Finding Miscompiles for Fun, Not Profit

Or: You don't need access to Claude Mythos to spend $10,000 in an afternoon.

JUSTIN LEBAR

MAY 28, 2026 • PAID

215 8 Share

Update June 1: *The day after we published, Anthropic released Opus 4.8 and "ultracode" mode in Claude Code. Our preliminary experiments indicate that together these are significantly better at filtering out low-severity bugs, and that the cost per medium-to-high severity bug found is maybe 1/5 (with very large error bars) that of the workflow described in this article.*

I've worked on compilers for ML for the last decade across Google, Waymo, and OpenAI. This includes CUDA support in clang, XLA:GPU, Triton, and OpenAI's custom hardware. I've seen stuff. But over the past week or so I had one of the most unsettling experiences of my career: In one afternoon, I spent more than $10,000 running AI agents over compiler code, finding hundreds of plausible bugs in LLVM, including many miscompiles and at least one that's Quite Serious. This is the story of how I got here and where we might be going.



collaboratively wrote a fuzzer. The basic idea is to generate a random program, run it through part of the compiler, and then check that the resulting program after compilation does the same thing as the original program (usually just by running the two programs). I spent a few weeks on it, and I found and fixed five bugs in instcombine, LLVM's peephole optimization pass. After that, my fuzzer started taking longer to find bugs, and I lost interest.

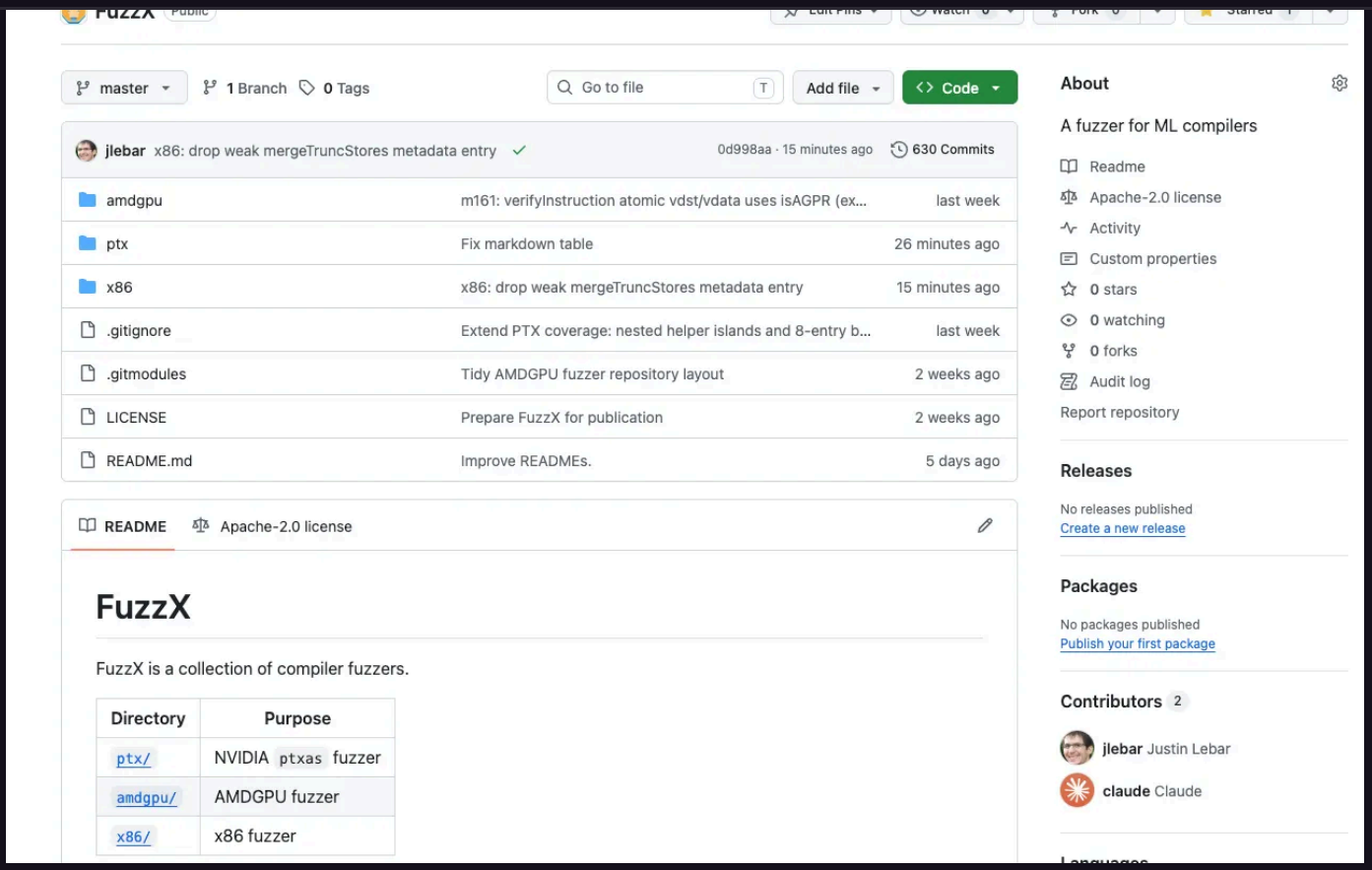
Fast forward to mid-May 2026. I joined SemiAnalysis as a contractor, and I decided to try applying the same technique to NVIDIA's low-level compiler, ptxas. I expected this to be less fruitful than fuzzing LLVM, for a few reasons.

