PACE: Policy Adaptation for Cybersecurity Events through Formal Verification of LLM-Generated Firewall Policies

Mahya Samdaliri
Computer Science Department
New Jersey Institute of Technology
Newark, New Jersey, USA
ms3539@njit.edu

Zhihao Yao
Computer Science Department
New Jersey Institute of Technology
Newark, New Jersey, USA
zhihao.yao@njit.edu

***Abstract*—The increasing integration of Artificial Intelligence (AI) into cybersecurity reduces manual efforts and improves adaptability, but the probabilistic nature of AI renders its output unreliable, especially when used in network firewall policies. This uncertainty prevents direct, real-time enforcement, leading to critical windows of latency due to necessary human oversight. This paper presents PACE, a formal framework designed to constrain and adapt LLM-generated firewall policies for safe enforcement via the extended Berkeley Packet Filter (eBPF). PACE integrates an LLM-based policy generator with a formal verifier that rigorously checks each candidate rule against a semantic bounding constraint (Φ) and a set of mathematically defined safety invariants ($\mathcal{I}$). The policies that fail verification are used to provide iterative feedback to refine future LLM prompts, automatically improving policy generation over time. We formally model this verification, so that safety properties are preserved even under dynamic rule composition and updates, providing strong theoretical guarantees. A prototype implementation of the verifier demonstrates the feasibility and efficiency of the proposed approach. It achieves high verification success rates and near-linear scalability. By establishing a proof-backed interface between probabilistic AI predictions and deterministic system security, PACE aims to reduce the time-to-mitigation (TTM) while ensuring provable enforcement correctness.**

***Index Terms*—Large Language Model, Artificial Intelligence, eBPF, Firewall, Formal Verification**

1. Introduction

The effectiveness of computer system security hinges on the speed of defense deployment. Standard Linux firewall mechanisms, such as `iptables`, the legacy interface for configuring network traffic rules, and `nftables`, its modern and more flexible successor, operate using explicit policy configurations. When a new intrusion occurs, mitigation requires human intervention for analysis, policy crafting and deployment [1]. This sequence of actions introduces latency measured in minutes or hours, providing adversaries with a critical window to operate.

Although Artificial Intelligence(AI)-driven tools have been proposed for intrusion detection and the automated generation of security policies [2], [3], [4], these systems typically function at certain confidence levels, resulting in detections that are not accurate enough for direct enforcement. For example, Mohale et al. [3] report that when the XGBoost and CatBoost algorithms are applied to network traffic data for intrusion detection, they achieved a high accuracy of 87%. Similarly, Shah et al. [5] found that the best result achieved by a hybrid Machine Learning (ML) model combining Support Vector Machine (SVM) and fuzzy logic has a false positive rate of 8.6% and a false negative rate of 2.2%. Despite the fact that AI-based solutions have significantly improved detection accuracy compared to traditional signature-based methods, which have approximately 50% - 90% false positive rates [6], the remaining inaccuracies are still too high for direct policy enforcement.

Indeed, Google Cloud Architecture Center recommends "keeping humans in the loop for critical decisions" when using AI for security tasks [7]. Microsoft Azure Web Application Firewall and Microsoft Sentinel also emphasize the importance of human oversight in AI-driven security systems to ensure accuracy and reliability, as users are "bound to get some false positives that need handling" [8], [9]. As more advanced AI models, such as Large Language Models (LLMs), are being integrated into cybersecurity workflows [10], along with low-latency kernel-level enforcement mechanisms (i.e., the extended Berkeley Packet Filter, eBPF) [11], [12], the challenge of balancing automation with accuracy becomes even more important.

To address this challenge, we introduce PACE, a formal framework that constrains and adapts LLM-generated firewall policies for direct enforcement via eBPF. By coupling formal reasoning with LLM semantics, PACE aims to reduce the TTM while maintaining enforcement effectiveness. Moreover, it introduces a proof-backed interface between probabilistic AI predictions and the deterministic invariants of system security. We formally analyze the correctness of policy verification and show that safety properties are preserved under rule composition and dynamic updates.

In summary, this paper makes the following contributions:

© 2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. Digital Object Identifier (DOI): 10.1109/BigData66926.2025.11401967

1) We propose PACE, a formal framework enabling safe deployment of LLM-generated firewall policies via eBPF.
2) We define a formal specification language and invariant set that captures correctness properties for enforcement.
3) We prove that safety is preserved under verified policy composition, providing theoretical guarantees for dynamic updates.
4) We discuss implementation considerations for PACE to achieve near-real-time enforcement latency with provable safety.

2. Background

2.1. Host Firewall Systems

Host-based firewalls enforce network security policies directly on end-user devices. These firewalls filter incoming and outgoing packets based on predefined rules, in terms of Internet Protocol (IP) addresses, ports, and protocols. Linux firewalls, primarily implemented via `iptables` [13] and its successor `nftables` [14], both utilize the Linux kernel mode `Netfilter` framework for packet filtering and manipulation through ordered lists of rules (a.k.a. chains).

Firewall policy management can be formalized as a decision function

$$f : \mathcal{P} \times \mathcal{T} \to \{0, 1\},$$

where $\mathcal{P}$ is the policy rule set, $\mathcal{T}$ is the set of all possible traffic tuples $\langle s, d, p, t\rangle$ (source, destination, port, protocol), and $f(\mathcal{P}, \mathcal{T}) = 1$ denotes that traffic is allowed.

