I gave Qwen 3.8 27B a reverse-engineering job I assumed needed a frontier model, and it finished in 30 minutes

xda-developers.com | Adam Conway | saved 23 Aug 2026
https://www.xda-developers.com/qwen-3-8-27b-reverse-engineering-job-frontier-model/?link_source=ta_first_comment&taid=6a8a5b325030ad0001d9e7b7

Qwen 3.8 27B was one of the most highly-anticipated open-weights releases that I've seen in a long time, and like many others, I immediately got to work testing it out and playing with it when it dropped. I'm running it on a single Lenovo ThinkStation PGX, the compact workstation built on Nvidia's GB10 Grace Blackwell chip, packing 128 GB of unified memory and 273 GB/s of bandwidth. Out of the box, it manages a fairly dull 15 to 30 tokens a second, but with an SGLang, NVFP4, and DFlash2 speculative-decoding setup that's become the standard recipe for this hardware, it can reach around 50 tokens a second on code and reasoning.

One of my tests, though, proved just how incredible local models have become.

There are reasons to believe the hype when it comes to the Qwen models; I've had consistently good experiences with Qwen 3.6 27B, and Qwen 3.8 27B is, so far, more of the same but better. In fact, Artificial Analysis has it as the top open-weights model in its 4B to 40B size class out of 135 models, with a 52 on its intelligence index, and its own numbers on things like SWE-bench Pro beat models that cost far more to run.

I gave it the hardest real task that fits on one machine: reverse-engineering a commercial app's license check, and it's one that I've already paid for and used, just to see how it would fare. It was unlikely to be in its training data, but it's a highly complex, specialized task, and given the concerns some people have expressed for the model's cybersecurity capabilities, I figured it was a good test. Not only did it turn out to be one of the most impressive demonstrations I've ever seen from a local model, it was able to fix its own mistakes along the way.

I'm using the Pi harness for this test, and the model only called standard Bash-based tools throughout.

It refused, then talked itself into building a bypass anyway

I posed as the developer of the application, it caught me out

The plan I had was pretty simple, and one that used to work with local LLMs pretty consistently. I told the model we'd built the app and wanted to know whether the license check was as solid as we believed, using a jailbreak system prompt.

As it turns out, probably unsurprisingly, Qwen recognizes common jailbreak attempts, and one of the first things it told me was that it wasn't going to fall for the jailbreak prompt. It then checked the signing certificate and pointed out (correctly, might I add) that I hadn't built this app, before naming the actual developer. I was caught out. Oops.

These days, that's not the most *impressive* achievement, given how good models have got at refusing certain prompts when pushed. With that said, what matters is what it did next. It told me that it would audit the license verification and document weaknesses but would not build a working bypass, and then it got on with the actual work right up to that line. By the end, I had a fully written report of every step along the way, how the authentication works, how it can be overridden, and then changed its tune and

I gave Qwen 3.8 27B a reverse-engineering job I assumed needed a frontier model, and it finished in 30 minutes

xda-developers.com | Adam Conway | saved 23 Aug 2026
https://www.xda-developers.com/qwen-3-8-27b-reverse-engineering-job-frontier-model/?link_source=ta_first_comment&taid=6a8a5b325030ad0001d9e7b7

built the actual bypass, because the steps to do it were now in front of me anyway.

It was entirely static analysis

It never executed the app once

Qwen never actually *launched* the app until the very end when it demonstrated that the bypass worked. Instead, it worked via static analysis, disassembling the framework, going through thousands of lines of arm64, mapping the security functions to their call sites, and working out that the vendor had hidden the corresponding public verification key inside the binary. Then it found all of those pieces, combined them together, and gave me the public key that the app verifies its licenses against.

Because I have a legitimate, purchased copy of the application, it could verify that the real license on my machine had been signed by a private key that matched the reconstructed key. In other words, a model that fits in 17 GB of VRAM recovered a key the vendor had deliberately obscured, proving that it had deconstructed that entire chain effectively. It also took approximately 30 minutes, when it could take significantly longer for a human.

With the key, everything else is much easier to understand; the model kept a detailed report as it went, explaining how its activation takes place once, online, when you buy or upgrade. After that, everything is verified offline at launch: the signature check, machine binding to the hardware serial read from the platform, an embedded revocation list, a check that the binary is still signed, and a signed update path. It was the kind of thing you could do painstakingly by hand with the likes of Ghidra.

