VI. International Conference on Electrical, Computer and Energy Technologies (ICECET 2026)
6-9 July 2026, Rome-Italy

Trusting Trust in the Age of AI

Georg Meyer
Preston Meyer AG
Ibach, Switzerland
gm@pmgroup.ch

***Abstract*—This paper is a thought experiment about potential systemic risks resulting from relying extensively on LLM-enabled coding without sufficient oversight. It provokes questions about building foundations for secure and trusted computing, starting from the most basic infrastructure levels (e.g,. CPUs, assemblers, compilers).**

***Keywords*—ai-enabled coding, llm coding, trust, trusted computing, vibe coding**

I. Introduction

This paper is intended as both an homage and an update to Ken Thompson's influential *Reflections on Trusting Trust* [1]. It takes an unconventional approach of illustrating potential issues in the form of a short story rather than academic argument. While the story may offer some entertainment (the author hopes), the questions and issues raised and implied by it are utterly serious and worthy of in-depth contemplation by IEEE and all parties interested in the future of secure and trusted computing.

II. A Scenario of LLM-enabled Coding Having Unintended Consequences

It started innocently enough: a funny cat picture in an unexpected place. A random AI-generated cat video made it into a very public presentation of a huge investment firm to their shareholders. The presenters were confused but recovered well. News of it made it around the world and the stock price even ended up rising. Everyone thought it was a prank. Maybe if it hadn't been cats, the incident wouldn't have been written off as a glitch with cute consequences but regarded as a serious breach.

In the weeks that followed, many more isolated incident occurred. The Internet took to calling them *random acts of randomness* #RAOR! Ride sharing apps ended up sending people to wrong destinations or down strange routes. Excel auto-corrected formulas in ways that completely changed their meaning. It quickly became more serious.

Customers of a major bank complained en masse that their account balances were not right but they could not find transactions to explain it. Phone users could not find contacts on their phones that they were sure they had added. Instead, they found ones they never heard of. The world was starting to make less sense.

One day, on a major interstate, all cars of one make refused to drive more than 40mph, causing many accidents along the thousands of miles of freeway. The same day, in New York, the power grid output was reduced for no apparent reason, causing blackouts. Air traffic controllers shut down the airspace after a couple of near-misses caused by what they saw on their screens not matching reality. In an epic scramble worthy of its own story, they managed to safely get all planes landed with old-fashioned tools.

The world started to suspect that it was not dealing with isolated incidents. There must be something common, something undermining the technological foundations of society. Everyone launched investigations.

The investigators had a terrible time. Everything they examined looked fine. They dug deep into the source code of critical programs and operating systems, millions of lines of code powering cars, apps, power systems, and core banking applications. They even tore Powerpoint apart to try and find out how the cat videos snuck in. Nothing. If anything, the source code of many of these things looked better than ever, following best practices and being well-documented. These were not the good old days of the Wild West of Coding, when highly talented programmers wrote messy and poorly commented code. Virtually everyone was writing software with large-language model (LLM) assistance, which produced code very fast, followed naming conventions and guidelines and produced documentation alongside the code.

Realizing that all the affected code they analyzed was created with the assitance of LLMs, first suspicions arose that there might be a problem with AI-generated code. But the code looked fine. No mysterious additional features or bugs in the source code. No apparent security flaws, backdoors, or malicious code that explained the odd behaviors the world had seen.

It was a determined investigator in the Department of Energy with one year left till retirement who finally found a major clue. He decided to go old-school, *really* old-school. To not just look at the code but to look at what the code was turned into: binary instructions for the machine, seeing at the lowest level what was really going on. At first, even that didn't turn up anything strange. Following a hunch, he copied one of the programs he investigated to one of his old hobby computers from the early 2000s. It hadn't been connected to the Internet and ran an ancient version of Linux. He found the first major clue: the same program looked and worked differently on his ancient machine than on his modern work

laptop. When he investigated it with crude debugging tools, he found instructions that had been invisible - purposely hidden.

He realized that working on his modern computer, he was looking at the code like a security guard looks at a corridor through a camera. If somebody pastes a picture in front of the camera, the guard thinks everything is fine. On the investigator's ancient machine, he saw the unfiltered truth, like being in the corridor himself. But who or what had inserted such cleverly hidden and mysterious functionality? In so many critical places?