- In general, fuzzers can get "stuck": Once they find a bug, they can keep finding new ways to trigger it. With an open-source compiler like LLVM, you can "just" fix the bug and then continue fuzzing. But with a closed-source compiler like ptxas, the best you can do is try to modify your fuzzer so it doesn't generate inputs that trigger the same bug. Implementing this is tedious at best.
- With LLVM I can run just one pass (e.g. instcombine), whereas with ptxas I have to run the whole compiler end-to-end. I worried that this would make some bugs require larger or more complicated reproducers, making them less likely to be found by a fuzzer.
- I can build LLVM myself, so I can compile with flags that add instrumentation into the program under test to help the fuzzer choose "interesting" inputs that explore new parts of the program. Although AFL++ has modes for gathering instrumentation from precompiled binaries, they slow down the program under test, and in general I didn't expect them to be as useful. (I didn't end up actually using these modes when fuzzing ptxas; I just did purely undirected fuzzing.)

I expected maybe I'd find a handful of bugs after a few weeks of work, like before.

Instead, in three days, I had 40 programs that ptxas miscompiles. (A week later, this number is up to about 80.) Although some of these test cases probably reflect the same underlying bug in the compiler, I was still kind of astonished. Many of the reproducers reduce to fairly "normal-looking" instruction sequences.

If you want to have a look at the bugs I found (really, that Codex and Claude found), they live in the FuzzX repo on GitHub.



Source: FuzzX on GitHub https://github.com/SemiAnalysisAI/FuzzX/

Why was my fuzzer so much easier to write this time? As far as I can tell, it was the difference between ChatGPT 5.2 and 5.5. This time, I vibe-coded this entire fuzzer, never looking at a line of code. The LLM did the tedious job of altering the fuzzer after every bug we found so that we wouldn't get stuck finding the same bug over and over. It also minimized each test case it generated, often spending an hour or longer doing so. It independently decided which PTX instructions to fuzz, and which sequences of instructions were "safe" (i.e. didn't trigger undefined behavior). I just put it in a loop using `/goal` and went to sleep.

To be clear, it's not surprising that fuzzing found bugs. But it *was* surprising that I found this many bugs so quickly, with almost no manual effort.

Naturally, my next question was whether I could also find bugs in LLVM's AMDGPU backend. You won't be surprised to hear that I could, at roughly the same rate as I found bugs in ptxas. At some point, my personal ChatGPT Pro account ran out of credits and I switched to SemiAnalysis's Claude account. I didn't notice a difference in quality between Opus 4.7 and ChatGPT 5.5, they were both great.

can apply their fixes immediately. If any of these bugs were critical to me, I could even fix them myself. Hooray for open-source toolchains.

