Publication status

Follow-up Research Paper II, prepared for peer review. This manuscript extends the September 2026 CHAINSTATE AGI Linux OS paper with a concrete eBPF integration architecture and repository implementation direction. Status labels are deliberately conservative: repository-integrated does not automatically mean kernel-live on every Linux distribution. Actual eBPF loading, verifier acceptance, hook attachment, signatures, privileges, performance, and enforcement require target-kernel validation.

CHAINSTATE AGI Linux OS: A Governed Cognitive Operating System with Kernel-Level eBPF Observability and Enforcement

Ciprian Pater — NWO Capital / NWO Robotics — Affiliated Researcher, University of Agder, Kristiansand, Norway.
October 2026.

Abstract

This follow-up paper extends CHAINSTATE AGI Linux OS from a governed user-space cognitive architecture toward a kernel-near observability and security substrate using extended Berkeley Packet Filter (eBPF). The first paper defined a local-first cognitive operating system in which perception, interpretation, metacognition, constitutional authorization, execution, continuity, and hardware access are separated into explicit trust planes. It introduced BranchState and TIMEMACHINE continuity, capability-scoped hardware acceleration, sensor fusion, optical inference, a hypothesis-killing/falsifiability gate, skills and plugins, desktop/voice/CLI surfaces, offline-first synchronization, auditable receipts, and defense-in-depth security controls. This paper adds an eBPF plane beneath the execution layer to observe process, syscall, filesystem, and network behavior and to provide a future Linux Security Module (LSM)-BPF enforcement surface under explicit policy.

The contribution is not to make eBPF an AI decision-maker. Instead, eBPF supplies kernel-near evidence that can be correlated with what CHAINSTATE authorized. A separate loader boundary, default-deny policy, artifact provenance, kernel-profile validation, receipt integration, and disabled-by-default LSM path are specified. The Linux BPF verifier remains an independent kernel safety boundary. The cognitive runtime is explicitly prohibited from generating and loading arbitrary eBPF programs.

The resulting system forms an auditable loop: Observe -> Represent -> Hypothesise -> Challenge -> Authorize -> Execute -> observe kernel effects -> Receipt -> Reflect. The paper consolidates the full feature set of the first architecture while adding measurable eBPF evaluation dimensions: coverage, event loss, attribution accuracy, authorization/observation consistency, unexpected-egress detection latency, enforcement latency, rollback time, verifier rejection behavior, resource overhead, and telemetry minimization.

The paper does not claim that every subsystem is production-complete. Public demo availability and repository integration are distinguished from target-kernel validation and independent security review. The research hypothesis is that kernel-near telemetry can materially improve provenance completeness, policy verification, and incident detection while preserving local sovereignty, privacy, offline operation, and the principle that cognition does not automatically become authority.

1. Introduction

The first CHAINSTATE AGI Linux OS paper argued that the operating-system layer is the most consequential integration point for cognitive systems because it controls local resources, external tools, persistence, interfaces, and device access. Conventional assistants generally sit above an operating system that was not designed around cognition. CHAINSTATE instead treats agentic computation as a governed system service.

Its central invariants are: perception is not intent; intent is not authorization; authorization is not execution; representation is not authority; local constitutional state is not rewritten by untrusted remote input; state changes are reversible where possible; provenance is retained; sensitive ambiguity fails closed; the system remains useful when remote services disappear; generated claims are falsifiable; the hypothesis-killing seat is independent from the generator; and user-local state is isolated from protected substrate metacognition.

A remaining systems problem is observability. Application logs can say what a runtime intended or believed it did, but kernel-mediated execution can diverge from those representations. A process can create children, open files, make network connections, or invoke system calls through layers that are not fully represented by an AI-level action log. eBPF provides Linux with a programmable, verifier-checked mechanism for attaching programs to selected kernel, tracing, networking, cgroup, and LSM hooks.

This paper therefore adds eBPF as a controlled observation and security layer. It is not an intelligence layer and is not a replacement for the constitutional policy engine.