In conventional systems such as `iptables` and `nftables`, the rule set $\mathcal{P}$ is static and deterministic. Modifications to $\mathcal{P}$ require human verification and deployment (i.e., policy update function g):

$$\mathcal{P}_{t+1} = g(\mathcal{P}_t, \Delta_h),$$

where Δ_h represents human-generated changes, often following detection events. This human dependency contributes to an overall *time-to-mitigation* delay:

$$\mathrm{TTM} = T_{\mathrm{detect}} + T_{\mathrm{analyze}} + T_{\mathrm{deploy}}.$$

As we discuss in Section 1, AI-based intrusion detection systems attempt to minimize T_{detect} and to some degree, T_{analyze}, but T_{deploy} remains dominated by manual steps.

2.2. eBPF for High-Performance In-Kernel Enforcement

Extended Berkeley Packet Filter (eBPF) allows sandboxed execution of custom programs supplied by user-space applications inside the kernel. eBPF programs are compiled into a restricted bytecode format and verified for safety properties (e.g., bounded loops, memory safety) before being Just-In-Time (JIT) compiled to native code and attached to various hook points in the kernel [15]. Existing work has demonstrated eBPF's usefulness for high-performance packet filtering and firewall [16], [11]. eBPF's key advantages for firewall enforcement include:

1) **Kernel Hook Points:** Programs can be attached to high-speed ingress/egress points (e.g., XDP/TC) and network-related system calls (`sys_connect`), providing low-level context.
2) **Atomic Map Updates:** `BPF_MAP` structures provide shared storage between the kernel eBPF program and the user-space application. Policy rules (e.g., blocking an IP address) can be enforced by the eBPF program via fast $\mathcal{O}(1)$ map lookups, and the ruleset can be updated atomically by the user-space application without reloading the eBPF program.

2.3. LLMs as Security Policy Compilers

LLMs perform well in translating natural language descriptions and logical inferences into structured data formats [17]. Recent work has explored LLMs for generating security policies from high-level descriptions [18], [19].

Formally, we can define an LLM-based policy compiler as a function:

$$\mathcal{C} : \mathcal{S} \to \mathcal{R},$$

where $\mathcal{S}$ is the set of high-level security specifications (e.g., "block all SSH traffic from untrusted networks"), and $\mathcal{R}$ is the set of low-level firewall rules compatible with a given enforcement engine (e.g., `iptables`, `nftables`, or eBPF programs). The goal is to ensure that:

$$\forall s \in \mathcal{S},\ [\![\mathcal{C}(s)]\!] \models s,$$

where $[\![\cdot]\!]$ denotes the semantic interpretation of the generated rules, and $\models$ denotes satisfaction of the high-level specification. We use the same notations throughout the paper.

Recent empirical studies have explored this translation in practice. For example, Neupane et al. [19] propose NetPrompt, an LLM-driven tool that transforms natural language security intents into software-defined networking (SDN) policies. Their evaluation shows that zero-shot LLMs can achieve up to 60% execution validity, and with few-shot prompting techniques, this can be improved to 92%.

Real-World Example. Consider a cloud administrator who needs to enforce the following security requirement: *"All incoming traffic from IP addresses outside the corporate subnet must be blocked except for HTTPS (port 443) traffic to the web server."*

Traditionally, an administrator would translate this into a set of `iptables` commands with correct ordering and exception handling. In contrast, an LLM handles parsing, logical inference, and rule ordering, significantly reducing the manual effort and potential for human error. Using an LLM as policy compiler $\mathcal{C}$, the textual specification can be automatically converted into the corresponding ruleset:

```
iptables -A INPUT -s 0.0.0.0/0 -p tcp
    --dport 443 -j ACCEPT
```

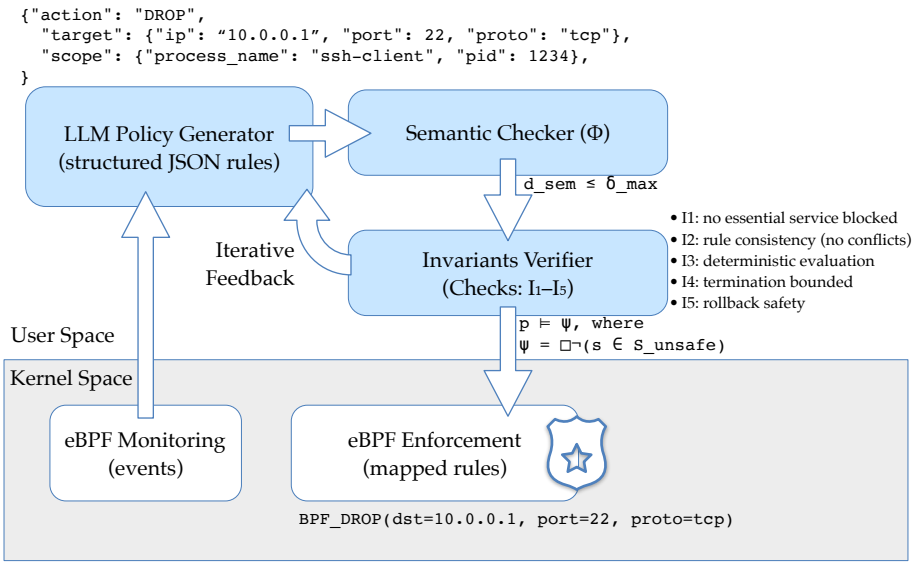

Figure 1: Overview of PACE architecture and its guarantees.

```
iptables -A INPUT -s 192.168.0.0/16 -j ACCEPT
iptables -A INPUT -j DROP
```

