine Learning Compilers

Now compilers are getting AI upgrades.

Instead of having fixed rules, they learn from data how to optimize code better than human-written rules. Sounds cool, right?

But there's a problem: How do we **prove an AI didn't mess up**?

Because AI models are black boxes, it's nearly impossible to predict or explain their behavior. This is one of the **biggest open problems** in compiler verification today.

The Future: Working Together Across Languages

Most software today mixes multiple languages—like a website using JavaScript, Python, and C libraries all at once. But verified compilers usually assume the whole program is written in just one language.

That's a problem.

The next big step? Making sure verified parts of programs can **safely interact** with unverified parts. This is called **compositional verification**—and it's the holy grail of modern compiler research.

Final Thoughts: Where We're Headed

In 2003, computer scientist Tony Hoare proposed the idea of a "verifying compiler"—one that could help you write correct programs and then make sure those programs stay correct as they're turned into machine code.

We're not there yet. But in the last 10 years, we've made massive strides.

- We've built verified compilers like CompCert.
- We've created tools to check each compilation like Alive and Vellvm.
- And we're now exploring how to verify AI-driven optimizations and multi-language software.

The goal? To make sure that no matter what you write—or how clever your compiler is—you can trust that the final program will do what you expect.

Want to Learn More?

- CompCert project

- Alive for LLVM

- Vellvm project
- Translation Validation explained

Would you like this turned into a downloadable PDF or blog-ready HTML version?