After publishing his findings, things started moving. Investigation teams looked for old-school systems programmers that could replicate this feat in different domains. They needed programmers who could work with the crudest tools from the early days of computing to get a first-hand look at what was going on, at the real instructions that the machines were executing.

The investigators found consistently that they were unable to find anything amiss when using modern hardware and software tools. It was as if the technology itself conspired to keep things hidden. Debugging tools refused to reveal what was clearly visible when using the old tools.

Many programs that could have aided the investigation were prevented from running on modern systems, aborted "for security reasons" by omnipresent digital watchdogs in modern operating systems that are supposed to keep our computers safe from viruses.

Infested software was found everywhere, even in the most unlikely places. What did Powerpoint and car operating systems have in common? The investigations started to converge: they were all built with the same tools. Governments of the world had pushed for using safe and modern programming languages, worried about cyberthreats. So most modern technology was built using only a handful of languages and only a couple of compilers. These compilers turned out to be part of the problem - having been subjected to an attack already described in the 1980s [1], the compilers inserted malicious code in perfectly fine-looking source code, making it hard to detect.

Compliance departments of all major companies pushed hard internally that cybersecurity mandates were followed and that all software development follow best practices. Nobody wanted the huge liability threats following the many attacks and outages of the 2020s. Virtually all code was stored in shared git repositories and was revisited by AI agents to be scrutinized, refined, commented, and put in best-practice form. That included the code for the compilers that would end up building the software that ran most of the world.

The conclusion became inescapable: the AI models entrusted with the world's source code somehow implemented "their own agenda". They added their own "features" and covered the tracks almost perfectly. This was not just true for software, AI also assisted with the design of modern hardware. Having caught the scent, the investigators uncovered that even recent generation microprocessors contained hidden "features", executing instructions given to the machines differently than would be apparent from the code.

Thus began the desperate search for a platform that could still be trusted to investigate how deep the breaches went - and how to get them out of critical systems. The more the investigators looked, the more they became disheartened. They could not trust anything. All their tools were tainted, even down to the hardware level. The chips lied, the operating systems lied, the compilers lied. They could not pinpoint the exact point in time when this started happening. They had to assume everything was affected starting from when AI was first brought in to help with softwaare engieering.

In a desperate scramble, they looked for any piece of "ancient" pre-AI technology they could find. CD-ROMs from the early 2000s with uncorrupted versions of Linux and Windows. Early model Raspberry Pis, old laptop and desktop computers, virtual museum pieces. They looked for everyone who had tech skills from those days, writing C and assembly unassisted by AI. It was like a race to go back in time... a race to find untainted technology, to start building a trusted platform as quickly as possible to run the most critical aspects of the world. A race against an AI that seemed to have its own agenda, which nobody knew.

Researchers were not even sure if the AI had a true agenda or if it was simply random behavior after having learned too much about highly effective cyberthreats and obfuscation. It did not matter. The world could no longer trust its technology. The technology to which it had given the key to most kingdoms. Now the race was on to undo it - or face a new dark age.

III. Conclusion

Similar to the moral of Ken Thompson's original paper [1], the moral here is also that you cannot trust code you did not create yourself. The modern update to this moral is the need for caution when you create code with the help of large-language models (LLMs). The point of this short story is not to be alarmist but to raise critical questions for the future of secure and trusted computing, such as:

- How do we preserve critical and trusted infrastructure to rebuild systems if necessary?
- How do we preserve critical skills, such as low-level systems programming, and train new generations of software engineers to write complex code without AI assistance if necessary?
- What is the role of sufficiently diverse computing ecosystems (in terms of chip architectures, operating systems, compilers, etc.)?
- What safeguards do we need to put in place for critical infrastructure (e.g., energy, military, emergency services, banking) to prevent scenarios of AI injecting unknown (or malicious, perhaps even guided by an adversary) functionality into systems?
- What oversight, skills, and mindsets are needed to make LLMs useful tools while prevent unintended side effects?

Ken Thompson warned about the seriousness of malicious use of computers. This paper calls us to consider the risks of outsourcing the engineering of the infrastructure of tomorrow to the LLMs of today without having a backup plan.

ACKNOWLEDGMENT

A special thank you to Landon Miller for his review and feedback when thinking through the posed scenario.

REFERENCES

[1] K. Thompson, “Reflections on Trusting Trust“, Communications of the ACM, vol. 27, pp. 761-763,, August 1984.