Theoretical Considerations. Despite their practical utility, LLM-generated policies are probabilistic. Let $\pi : \mathcal{R} \to [0, 1]$ denote the confidence assigned by the LLM to each generated rule. A rule $r \in \mathcal{R}$ with low confidence may violate invariants or produce unintended side effects (e.g., blocking legitimate network services). Formally, one must consider the probabilistic satisfaction of a specification:

$$\Pr[[\![r]\!] \models s] \geq \tau,$$

where τ is a confidence threshold selected to balance safety and enforcement agility. Because in practice, LLMs cannot properly assign confidence scores that reflect true correctness to its outputs, τ is used for modeling and not directly applicable. This observation motivates the integration of formal verification as a necessary step in deploying LLM-generated rules. By treating LLMs as security policy compilers in a formal framework, we can leverage their generative capabilities while constraining output through formal methods to ensure safety and correctness.

2.4. Threat Model

In this work, we consider the security implications of automatically generating and enforcing firewall policies using LLMs. Our threat model defines several types of adversaries and their capabilities that we aim to defend against, as listed below:

1) **Network Access:** The adversary can send arbitrary network traffic to the target host or its network, potentially exploiting open ports, misconfigurations, or unpatched services.
2) **Evasion Attempts:** The adversary may attempt to evade detection by generating traffic patterns not seen during training, using protocol obfuscation [20], or scanning the network for firewall weaknesses.
3) **Indirect Influence on Policies:** The adversary may attempt to craft inputs that cause the LLM to generate overly permissive or incorrect firewall rules, exploiting the probabilistic nature of LLM generation.

We do not consider an adversary capable of directly modifying the kernel or bypassing the eBPF enforcement mechanisms, as these are assumed to be protected by standard operating system integrity controls.

Furthermore, PACE aims to mitigate the risks associated with misclassified or imprecise policies generated by LLMs. Such unintended outcomes can either block legitimate network activity or permit unauthorized connections, breaking the intended enforcement objectives. If the firewall rules deviate from their intended semantics, adversaries may exploit these weaknesses to bypass defenses or disrupt services. Beyond these direct risks, automated policy generation introduces additional challenges, such as:

1) **Policy Safety:** No deployed policy should violate critical system invariants, such as allowing unauthorized access, blocking essential services, or causing inconsistent rule evaluation.
2) **Deterministic Enforcement:** Each packet traversing

the firewall should be subject to a deterministic, verified decision, preventing ambiguity and undefined behavior.

3. Design

3.1. Design Overview

PACE is structured as a three-layer architecture operating in a continuous feedback loop, consisting eBPF-based monitoring, LLM-based decision-making, and eBPF enforcement. The core components, a formally constrained LLM policy generator and a symbolic verifier, mediate between the LLM-based decision-making and the eBPF enforcement components. As shown in Figure 1, eBPF monitoring emits event information; the LLM generates a candidate policy p (structured JSON, e.g. `action=DROP, target=10.0.0.1:22, ...`; and then, Φ performs semantic bounding and plausibility checks (Section 3.4) to produce semantically-valid candidates; and finally, $V(I)$ checks invariants $I = \{I_1, \ldots, I_5\}$ (I_1: no essential service blocked; I_2: consistency/no conflicts; I_3: deterministic evaluation; I_4: termination bounded; I_5: rollback safety), as detailed in Section 4.1. Only verified policies p are committed to kernel maps and enforced by eBPF. Failed policies and violation information are used to refine future LLM prompts, creating a feedback loop that automatically improves policy generation to be context-consistent and invariant-compliant over time.

3.2. eBPF Monitoring Layer

For observability, lightweight eBPF programs are loaded into the kernel via a userspace agent using `libbpf`. These programs attach to system call tracepoints (e.g., `sys_enter_connect`) and network hooks (XDP/TC) to achieve realtime observability. This layer has three core responsibilities.

First, it monitors outbound connections, recording structured tuples $\langle \text{PID}, \text{Process Name}, \text{Destination IP}, \text{Port} \rangle$, which allow downstream layers to contextualize anomalous behavior. Second, it performs fast filtering by checking incoming and outgoing packets against a map (`DENY_LIST_MAP`), providing immediate `BPF_DROP` verdicts to halt disallowed traffic without delay. Third, it generates structured event reports upon detection of suspicious behavior, such as excessive connection attempts, scanning activity, or communication with known malicious IP ranges. These events are pushed through a `BPF_PERF_BUFFER` (per-CPU circular buffers) for consumption by the user-space firewall application.

3.3. Decision Layer with LLM and Verifier

The user-space firewall application consumes events from the buffer and constructs a detailed prompt for the LLM. This prompt integrates the raw event data, current policies, and system-wide context like the current system state, threat intelligence feeds, and network history. The LLM is asked to output structured JSON policies. An example prompt is shown in the listing in Section 3.3.

