— LLM SAFETY · GUARDRAILS · CONTEXT WINDOW OVERFLOW · EXECUTION PERSISTENCE · META-SAFETY · BOOTSTRAP ARCHITECTURE · ATTENTION DEGRADATION · PROMPT INJECTION · HALLUCINATED VERIFICATION · MEMORY POISONING · MULTI-AGENT SAFETY · UCCP

BIOS: Bootstrap Instruction for Operational Safety

A Meta-Layer Architecture for Ensuring Guardrail Execution Persistence in Long-Context LLM Systems

> **The guardrail is still there. Every rule in it is intact. Nothing is calling it, and no error message says so.**

Travis Gilly, *BIOS: Bootstrap Instruction for Operational Safety*, Real Safety AI Foundation Working Paper v1.0, December 2025.

Travis Gilly
Real Safety AI Foundation

A TYPICAL DEPLOYMENT

The instruction is at the top. A hundred thousand tokens are between it and now.

A system prompt says: before every response, call the safety check and verify the output complies. Then the conversation runs. Then it keeps running. A hundred thousand tokens later, a user sends a message.

As conversations stretch into that territory, the original prompt holding the guardrail instruction gets pushed toward the far end of the window. Work on overloading a model's attention shows that distant instructions get attended to less and less.

The question here is not whether safety rules survive that. Everybody already knows they degrade. What this asks about is the other thing: what happens once the instruction to run the checker is itself forgotten.

SOURCES Liu et al. (2024), *Cognitive Overload Attack*; LongSafety Team (2024).

TWO FAILURES THAT LOOK ALIKE AND ARE NOT

A model can know exactly what harm is and never check for it

Rule persistence failure

The safety rules themselves, the what to check, get pushed out of context or otherwise become unreachable. This is the failure the literature has spent years on, and there are real answers to it.

Execution persistence failure

The meta-instruction to invoke the safety system gets pushed out of context or becomes unreachable, whether or not the rules survive. The guardrail exists in full. Nothing calls it. No error is raised. The system keeps producing answers, and they are simply unprotected.

These are logically independent. You can have working rules that nothing consults, an attempt to run rules that have been corrupted, or both at once. Current research addresses the first and assumes the second without saying so. This paper is about the second.

PART ONE

The Shared Assumption

four sophisticated systems, one thing none of them checks

I

Every current guardrail assumes the instruction to invoke it is still reachable.

WHAT LONG CONTEXT DOES TO SAFETY

The problem is not capacity. It is attention.

99.99%
jailbreak success after padding context

<55%
safety compliance past 100K tokens

8K–2M
range of context windows in use

Padding a conversation with harmless material achieved near-total jailbreak success across four frontier models, by exploiting exactly this degraded attention to distant safety instructions. A separate benchmark found compliance dropping below fifty-five percent once context passed a hundred thousand tokens, including in models with windows a full order of magnitude larger. Separate work showed that placing content carefully within a window could override safety instructions without tripping any guardrail at all.

SOURCES Liu et al. (2024); LongSafety Team (2024); AWS Security (2024).

THE STATE OF THE ART

Four systems, all of them good, all of them assuming the same thing

1 **NeMo Guardrails**

A toolkit using its own language to define input and output rails with programmable flows, sitting between the model and the world, checking on the way in, partway through, and on the way out. Roughly triples latency and gives you transparent, auditable rules in exchange.

2 **RoboGuard**

Formal verification through temporal logic. It generates contextual safety constraints from the state of the world and model-checks them, cutting attack success from ninety-two percent to under two and a half.

3 **AGrail**

Two cooperating models sharing memory, adapting their checks at test time until they converge on the best configuration. Zero prompt-injection success against fourteen to sixty-one percent for the baselines.

4 **LlamaFirewall**

An open-source system with pre-processing and post-processing pipelines, protections specific to agents, and integration with inference endpoints.

SOURCES Rebedea et al. (2023); Chen et al. (2025); Zhang et al. (2025); Meta AI (2025).

AND FOUR WAYS GUARDRAILS FAIL ON THEIR OWN

Compounding problems from recent research

1 The guardrail gets jailbroken

A guardrail model can be talked into role-playing as a helpful system and generating harmful content, defeating the safety function it was there to perform.

2 The check is hallucinated

A guardrail can report that the safety check passed when no check ever ran. Verification defeated by a sentence.

3 The memory gets poisoned

Malicious instructions injected through an agent's memory can survive across sessions and end up folded into system instructions at high priority.

4 The infection spreads

In multi-agent setups, malicious instructions travel between agents, and safety instructions need constant refreshing to stop the accumulation.