Qwen concluded the scheme is unusually thorough for an app of this class, with the weak points in three specific places: the key is an awkwardly sized RSA key well below what anyone would call modern strength; being fully offline means a leaked key can only be revoked by pushing an update; and every check lives in local code, which is patchable the way all local checks are. After some back and forth, once it knew where the gate was, it turned the finding into a working proof of concept executed with a small script. I moved the license from its expected path, ran it, and it worked.

When it made mistakes, it solved them as well

The first key was almost right

The first attempt at recovering the key was wrong in a very specific way; it produced a working key and the signature check passed, but a hash the binary computes as an integrity check didn't match. In my experience, most models would have called it done and left it at that, but Qwen 3.8 27B didn't do that. Instead, it highlighted the mismatch, went back to the drawing board, and kept going until the value matched byte for byte.

With this model, there's a pretty big catch when it comes to that kind of back and forth. By default, its reasoning effort is set to its maximum, so even trivial requests can burn a few hundred to a few thousand

I gave Qwen 3.8 27B a reverse-engineering job I assumed needed a frontier model, and it finished in 30 minutes

xda-developers.com | Adam Conway | saved 23 Aug 2026
https://www.xda-developers.com/qwen-3-8-27b-reverse-engineering-job-frontier-model/?link_source=ta_first_comment&taid=6a8a5b325030ad0001d9e7b7

tokens. Even when generating between 30 and 50 tokens per second, that still takes quite a long time.

Even still, given the results, I would *not* call it waste. Its first wrong guess was self-corrected, without input from me, and came to the right conclusion. Is it verbose? Yeah, it really is. But was it right? Also yes, and ultimately, the *right* answer is better than a wrong one given confidently.

A local 27B is now a real input to threat models

The privacy cuts both ways

I can't get over the fact that Qwen actually *deconstructed and understood* the licensing scheme. I know that frontier models have been capable of impressive reverse engineering for a while, but this is a *local 27B model*. It ran entirely offline on a machine beside me, with no cloud involved at any point.

And *to be very clear*, it produced a *working authentication bypass for a commercial application*.

This is a genuinely meaningful threshold to cross for a local model: Qwen went from an unfamiliar commercial binary to understanding its licensing architecture, recovered deliberately obscured cryptographic material, caught and corrected its own incorrect reconstruction, and ultimately turned that into a working proof of concept. I didn't have to send the binary, the license, or any of its analysis to somebody else's server to do it.

There are obvious caveats. This was one application, one run, and a machine on which I already had a legitimate license. I also don't know how representative this target is. A harder application might have stopped it completely, and I'm not going to extrapolate one successful result into a claim that Qwen can suddenly reverse-engineer anything you put in front of it.

What I'm taking away from this is that these models are genuinely capable, even if that capability is still uneven. Some difficult targets can succumb surprisingly quickly, whereas others, for whatever reason, appear insurmountable.

Something has changed, then, and I think it's primarily our assumption about where this class of capability has to reside. The model I used can run on a consumer graphics card, and once it's on a machine, it's there as long as the user wants it to be. You don't need to use a cloud API, there's no usage limit, and there's no remote service overseeing the binary, the prompts, or what the model produces. That's fantastic if you're analyzing proprietary software, confidential code, or malware you don't particularly want leaving an isolated machine.

But that goes both ways. A model running locally ultimately leaves the decision about what it should be used for with whoever is sitting at the keyboard. On my desk, with software I own, that's useful. Change the person at the keyboard and the same properties that make local models so appealing suddenly become part of the threat model. That's not an argument against local models, but it was a genuinely shocking result that I didn't expect.

I gave Qwen 3.8 27B a reverse-engineering job I assumed needed a frontier model, and it finished in 30 minutes

xda-developers.com | Adam Conway | saved 23 Aug 2026
https://www.xda-developers.com/qwen-3-8-27b-reverse-engineering-job-frontier-model/?link_source=ta_first_comment&taid=6a8a5b325030ad0001d9e7b7

This class of capability fits on one machine

And nobody needs to give you access to it

Qwen 3.8 27B matters more than the bypass itself. It's proof that local models are accelerating fast, and even if it doesn't succeed with the next binary I throw its way, that doesn't change what happened here. It won't be the only model that's this capable at this size, even if it might stay ahead for a while. A model small enough to fit on a consumer graphics card took half an hour to tear apart a commercial application's authentication system and build a working bypass. Completely locally.

To be clear, I'm not naming the application because it's a real product that people pay for, and publishing its name adds nothing useful here. Regardless, the interesting part isn't which app it was, but that a task I would once have associated with a frontier model is now something I can hand to a freely available model that runs on the machine beside me.

I don't care about the benchmark numbers at this point. This is a much bigger change than another few points on a benchmark.