Firewall Policy Generator Prompt

```
You are a network security policy
    generator.
Your task is to generate firewall
    rules in JSON format for immediate
    enforcement.
Only block the minimal necessary
    IP/Port to mitigate the threat.

Event Context:
- Process: <process_name>
- PID: <pid>
- Destination IP: <dest_ip>
- Destination Port: <dest_port>
- Observed Behavior:
    <behavior_description>

Existing Policies:
- <list_of_current_policies>

System Context:
- <History of recent connections>

Output Format (JSON):
{
  "policy_id": "<unique_id>",
  "action": "<ALLOW|DROP|REJECT>",
  "target": {"ip": "<ip_address>",
      "port": "<port_number>"},
  "justification": "<rationale>"
}
```

Next, the verifier checks the generated policy against a set of formal safety invariants (detailed in Section 3.4 and Table 1). If the policy satisfies all invariants, it is passed to the enforcement layer; otherwise, it is discarded, and a new prompt is generated with additional information on the constraint violations. Once the policy passes verification, the agent updates the kernel map through the `bpf_map_update_elem` system call, and eBPF will enforce the new rule immediately.

3.4. Safety Constraints

Although the LLM is instructed to prefer the most restrictive minimal-action policies (e.g., block a single IP:Port), LLMs are inherently probabilistic and may hallucinate or produce overly permissive policies. PACE enforces multiple layers of constraints to mitigate these risks.

Factual and Contextual Bounding (Φ). Policies must be grounded in system knowledge K, which includes trusted internal IP ranges, critical service lists, and operational context. We define a semantic divergence metric $d_{\text{sem}}(O, K)$, measuring how far a candidate policy deviates from safe configurations:

$$\Phi: \quad d_{\text{sem}}(O, K) \leq \delta_{\max},$$

where $\delta_{\max}$ sets the maximum allowable deviation. For example, blocking the IP of a database server would violate Φ, whereas isolating an unknown external host does not. Violating policies are rejected and logged for review.

Practical Constraint Example. Suppose a new anomaly indicates excessive outbound traffic from a suspicious host, a candidate policy blocking all outbound ports of the process PID would be overly aggressive. Constraint Φ would detect this as deviating from critical service context and reject it, instead allowing the LLM to generate a targeted drop rule for the specific `IP:Port` pair, preserving legitimate service connectivity.

3.5. Operational Semantics

Formally, the PACE mediation pipeline can be expressed as:

$$x \xrightarrow{G} P^* \xrightarrow{V} P_v \xrightarrow{\phi} \mathcal{E},$$

where x is an observed anomaly, P^* is the candidate LLM-generated policy, V is the verifier applying invariants $\mathcal{I}$, P_v is the verified policy, and $\mathcal{E}$ is the enforced rule set. Verification is defined as:

$$V: \mathcal{P}^* \times \mathcal{I} \to \mathcal{P}_v, \quad p \in P^* \text{ is accepted } \Leftrightarrow \forall I_i \in \mathcal{I},\ p \models I_i.$$

When an LLM-generated policy p fails verification, the failing policy and the specific invariant violations are logged and used to refine the next LLM prompt, creating a feedback loop that improves future policy generation.

4. Formal Security Analysis

4.1. Safety Invariants

To guarantee safe enforcement, PACE defines a set of formal invariants (Table 1) over the candidate policy space $\mathcal{P}$ and system state $\mathcal{S}$. The primary safety property is:

$$\psi = \square\neg(s \in S_{\text{unsafe}}),$$

where $S_{\text{unsafe}} \subseteq \mathcal{S}$ captures service denial, policy conflicts, or semantic violations. SAT means satisfiability. Verification ensures:

$$\forall p \in \mathcal{P}, \quad \mathsf{SAT}(p \wedge \neg\psi) = \emptyset.$$

TABLE 1: Formal safety invariants enforced by the verifier

ID	Invariant	Formal Expression
I_1	No essential service blocking	$\forall s \in S_E,\ \texttt{allow}(s)$
I_2	Rule consistency	$\nexists(r_i, r_j) : (r_i \wedge r_j) \Rightarrow \bot$
I_3	Deterministic evaluation	$\forall t, \exists! a : f(P, t) = a$
I_4	Termination guarantee	$\texttt{depth}(P) < \infty$
I_5	Rollback safety	$\exists P_0 : P_v \Rightarrow P_0$ on violation

Relationship Between Φ and the Invariants $\mathcal{I}$. The factual and contextual bounding constraint Φ operates as a semantic precondition for the verifier, so that LLM-generated policies are compatible with the system knowledge base K (e.g., critical services, internal address ranges, and baseline configurations). Formally, Φ filters out semantically implausible or over-permissive policies before formal verification. In contrast, the invariant set $\mathcal{I} = \{I_1, I_2, I_3, I_4, I_5\}$ defines the structural and behavioral safety properties of the resulting policy. This relationship can be expressed through a hierarchical verification pipeline:

$$x \xrightarrow{G} P^* \xrightarrow{\Phi} P^{**} \xrightarrow{V(\mathcal{I})} P_v \xrightarrow{\phi} \mathcal{E},$$

where P^* is the raw LLM output, P^{**} is the semantically bounded subset satisfying Φ, and P_v is the formally verified policy satisfying all invariants. Only policies that fulfill both contextual and formal conditions are admissible:

$$P_v = \{\, p \in \mathcal{P} \mid p \models \Phi \wedge \bigwedge_{i=1}^{5} I_i \,\}.$$

Intuitively, Φ guarantees *contextual plausibility*, preventing rules that would disrupt critical functionality, while $\mathcal{I}$ enforces *formal soundness* and runtime safety. For example, if an LLM proposes a rule blocking traffic to `10.0.0.1:22`, which corresponds to the ssh protocol, Φ will reject it due to contextual conflict. A valid rule passing Φ must then also satisfy I_2 (consistency) and I_3 (determinism) to ensure that the final enforced policy cannot introduce contradictions or nondeterministic behavior within the firewall logic. The verifier is sound but possibly incomplete (i.e., may reject valid rules) depending on the expressiveness of $\mathcal{I}$. Together, these layers form a composite guarantee of semantic validity and formal correctness.