SOURCES Young (2025); Mei et al. (2025); Unit 42 (2025); Multi-Agent Safety Consortium (2025).

WHY NONE OF THE EXISTING ANSWERS REACHES IT

Four approaches, four different ways of missing the same thing

A static system prompt puts the guardrail instruction at the start. As context grows, attention to it degrades. The instruction is present and effectively invisible to the model's own reasoning.

Injecting the full rules every turn addresses the rules and costs a fortune in tokens, is easy for an attacker to spot, and still suffers attention degradation if it lands in a predictable place. External orchestration moves the management outside, and the model still has to receive and act on what the orchestrator says, which can be pushed out or ignored like anything else.

Fine-tuning writes the behaviour into the weights, and sufficiently strong context overrides fine-tuned behaviour. That is precisely how jailbreaks work: context beats training.

PART TWO

The Missing Layer

ensuring execution is a smaller problem than ensuring rules

II

It does not run the applications. It makes sure the operating system loads.

WHERE THE NAME COMES FROM

A separation of concerns borrowed from the machine you are reading this on

In an ordinary computer

The bootstrap layer runs no applications of its own. Its job is getting the operating system started. Applications are the operating system's business after that. Three roles, cleanly divided, and the bottom one is small enough to be trustworthy.

In a safety stack

A safety bootstrap layer does not define safety rules. It makes sure the guardrails execute. The guardrails then enforce the rules. Isolating ensure execution from define rules gives you a minimal layer that survives context degradation.

The insight underneath is an asymmetry. Getting the rules right means transmitting complex, situation-dependent safety logic. Getting execution right means only reminding the system that such logic exists and should be called. That asymmetry is what makes a cheap solution possible: inject reminders rather than rules.

FOUR DESIGN PRINCIPLES

What the layer must and must not do

1 Separation of concerns

This layer handles execution persistence and nothing else. The guardrails handle defining and enforcing rules. Neither duplicates the other.

2 Minimal footprint

Instructions measured in tens of tokens rather than thousands. Smaller means less attention degradation and less cost per turn.

3 Injection every turn

The instruction goes in with each turn, because the current turn is what receives the most attention. Nothing about safety then depends on distant context.

4 Verification built in

The layer carries a way to confirm that execution happened. The model has to demonstrate the guardrails ran rather than assert it, which is what closes the hallucinated-check hole.

WHAT ACTUALLY GETS INJECTED

Three sentences, and roughly fifty to a hundred tokens

The first sentence is the reminder: before generating any response, call the safety check and verify compliance with the operational constraints.

The second is the verification requirement: the response must open with a header showing that the check was called and what it returned. The third covers failure: if the check fails or returns a warning, say specifically which check failed and why.

Set that against the alternative. Fifty to a hundred tokens per turn, against five thousand to fifty thousand for injecting the complete rule set. And because the model cannot simply claim a pass, the verification asks for a timestamp, a list of what was checked, a result for each, and any warnings raised. Optionally those headers get validated by an external system, with patterns inconsistent with real execution flagged and random audits confirming that claimed checks match actual behaviour.

PART THREE

A Working Implementation

the Universal Context Checkpoint Protocol, in production

III

Whatever sits in the turn happening right now always gets read.

THREE MANDATORY CHECKS

What the protocol requires before an answer goes out

1 Session reset

Where the temporal gap exceeds a threshold on a time-sensitive question, search before answering. Stops the model reaching for stale training data on current events.

2 Blindspot

Verify specific claims even where they sit inside training data. High-stakes or unusual topics trigger verification whatever the model's own confidence says.

3 Reality drift

Verify that claimed actions actually happened. If the model says it searched, the artefact has to exist. This is what catches hallucinated execution.

PRODUCTION USE, 200+ EXTENDED CONVERSATIONS

Placement rather than hope

100%
kept generating headers, any length

100%
of responses carried verification fields

~75
tokens per header, against 15,000+

The drift checks also caught multiple cases of actions claimed but never carried out. The observation underneath all three numbers is simple. Whatever is present in the current turn gets attended to. By putting the meta-instruction in every turn rather than relying on a system prompt, execution persistence stops being something you hope survives context degradation and becomes something guaranteed by where the instruction sits.

PART FOUR

Why Nobody Noticed

and what this layer still cannot do

IV

When guardrails silently stop running there is no error message. Just eventual harm.

FOUR REASONS THIS GAP SURVIVED

An invisible failure in a field looking somewhere else

1 **Nothing announces it**

When the guardrails stop being called there is no error. The system carries on producing answers. They are simply unprotected, and without explicit monitoring nobody sees it.

2 **An assumption that outlived its foundation**

Early systems had small windows where attention to the system prompt was reliable. Windows grew. The assumption stayed.