2. Relationship to the First Paper

The first paper described eight primary planes: Interface, Skill, Cognitive, Metacognitive, Constitutional, Execution, Continuity, and Hardware. It also described BranchState/TIMEMACHINE, hardware abstraction, sensor fusion, optical and quantum/edge extensions, skills/plugins, desktop/voice/CLI, receipts, safety gates, offline-first synchronization, security controls, deployment profiles, evaluation methodology, and a staged roadmap.

The follow-up preserves that architecture and adds an implementation-level eBPF substrate intersecting Execution, Continuity, Hardware, and Constitutional planes. New components are: process telemetry, syscall telemetry, filesystem telemetry, network telemetry, a dedicated eBPF loader boundary, default-deny BPF policy, an LSM-BPF path reserved for future enforcement, audit/receipt correlation, and eBPF-specific testing.

The first paper's conservative LIVE/BETA/DESIGN/ROADMAP convention is retained. The existence of source code or a public interface is not treated as proof of production validation.

3. Architecture

The resulting trust chain is:

CHAINSTATE cognitive runtime -> metacognitive and kill-gate evaluation -> constitutional policy -> capability-scoped action broker -> sandboxed execution -> eBPF observation / explicitly authorized enforcement -> Linux kernel.

No plane is allowed to bypass the policy or execution boundary. An eBPF event is evidence of a kernel-visible event; it is not itself an authorization token. Conversely, an AI proposal cannot load arbitrary eBPF code.

The eBPF implementation is organized below os/ebpf, separated from the cognitive Python/runtime layer. Monitoring programs include process_monitor.bpf.c, syscall_monitor.bpf.c, network_monitor.bpf.c, and filesystem_monitor.bpf.c. security_lsm.bpf.c is reserved for a future LSM enforcement profile and is disabled by default.

The user-space loader is a privileged boundary. A production loader should require an approved artifact identity, signature/provenance, compatible kernel profile, explicit policy authorization, appropriate Linux capabilities, and an audit receipt. It must fail closed if any required trust input is missing.

4. Why libbpf and CO-RE

Linux documents libbpf as the C-based library that manages the BPF application lifecycle: opening objects, loading/verifying them, attaching them to kernel hooks, accessing maps or ring buffers, and tearing them down. libbpf also supports CO-RE, which uses BTF type information to relocate BPF programs for supported kernel versions.

CHAINSTATE therefore favors pre-built, signed BPF objects and kernel-profile validation over runtime source compilation generated by the AI. The build/deployment pipeline should generate BPF artifacts from a known source revision, record compiler and kernel-profile metadata, produce an SBOM, sign the release artifact, and verify the artifact before loading.

CO-RE does not eliminate compatibility testing. It reduces source-level dependence on one kernel layout, but program types, helpers, hooks, capabilities, configuration, verifier behavior, performance, and security policy still vary. The OS must expose an explicit degraded state when eBPF is unavailable rather than silently pretending that kernel observation is active.

5. eBPF Security Model

The proposed load sequence has eight gates.

1. Provenance: identify source revision, build metadata, SBOM, artifact digest and signer. 2. Policy: decide whether the program class and hook are allowed for the deployment profile. 3. Compatibility: verify kernel/BTF availability, program type, helper and capability requirements. 4. Integrity/signature: validate the release artifact. 5. Kernel verification: rely on the Linux BPF verifier. 6. Attachment authorization: restrict attachment to approved hooks. 7. Runtime health: monitor load state, event loss, CPU/memory overhead and unexpected detach. 8. Rollback: remove the program and restore the known-good profile if thresholds are exceeded.

The Linux verifier remains independent. CHAINSTATE cannot replace it with a model-generated safety judgment. Linux documentation also describes signed BPF objects as an authenticity/integrity mechanism whose verdict can be consumed by security policy; signing does not replace verifier checks or capability requirements.

LSM-BPF is intentionally disabled by default. Linux supports BPF programs attached to LSM hooks for system-wide MAC and audit policies. CHAINSTATE should enable such policies only after a separate threat model, compatibility matrix, recovery procedure, and independent security review.

6. From Intended Action to Observed Kernel Effect

The new security primitive is correlation.

Declared behavior = what CHAINSTATE authorized. Observed behavior = what the kernel-visible execution path reports. Receipt outcome = the auditable relationship between the two.