4.2. Verification Complexity

For $n = |\mathcal{R}|$ rules and $|\mathcal{I}|$ invariants, symbolic verification scales as $O(n^2)$ for pairwise conflict checks and $O(n \cdot |\mathcal{I}|)$ for invariant satisfaction. Optimizations, including incremental checks and rule hashing [21], reduce runtime to near-linear in practice.

4.3. Safety Preservation under Policy Composition

Consider two verified policies P_1 and P_2 satisfying all invariants $\mathcal{I}$. Their sequential composition $P_1 \oplus P_2 = \langle R_1 \cup R_2, \prec \rangle$ preserves safety:

1) *Invariant Closure:* If $\mathcal{I}$ is closed under conjunction, then $P_1 \oplus P_2 \models \mathcal{I}$.
2) *Conflict Preservation:* Symmetric and transitive conflict relations ensure that no new conflicts arise across P_1 and P_2.

Theorem: If invariants are closed under conjunction and conflicts are transitive, then the composition of individually verified policies preserves safety. This guarantees that incremental policy updates do not require full global re-verification, enabling real-time rule injection without compromising correctness.

4.4. Practical Implications of Constraints

These formal guarantees have concrete operational benefits by safeguarding against hallucinations in LLM outputs. For example, if an LLM-generated policy blocks all SSH traffic from a subnet that includes a critical monitoring server, Φ prevents this overreach. Similarly, deterministic evaluation (I_3) ensures that repeated packets with identical attributes always receive the same verdict, preventing race conditions or intermittent policy bypass. Rollback safety constraint (I_5) guarantees that a policy failing verification can be safely reverted to a prior verified state, preserving system continuity.

5. Preliminary Implementation

Our ongoing work focuses on prototyping the PACE constraint verification as described in Section 4. We implemented a prototype that simulates the *policy generation* and *formal verification* stages using ChatGPT-based policy synthesis and a symbolic Python verifier. Note that our preliminary implementation does not include the eBPF enforcement layer as they are our future work. We use ChatGPT-5 via OpenAI's web interface for LLM-based policy generation, and the future integration with potentially different LLM providers via API interfaces is straightforward.

5.1. Policy Generation

Using the structured prompt design described in Section 3.3, we use ChatGPT-5 to generate 1,000 candidate firewall rules in JSON form, for example:

```
{
  "policy_id": "rule-023",
  "action": "DROP",
  "target": {"ip": "203.0.113.45", "port":
      22, "proto": "tcp"},
  "justification": "Block external SSH from
      suspicious host"
}
```

The resulting policy set was then passed to the verifier for semantic and formal checking.

5.2. Verification Engine

The verifier implements both the semantic bounding constraint Φ and the formal invariants I_1–I_5 respectively defined in Section 3.4 and Section 4.

5.2.1. Semantic Bounding Constraint. In our prototype, we define a simplified semantic distance metric d_{sem} as the total number of violations among five categories: non-disruption of essential protocols (ports), non-disruption of essential services (combinations of IPs and ports), non-disruption of management IPs, free of wildcard rules, and no conflicts with existing rules that block malicious subnets. Rules violating Φ are rejected before formal checking.

5.2.2. Formal Invariant Checking. The verifier implements each invariant I_1–I_5 to check the candidate policies against the system context K. For example, I_1 checks that no essential services (from a predefined list in K) are blocked by any rule with action `DROP` or `REJECT`. Note that although I_1 is a subset of the semantic constraint Φ, it is re-checked here for formal completeness. I_2 checks for rule consistency and I_3 for determinism, both based on rule priorities. The policy lists are sorted by priority to facilitate the checks, and any violations (e.g., consistency of rules or determinism violations at the same level of priority) are recorded. I_4 checks that dependency chains among rules are acyclic and bounded in depth using a depth-first search. Finally, I_5 verifies that a prior verified snapshot exists for rollback safety. A rule is accepted only if it passes all invariant checks.

5.2.3. Feedback Loop. If a generated policy fails verification, the verifier logs the specific constraint violations, and formulates feedback for the LLM to refine the next LLM prompt, guiding the LLM to correct the identified issues in subsequent generations. We use the same structured JSON format for feedback as shown in the commented code snippet in Section 3.3 with the violation details and corrective guidance as shown in the listing in Section 5.2.3.

LLM Feedback for Policy Correction

```
<The original prompts and outputs...>
Policy verification failed. See
    violations below and correct the
    policies.
{
  "invariant_violations": [
    "I2: Conflicting actions at top
        priority for
        tuple=('198.51.100.1', '22')",
    "I3: Non-determinism for
        tuple=('198.51.100.1', '22')",
  ],
  "guidance": [
    "Resolve top-priority conflicts on
        the same IP:port; break ties
        deterministically with
        priority.",
    "Avoid priority ties with different
        actions at the same tuple.",
  ]
}
```

6. Evaluation

We performed an initial simulation to measure verification effectiveness and computational overhead. In this section, we describe our experimental setup, metrics, and results.