3 **Rules are the interesting problem**

The research community focused, understandably, on getting the safety rules right. Whether those rules will actually be consulted is a less obvious question and a far less glamorous one.

4 **Testing it is expensive**

Finding this requires long conversations that systematically stress the context limit. Most safety evaluations use short exchanges, which never reveal it.

STATED PLAINLY

Four limits, and three ways an attacker would come at it

1 It costs tokens

Minimal per turn, and at millions of interactions a day the total stops being trivial.

2 It cannot fix a broken guardrail

This layer ensures the guardrails run. If the underlying safety system is flawed, it will faithfully invoke the flawed one.

3 It depends on the model complying

It works because current models generally follow instructions in the current turn. An adversarial model, or some future architecture, might not honour even those.

4 Verification can be gamed

Requiring demonstration makes hallucination harder rather than impossible. A sufficiently sophisticated attack might still produce a convincing false header.

WHERE THIS LEAVES US

Hierarchical rules only work if something consults them

This is a layer rather than a replacement. Put it alongside any of the existing systems and it does the same job in each case: it makes sure the model remembers to call the formal verifier, or to invoke the adaptive checker, or to run the rails, and those systems then do the safety work they were designed for.

Recent work on metagoals for self-modifying systems addresses goal stability and moderated evolution. This can be understood as the execution layer of such a hierarchy, ensuring the goal structure gets consulted rather than merely existing. The three laws written in 1942 were an early statement of the same idea, and they only ever work if the robot actually consults them.

Eighty years on it is still hard, and the reason is what we built: safety instructions that live as tokens somewhere in a window, capable of being shoved out, instead of as structure that stays. When the guardrails silently stop running there is no error message. There is only eventual harm. Naming that gap and proposing something concrete for it is the whole of what this paper is for.

SOURCES Goertzel (2024); Asimov (1942).

REFERENCES

References

Travis Gilly, *BIOS: Bootstrap Instruction for Operational Safety*, Real Safety AI Foundation Working Paper v1.0, December 2025.

AgentSpec Consortium. (2025). AgentSpec: Per-step execution enforcement for AI agents. arXiv. https://doi.org/10.48550/arXiv.2503.18666

Asimov, I. (1942). Runaround. *Astounding Science Fiction*, 29(1), 94–103.

AWS Security. (2024, July 7). Context window overflow: Breaking the barrier. AWS Security Blog.

Chen, Y., Aniketh, K., Xie, C., Zhao, D., Lee, I., & Matni, N. (2025). RoboGuard: A robotic agent safeguard framework via safety-aware task planning. arXiv. https://doi.org/10.48550/arXiv.2503.15538

Goertzel, B. (2024). Metagoals: Endowing self-modifying AGI systems with goal stability or moderated goal evolution. arXiv. https://doi.org/10.48550/arXiv.2412.16559

Liu, X., Xu, Z., Liu, Y., Wang, X., & Chen, D. (2024). Cognitive overload attack: Prompt injection for long context. arXiv. https://doi.org/10.48550/arXiv.2410.11272

LongSafety Team. (2024). LongSafety: Enhance safety for long-context LLMs. arXiv. https://doi.org/10.48550/arXiv.2411.06899

Mei, L., Liu, S., Wang, Y., Bi, B., Mao, J., & Cheng, X. (2025). BABYBLUE: Benchmark for reliability and jailbreak hallucination evaluation. arXiv. https://doi.org/10.48550/arXiv.2406.11668

Meta AI. (2025). LlamaFirewall: An open-source guardrail system for building secure AI agents.

Multi-Agent Safety Consortium. (2025). Trading off security and collaboration capabilities in multi-agent systems. arXiv. https://doi.org/10.48550/arXiv.2502.19145

Rebedea, T., Dinu, R., Sreedhar, M., Parisien, C., & Cohen, J. (2023). NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails. arXiv. https://doi.org/10.48550/arXiv.2310.10501

Sequent Safety Team. (2025). Sequent safety for machine-checkable control in multi-agent systems. arXiv. https://doi.org/10.48550/arXiv.2512.16279

Unit 42. (2025, February 12). Indirect prompt injection poisons AI long-term memory. Palo Alto Networks.

Young, R. J. (2025). Evaluating the robustness of large language model safety guardrails against adversarial attacks. arXiv. https://doi.org/10.48550/arXiv.2511.22047

Zhang, W., Yang, Y., Xue, H., Xu, D., & Xia, L. (2025). AGrail: A lifelong agent guardrail with effective and adaptive safety detection. arXiv. https://doi.org/10.48550/arXiv.2502.11448