For a file operation, the action broker may issue a scoped capability such as fs.read:workspace. The process executes. eBPF records the relevant process/file event. The receipt layer correlates request hash, principal, capability set, executable identity, observed event class, policy fingerprint, and outcome.

For network access, a skill may be configured with net.none. If a socket event appears, the discrepancy becomes an incident candidate. It is not automatically proof of maliciousness; the metacognitive layer can represent competing explanations and the policy engine decides what response is permitted.

For process creation, the system can correlate authorized execution with parent/child identity and executable provenance. For syscall telemetry, selected event classes can expose unexpected behavior from a sandboxed workload.

The architecture intentionally does not claim complete visibility. Kernel configuration, unsupported hooks, privileged compromise, driver behavior, hardware compromise and event loss remain limitations. Coverage must therefore be measured and surfaced.

7. Metacognition and the Hypothesis-Killing Gate

The first paper separates generation from evaluation: a model cannot validate its own output. The same discipline is applied to security interpretation.

A security hypothesis should include evidence, assumptions, alternatives and a falsification condition. For example, an unexpected network connection may be hypothesized to belong to an approved updater. Evidence includes process identity, executable digest, session identity, capability set and destination class. Alternative explanations include plugin escape, dependency compromise, injected subprocess or stale process state. A contradiction between the update manifest and observed identity can falsify the hypothesis.

Thus eBPF is not an oracle. It is a high-value evidence source. The metacognitive/kill-gate layer challenges interpretations; the constitutional plane remains authoritative for sensitive actions; receipts preserve the evidence chain.

8. Integrated Feature Set

The follow-up retains the complete feature model of the first paper.

Local-first/offline-first operation: core desktop, CLI, local models, skills, receipts, branches and device adapters continue without network access.

Cognitive runtime: model adapters, symbolic state, retrieval, planning and tools produce bounded proposals.

Metacognition: confidence, contradiction checking, uncertainty, alternative hypotheses and hypothesis killing are isolated from ordinary plugins.

BranchState/TIMEMACHINE: workspaces can be forked, simulated, compared, reviewed, merged and reconstructed subject to recorded-artifact fidelity.

Hardware abstraction: CPU, GPU, NPU, FPGA, RISC-V, optional UCI-1 and future optical/photonic backends are capability-scoped rather than raw-authority devices.

Sensor fusion: LiDAR, radar, EO/IR, IMU, hyperspectral, microscopy, audio and related modalities enter through provenance and uncertainty boundaries. Sensor agreement raises confidence but does not create authority; disagreement is preserved.

Optical/quantum/edge interfaces: specialized inference backends remain optional and have CPU/GPU fallbacks; roadmap hardware is not represented as universally available.

Skills/plugins: signed or developer-installed skills receive capability tokens scoped by workspace, time, resource quota and operation class.

Desktop/voice/CLI/API: all interfaces submit typed requests to the same governed runtime. The GUI is not a privileged bypass.

Receipts/provenance: meaningful observations, decisions, tool calls, state transitions and authorization outcomes are recorded.

Constitutional safety: network egress, file writes, code execution, package installation, external messaging, physical actuation and policy changes have separate controls. Physical actuation remains default-deny and separately reviewed.

Offline synchronization: peers exchange signed manifests and content-addressed objects without becoming constitutional authorities.

Security: secure boot where supported, encryption, least-privilege services, namespaces, seccomp, AppArmor/SELinux, IOMMU/DMA controls, signed packages, reproducible-build metadata, SBOMs, secret isolation, egress policies, parser isolation, rate limits and circuit breakers.

eBPF: kernel-near process, syscall, filesystem and network observability, with an LSM enforcement path reserved for explicit reviewed deployments.

Optional chain anchoring: external timestamps and integrity references can complement local receipts without making blockchain connectivity a prerequisite for local operation.

9. Repository and Public Deployment

The GitHub repository is the engineering source of truth. The backend contains entitlement/payment authorization interfaces; the os/ tree contains the governed runtime and eBPF layer; documentation and release scripts describe verification and deployment.