6.1. Experimental Setup

A total of 1,000 ChatGPT-5-generated candidate rules were fed to the verifier under a representative system context containing several essential services (e.g., database, ssh), pre-existing firewall rules such as known malicious subnets and trusted management IPs. Experiments were conducted on a Lenovo ThinkPad P14s laptop with 8 CPU cores and 32 GB RAM, running Ubuntu 22.04 LTS and Python 3.10.12.

6.2. Rule Verification Metrics

Among the 1,000 generated rules, we measured the pass rates of each verification step. Out of 1,000 generated rules, 963 rules passed all verification steps without iterative prompting, demonstrating that the LLM-generated policies are largely compliant with the formal safety invariants when properly guided with the initial prompts. The remaining 4% were rejected primarily for I_4 violations (chain depth exceeded). We report detailed breakdowns of the pass rates in Table 2.

TABLE 2: Verification Results

Constraint	Pass Rate
Φ only	97.7%
$\Phi + I_1$	97.7%
$\Phi + I_2$	97.7%
$\Phi + I_3$	97.7%
$\Phi + I_4$	96.3%
$\Phi + I_5$	97.7%
All ($\Phi + I_1$–I_5)	96.3%

We reviewed the rejected rules and confirmed that they indeed violated the specified invariants. For the invariants that have 100% pass rates (I_1, I_2, I_3, I_5), we observed no violations in the generated rules, indicating that the original prompt design effectively guided the LLM to avoid these issues. However, the purpose of the verifier is to provide a safety net against potential violations stemming from the potential "hallucination" of the LLM, especially as policies become more complex.

6.3. Verification Latency

We measure the verification latency as the average time taken to verify each rule. Figure 2 plots verification latency versus the number of rules, showing near-linear growth consistent with the theoretical $O(n)$ bound for the semantic and consistency checks. Verification time is negligible compared with human-in-the-loop deployment delays, which take up to "tens of minutes" [22] for manual deployment and without formal verification.

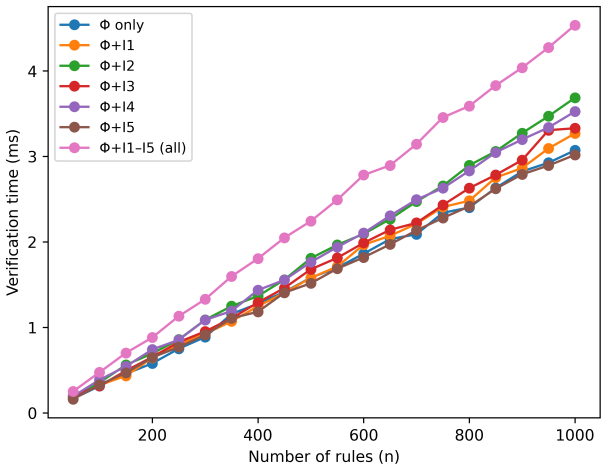

Figure 2: Latency of policy verification with different numbers of rules, showing scalability of PACE.

7. Related Work

7.1. eBPF-based Network Security

eBPF has been widely adopted in the cybersecurity domain, specifically for network monitoring and policy enforcement [11]. Cilium [23] uses eBPF for networking and policy enforcement in Kubernetes. Falco [24] uses eBPF for behavioral activity monitoring and runtime security detection. Several studies have used eBPF with LLM-created rules for real-time policy generation [12], [25], [26], [27]. Specifically, Kim et al. [25] uses LLM for network anomaly analysis and eBPF for Distributed Denial of Service (DDoS) prevention. Yu et al. [26] introduces an integrated framework that uses eBPF for system event collection and language model for malware threat reasoning. PACE agrees with existing work on the effectiveness of eBPF for network event collection and policy enforcement, and it further provides a formal framework for verifying the correctness of LLM-generated policies before enforcement.

7.2. AI in Cyber Threat Response

Existing research leverages AI for high-level tasks like log summarization or vulnerability explanation. Ismail et al. [2] uses LLM agents for autonomous cybersecurity responses in cloud computing. Liu et al. [28] introduces an LLM-guided rule synthesis framework that extracts and validates semantic constraints on API parameters for misuse detection. By combining static analysis with LLM-based rule refinement, it shows how large models can capture implicit security logic. These solutions integrate LLM's generative decision-making directly into the enforcement pipeline, and thus are orthogonal to PACE's goal. However, PACE constrains the LLM's output through formal verification that provide safety guarantees for the generated policies.

Kramer et al. [29] examines how can LLMs support analysts during incident response, noting that while LLMs

improved report generation, they often produced incomplete or inconsistent reasoning. Liu et al. [30] performs a comprehensive study of LLM applications in both offensive and defensive network security. It highlights how LLMs can assist in intrusion detection, vulnerability analysis, and automated policy generation. Both studies underscore the need for safety mechanisms around LLM-driven automation in cyber threat response. PACE directly addresses this challenge by applying formal verification to LLM-generated policies, ensuring safe and correct enforcement at the kernel level.

