DOI: doi.org/10.47363/JAICC/ICAIC2025/2025(4)30

International Conference on
Artificial Intelligence and Cybersecurity (ICAIC 2025)

Conference Proceedings November 27-28, 2025 (Virtual)

Securing AI-Generated Code with Runtime Verification

Dr. Zakaria Najm

Nestria AI, Singapore

Abstract

Securing AI-Generated Code with Runtime Verification AI-driven code assistants (e.g., GitHub Copilot, ChatGPT) accelerate development but also surface security and compliance risks: generated code can replicate insecure idioms from training corpora, display unconventional control/data flows, and evade assumptions baked into static analysis. We propose formal runtime verification (RV) as a dedicated security layer for such untrusted code. The approach monitors executions against declarative specifications covering safety (nothing bad happens?), liveness (?something good eventually happens?), and hyperproperties that capture relational behaviors (e.g., information flow and cross-run consistency). We introduce a compact property language with parameterized temporal operators and a fragment of HyperLTL suited to on-the-fly checking, and a stream-based monitoring architecture that ingests typed events produced by compile-time instrumentation and selective dynamic hooks. We implement Copilot-RV, extending the Copilot framework with security predicates (taint carriers, privilege boundaries, resource budgets), sanitizer oracles, and trace-windowed evaluation to bound overhead. Contracts express API usage (memory safety, file/network boundaries, TLS certificate pinning and key-usage constraints), capability invariants, and declassification rules for logs and telemetry. Monitors are compiled to C and linked with the target, emitting verdicts and mitigation actions (terminate, isolate, roll back, quarantine outputs) immediately upon detection of a bad prefix. Our evaluation spans 1,247 LLM-generated C programs drawn from parsing, networking, serialization, and embedded-style routines. Vulnerabilities are seeded via 18 CWE families and adversarial prompt patterns. Runtime monitors detect 94.3% of injected flaws with 3.7% median runtime overhead and 2.1% false positives; median detection latency is sub-millisecond for memory-safety and capability-violation guards. An ablation comparing naive dynamic sanitizers to RV semantics shows a +24% relative recall gain at similar overheads. A hybrid static?dynamic pipeline?where RV feedback guides targeted fuzzing and patch suggestions?reduces manual verification effort by 73% relative to static analysis alone. We prove soundness and completeness for safety/co-safety classes, with latency bounded by specification lookahead and resource usage linear in active predicates and window size. Integration with DevSecOps is straightforward: specifications are versioned with code; policy packs are derived from prompts/diffs; monitors gate merges in CI and run in production with budgeted sampling on hot paths. While RV remains coverage-dependent and some hyperproperties require approximations, the results indicate that formal runtime monitors provide a practical, mathematically grounded backstop for AI-generated code, narrowing the gap between rapid AI-assisted development and the reliability demands of safety- and mission-critical software.