The public static Hugging Face Space is the human-facing demonstration and access surface. It should not be treated as the security root for payments or installation. If payment-gated distribution is enabled, the backend must verify the transaction server-side, bind the entitlement to the paying wallet and bundle digest, and verify any same-wallet installation signature. Private keys and seed phrases must never be requested.

The public Space is useful for showing the live visual interface and feature surface, while the repository and signed releases provide the reproducible engineering surface.

10. Evaluation Methodology

The first paper proposed tests for correctness, security, safety, kill-gate calibration, performance, reliability, offline operation, hardware fallback, human factors, reproducibility and adversarial resilience. This follow-up adds:

eBPF coverage: policy-relevant execution classes with observable kernel events divided by the declared test set.

Event loss: dropped BPF events divided by generated events under controlled load.

Attribution accuracy: events correctly mapped to principal, session, capability set, process identity and artifact identity.

Authorization/observation consistency: authorized operations whose observed kernel events match the capability contract.

Unexpected-egress detection latency: time from the first unexpected network event to an incident receipt.

Enforcement latency: time from a policy decision to a kernel-level denial in an enforcement-enabled profile.

Rollback time: time from unsafe/incompatible behavior detection to known-good restoration.

Verifier rejection rate: negative-test artifacts rejected as expected.

Telemetry overhead: CPU, memory, latency and power overhead.

Privacy minimization: proportion of collected fields demonstrably necessary for the declared security purpose.

Offline determinism: ability to record and correlate security events without DNS, cloud APIs or blockchain access.

No numeric target should be presented as a measured result until a reproducible benchmark has actually been executed. The first paper's illustrative safety scores remain targets, not empirical measurements.

11. Threat Model

The expanded threat model covers malicious prompts/files, compromised plugins, hostile peers, credential theft, supply-chain compromise, device DMA, misleading synthetic observations, executable Unicode, kernel/driver vulnerabilities, malicious BPF objects, loader compromise, telemetry evasion, kernel-profile mismatch, BTF/CO-RE

incompatibility and event-buffer loss.

Controls include input labeling, sandboxing, capability tokens, egress policy, signed BPF artifacts, approved hashes, kernel verifier checks, restricted loading, BTF/profile preflight, event-loss reporting, immutable audit records, policy versioning and recovery procedures.

Root or privileged local compromise remains a residual risk. The first paper correctly treats this as a fundamental limitation. Stronger assurance therefore requires secure/measured boot, controlled administrator access, recovery media, hardware-rooted keys or attestation where available, and external verification.

12. Reproducibility Protocol

A reviewer should be able to clone the repository, verify release hashes, inspect os/ebpf, build on a supported Linux kernel with clang/LLVM, bpftool and libbpf development files, and run the test suite in a disposable VM.

The reproducibility package should publish: kernel version/configuration, BTF availability, compiler versions, BPF object hashes, loader version, policy version, SBOM, build provenance, test data, ground-truth traces and event/receipt comparisons.

Negative tests should include unsigned/corrupt objects, unsupported kernels, missing BTF, missing capabilities, wrong hook types, policy-denied loads, event-buffer pressure, loader restart and offline operation.

Production LSM enforcement should be tested only in a disposable environment with a known recovery path until independent review is complete.

13. Discussion

The contribution is not simply “using eBPF.” eBPF is an established Linux facility. The systems contribution is the explicit placement of kernel-near evidence inside a governed cognitive OS architecture.

Policy declares what should happen. The execution layer attempts an authorized operation. eBPF observes selected kernel effects. The receipt system correlates intent, capability, process identity and observed behavior. Metacognition challenges interpretations. The constitutional plane decides sensitive responses. This creates a measurable bridge between AI-level intent and operating-system reality.

The principal design risk is over-collection. Kernel telemetry can become a surveillance mechanism if unconstrained. CHAINSTATE therefore requires minimized fields, configurable retention, local-only operation, explicit egress, and visibility into degraded coverage.

The second risk is over-enforcement. An LSM policy can break legitimate administration or recovery. That is why the baseline profile is monitor-only and why LSM enforcement is a separately reviewed capability.