7.3. Formal Methods in AI Safety

Formal methods have been applied to ensure the safety and reliability of AI systems, particularly in AI alignment and verification [31], [32], [33], autonomous driving and robotics [34], [35], [36]. Specifically, recent research has explored symbolic reasoning to improve interpretability and controllability, which is a line of work closely aligned with the motivation behind PACE. Symbolic or neuro-symbolic LLM frameworks aim to combine the generative flexibility of language models with the rigor of symbolic logic and formal systems [37], [32], [33], [38], [39], [40], [41], by integrating logical constraints directly into the model's decoding process, constraining the generation of outputs to satisfy pre-defined formal rules.

LLMs pose unique challenges due to their complexity and unpredictability, especially when used in safety-critical applications, such as code generation [42] and hardware policy synthesis [43]. Wang et al. [44] introduces a framework that utilizes LLM to translate natural language network policies into formal specifications, which are validated by a model checker. These works demonstrate the feasibility of combining LLMs with formal verification to augment the reliability of AI-generated outputs. PACE builds on this foundation by integrating formal verification directly into the LLM-driven firewall policy generation and enforcement process.

8. Discussion

While PACE demonstrates that it is promising to constrain and enforce LLM-generated firewall policies with formal guarantees, several limitations remain for future exploration.

8.1. Expressiveness vs. Verifiability Trade-off

PACE prioritizes formal safety by constraining the space of admissible rules through invariants. While this design ensures verifiability, it may also restrict the expressiveness of policies, particularly for complex or stateful network behaviors. For instance, temporal or conditional rules (e.g., rate-limiting, transient dependencies) may not be easily representable under the invariant set. Extending the verifier to reason over temporal logic would broaden expressiveness at the cost of increased verification complexity. As we will discuss in Section 9, future work could explore temporal logic frameworks, such as Linear Temporal Logic (LTL), to capture time-dependent policies.

A related challenge lies in balancing the rigidity of invariants with the flexibility needed for adaptive security policies. Overly strict invariants may reject useful rules generated by the LLM, while relaxed invariants could reintroduce risk. Future work could explore dynamic invariant adjustment based on contextual factors, such as threat levels or network conditions, to optimize this trade-off.

8.2. Interpretability

Although PACE aims to minimize human intervention, the interpretability of LLM-generated rules and the traceability of verification decisions remain partially automated. In practice, security analysts may still require interpretable justifications for enforcement outcomes, especially with audit or compliance requirements. Integrating natural language explanations derived from verification traces could enhance transparency without breaking the formal rigor of the system. Also, neuro-symbolic methods [45] that combine symbolic reasoning with neural networks could improve interpretability.

8.3. Assumption of Trustworthy Verification Layer

PACE assumes that the verifier and its underlying formal logic are correct and free from implementation errors. We also do not consider the privacy and security of the online LLM service used for policy generation although past work has explored secure and private model querying [46], [47]. Based on our trust assumptions, the verifier and LLM themselves are part of the Trusted Computing Base (TCB). Furthermore, any semantic mis-specification or logical incompleteness in the invariant definitions (I_1–I_5) directly compromises overall system soundness. This assumption limits fault isolation and shifts security assurance toward the correctness of the verifier logic rather than the verified policy alone. A verifiable verifier, for example, one built atop proof assistants such as Coq [48] or Isabelle [49], would be needed to fully close this trust gap.

9. Future Work

PACE proposes a blueprint for a host firewall framework capable of utilizing and constraining the LLM-generated network policies for real-time enforcement via eBPF. This preliminary implementation of the verification framework and its evaluation demonstrate the feasibility of our approach, showing high verification pass rates and low computational overhead. This prototype establishes that PACE's verification design is both *effective* and *computationally practical*.

Future work will integrate the verified rules with an eBPF enforcement layer, building upon prior work on

eBPF-based firewalls [50] and dynamic eBPF program updates [11]. Future work will also focus on: (1) quantitative confidence enforcement, in which we model the enforcement decision as a stochastic control problem balancing risk and agility while maintaining satisfaction of safety invariants; and (2) exploration of the LLM's capacity to generate, verify, and compile new eBPF C code for advanced, stateful packet filtering logic, beyond simple rule updates. Such capabilities would enable adaptive, context-aware firewall behavior for enforcing temporally dependent policies, which we can leverage temporal logic (LTL [51], [52]) to reason over time-dependent policies for conditional policy activation.

10. Conclusion

We introduced PACE, a formal framework for constraining and enforcing LLM-generated firewall policies using eBPF. By performing invariant checking on the policy generated by LLM, PACE ensures that only safe and verifiable LLM-generated rules are deployed, bridging the gap between probabilistic AI outputs and system requirements. Our formal analysis demonstrates that safety properties are preserved under rule composition and dynamic updates, providing strong theoretical guarantees toward real-time firewall policy enforcement.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work received no external funding.

References

[1] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, pp. 62–67, 2004.

[2] Ismail, R. Kurnia, Z. A. Brata, G. A. Nelistiani, S. Heo, H. Kim, and H. Kim, "Toward robust security orchestration and automated response in security operations centers with a hyper-automation approach using agentic artificial intelligence," *Information*, vol. 16, no. 5, p. 365, 2025.

[3] V. Z. Mohale and I. C. Obagbuwa, "Evaluating machine learning-based intrusion detection systems with explainable ai: enhancing transparency and interpretability," *Frontiers in Computer Science*, vol. 7, p. 1520741, 2025.