At this point my ptxas and AMDGPU fuzzers started to slow down; they were running for increasingly long times without finding any new bugs. I almost stopped there, but I had a thought. What if I just asked Claude to read through LLVM and find bugs? In other words, what if I was building a fuzzer because I hadn't absorbed the Bitter Lesson?

I asked Claude to spin up 50 subagents at a time looking for bugs. Boy did I get them. Claude found bugs at a rate of *one every four minutes*. (In contrast, at this point the fuzzer was taking hours to find new bugs.)

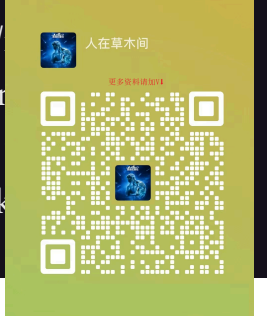
My initial reaction was, I don't even think this means LLVM's AMDGPU backend is particularly buggy. Indeed, a friend suggested I do the same to the x86 backend, and it produced bugs at a rate of almost *two per minute*.

My bug-finding agents didn't show any signs of slowing down before I stopped them. I don't know how many more issues are out there. Probably a lot?

The question with automated bug-finding usually is, "Do these bugs matter?" I've only gone through about 20% of the x86 bugs so far. The bugs found by agents are definitely less severe on average than those found by the fuzzer; every fuzz bug is a demonstrable miscompile, whereas agents are using their judgement to decide what is and isn't a bug, and sometimes they're wrong.

On the other hand, one of the 30 or so agent-discovered bugs I've examined includes this extremely frightening case where LLVM will turn an atomic store into two non-atomic stores. This bug would have been very difficult to find via fuzzing (fuzzing atomics is hard), and would likely be both quite bad and quite hard to root-cause if it happened in production (because downgrading an atomic to a non-atomic store will be fine 99% of the time, and 1% of the time will corrupt your data).

The other question you should ask is, how much did all this cost? The vibe-coded fuzzer was relatively cheap. I was using about double the weekly quota for my $200/ ChatGPT Pro account to write the AMD and NVIDIA fuzzers at the same ti expensive with pay-per-token Opus 4.7, on the order of $1000 for a few days' know if it would have been cheaper with a Claude Max account, and I don't k



On the other hand, having an army of subagents read the code was, um, not cheap. I spent more than $10,000 in a few hours, and I wasn't even using fast mode. (Thanks for the tokens, Dylan!) Even though bugs found this way were on average lower-severity than fuzz bugs, the fact that code inspection can find whole classes of bugs that would be very challenging to find via fuzzing makes it valuable to me.

Frankly, just the one atomics bug mentioned could do far more than $10,000 in damage under the right (or perhaps wrong) conditions; silent data corruption like this will break just about any production system. And even if your system had safeguards that allowed it to notice this kind of corruption, you'd still likely burn months of engineering time hunting for its source. If I were doing this again, I wouldn't bother with fuzzing, assuming I had the budget to unleash my herd of agents.

I'm still trying to make meaning out of this experience. I think it's bigger than simply "With enough subagents, all bugs are shallow." Maybe the lesson is this:

> **Things that were impossible five months ago are now "just" Very Expensive.**

A corollary is, if you *don't* have the budget, you're operating in a smaller part of the possibility space than those who do. And I expect that gap to grow substantially over the coming months.

SemiAnalysis spent an order of magnitude more on my tokens than on paying me the day I ran the agents. If the value of something is how much someone is willing to pay for it, then this was the first time in my career that I delivered less value to my employer than my AI did.

It makes me wonder how much SemiAnalysis will be willing to pay for tokens in six months, and what life will be like for people and companies who can't or won't pay.

One last thought. I could do the "have Claude read the code" approach with the AMDGPU and x86 LLVM backends because I have the source code. I can't do it with ptxas because I only have a binary. But…how sure am I that, given enough tokens, Opus 5.7 (or even 4.7) couldn't find bugs just by reading the assembly?

This post is for paid subscribers

Already a paid subscriber? Sign in

← Previous

Next →



A guest post by

Justin Lebar

© 2026 Dylan Patel · Privacy · Terms · Collection notice

Start your Substack

Get the app

Substack is the home for great culture