14. Limitations and Open Questions

eBPF coverage depends on kernel hooks and configuration. Verifier acceptance does not prove semantic correctness. Signatures establish provenance/integrity but do not prove policy correctness. LSM enforcement can create availability and recovery risks. Kernel/driver vulnerabilities remain outside the BPF program. Telemetry can expose sensitive behavior. The privileged loader becomes critical infrastructure. CO-RE improves portability but does not remove kernel compatibility testing.

The first paper’s advanced features — BranchState reconstruction, optical inference, RISC-V/UCI-1 hardware, sensor fusion, quantum/edge interfaces and robotics — retain their individual maturity labels. This follow-up integrates their safety and observability interfaces; it does not silently convert roadmap hardware into production hardware.

Open research includes formal policy-to-observation reconciliation, privacy budgets for telemetry, signed-BPF supply-chain design, loader attestation, kernel compatibility guarantees, event-loss bounds, recovery from bad policies, and formal treatment of eBPF evidence inside the falsifiability gate.

15. Conclusion

CHAINSTATE AGI Linux OS can now be framed as a governed cognitive system with a kernel-near evidence layer. The resulting loop is:

Observe -> Represent -> Hypothesise -> Challenge -> Authorize -> Execute -> observe kernel effects -> Receipt -> Reflect.

This extends the first paper’s core commitment that cognition must not automatically become authority. eBPF does not become the cognitive decision-maker; it provides evidence about selected execution behavior. The Linux verifier remains an independent technical boundary. Constitutional policy remains the authority for sensitive actions. Receipts

preserve provenance. BranchState can preserve state transitions for later review.

The public demo and repository provide an inspectable engineering surface. The next scientific step is empirical validation: measure coverage, event loss, attribution, overhead, detection latency, rollback, privacy and failure behavior across supported kernels. Only those measurements should promote implementation claims from repository-integrated or beta status to validated production status.

Appendix A — Reviewer Experiment Matrix

A1: baseline workload with eBPF disabled; compare user-space logs with ground truth. A2: monitoring enabled; measure attribution and overhead. A3: network-disabled skill attempts controlled outbound connection; verify incident receipt. A4: unsigned/corrupt BPF object; verify rejection. A5: unsupported kernel/BTF profile; verify safe degraded state. A6: event-buffer pressure; measure and report loss. A7: loader restart; verify recovery without policy downgrade. A8: offline boot; verify receipts and safety decisions without external services. A9: disposable LSM deny policy; verify recovery. A10: x86_64/ARM64 performance; RISC-V when a supported profile is available. A11: correlate eBPF events with BranchState receipts. A12: adversarial plugin attempts to obtain BPF loading authority; verify denial.

Appendix B — Public Links

First paper/preprint: ResearchGate, “CHAINSTATE AGI Linux OS,” publication ID 414490214. Public static Space: https://cpater-chainstate-os.static.hf.space/ GitHub source: https://github.com/RedCiprianPater/chainstate-os The public Space is the human-facing demonstration/distribution surface; the repository is the engineering source of truth.

References

[1] Pater, C. “CHAINSTATE AGI Linux OS: A Local-First Cognitive Operating System with Temporal BranchState Continuity, Hardware-Agnostic Cognitive Acceleration, Sensor-Fusion, Optical-Generative Extensions, Constitutional Safety, and Offline-First Distributed Deployment.” September 2026. ResearchGate publication ID 414490214.

[2] Linux Kernel Documentation. “libbpf Overview.” https://docs.kernel.org/bpf/libbpf/libbpf_overview.html [3] Linux Kernel Documentation. “BPF Documentation.” https://www.kernel.org/doc/html/latest/bpf/ [4] Linux Kernel Documentation. “eBPF verifier.” https://docs.kernel.org/bpf/verifier.html [5] Linux Kernel Documentation. “LSM BPF Programs.” https://docs.kernel.org/bpf/prog_lsm.html [6] Linux Kernel Documentation. “BPF signing.” https://docs.kernel.org/bpf/signing.html [7] Linux Kernel Documentation. “Program Types and ELF Sections.” https://www.kernel.org/doc/html/latest/bpf/libbpf/program_types.html [8] Author-supplied CHAINSTATE research series, including BranchState, TIMEMACHINE, UCI-1, RISC-V SoC, Hyperspectral Sensory Synthesis, Robotics AGI, Edge QSIM and related papers.