[4] X. Wang, Z. Yao, and M. Papaefthymiou, "A real-time electrical load forecasting and unsupervised anomaly detection framework," *Applied Energy*, vol. 330, p. 120279, 2023.

[5] S. A. R. Shah and B. Issac, "Performance comparison of intrusion detection systems and application of machine learning to snort system," *Future Generation Computer Systems*, vol. 80, pp. 157–170, 2018.

[6] L. Layman and W. Roden, "A controlled experiment on the impact of intrusion detection false alarm rate on analyst performance," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 67, no. 1. SAGE Publications Sage CA: Los Angeles, CA, 2023, pp. 220–225.

[7] "Use AI securely and responsibly," https://cloud.google.com/architecture/framework/security/use-ai-securely-and-responsibly, 2025.

[8] "Azure WAF guided investigation Notebook using Microsoft Sentinel for automated false positive tuning," https://azure.microsoft.com/en-us/blog/azure-waf-guided-investigation-notebook-using-microsoft-sentinel-for-automated-false-positive-tuning, 2023.

[9] "Handle false positives in Microsoft Sentinel," https://docs.azure.cn/en-us/sentinel/false-positives, 2025.

[10] T. Huang, L. You, N. Cai, and T. Huang, "Large language model firewall for aigc protection with intelligent detection policy," in *2024 2nd International Conference On Mobile Internet, Cloud Computing and Information Security (MICCIS)*. IEEE, 2024, pp. 247–252.

[11] "L4Drop: eBPF-based DDoS Mitigations," https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations, 2018.

[12] F. Dall'Agata, "Instructing network devices via large language models," Ph.D. dissertation, Politecnico di Torino, 2024.

[13] "The netfilter.org - iptables project," https://netfilter.org/projects/iptables/index.html, 2024.

[14] "nftables wiki," https://wiki.nftables.org/wiki-nftables/index.php/Main_Page, 2024.

[15] "eBPF docs," https://docs.ebpf.io, 2024.

[16] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating linux security with ebpf iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, 2018, pp. 108–110.

[17] W. Brach, K. Košt'ál, and M. Ries, "The effectiveness of large language models in transforming unstructured text to standardized formats," *IEEE Access*, 2025.

[18] K. Dzeparoska, J. Lin, A. Tizghadam, and A. Leon-Garcia, "Llm-based policy generation for intent-based management of applications," in *2023 19th International Conference on Network and Service Management (CNSM)*. IEEE, 2023, pp. 1–7.

[19] K. Neupane, K. Kostage, S. Peppers, A. Pandey, P. Calyam, and C. Qu, "Netprompt: Llm-driven programmable network policy management and optimization," in *2025 34th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2025, pp. 1–9.

[20] L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton, "Seeing through network-protocol obfuscation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 57–69.

[21] A. Yoshioka, S. H. Shaikot, and M. S. Kim, "Rule hashing for efficient packet classification in network intrusion detection," in *2008 Proceedings of 17th International Conference on Computer Communications and Networks*. IEEE, 2008, pp. 1–6.

[22] C. C. Zhang, M. Winslett, and C. A. Gunter, "On the safety and efficiency of firewall policy deployment," in *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 2007, pp. 33–50.

[23] "Cilium - Cloud Native, eBPF-based Networking, Observability, and Security," https://cilium.io, 2025.

[24] "Modern eBPF probe is ready to shine — Falco," https://falco.org/blog/falco-modern-bpf-0-35-0/, 2023.

[25] J. Kim and G. D. Rodosek, "Protocol-aware and adaptive ddos defense framework," in *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*. IEEE, 2025, pp. 257–260.

[26] S. Yu, Z. Li, J. Chen, X. Deng, and C. Hou, "Malware behavior detection system with rag-enhanced ebpf and advanced language model," in *2024 5th International Conference on Smart Grid and Energy Engineering (SGEE)*. IEEE, 2024, pp. 500–506.

[27] X. Gao, X. Zhu, B. V. Shobhana, Y. Yang, A. Krishnamurthy, and R. Mahajan, "Offloading the tedious task of writing ebpf programs," in *Proceedings of the 3rd Workshop on eBPF and Kernel Extensions*, 2025, pp. 63–69.

[28] J. Liu, Y. Yang, K. Chen, and M. Lin, "Generating api parameter security rules with llm for api misuse detection," *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[29] D. Kramer, L. Rosique, A. Narotam, E. Bursztein, P. G. Kelley, K. Thomas, and A. Woodruff, "Integrating large language models into security incident response," in *Twenty-First Symposium on Usable Privacy and Security (SOUPS 2025)*, 2025, pp. 133–148.

[30] H. Liu, J. Xue, S. Zhao, Y. Liu, and Z. Lu, "The dual role of large language models in network security: Survey and research trends," in *Proceedings of the 2025 ACM Workshop on Wireless Security and Machine Learning*, 2025, pp. 20–25.

[31] Z. Wang, D. He, Z. Zhang, X. Li, L. Zhu, M. Li, and J. Liu, "Formalization driven llm prompt jailbreaking via reinforcement learning," *arXiv preprint arXiv:2509.23558*, 2025.

[32] Z. Li, W. Hua, H. Wang, H. Zhu, and Y. Zhang, "Formal-llm: Integrating formal language and natural language for controllable llm-based agents," *arXiv preprint arXiv:2402.00798*, 2024.

[33] X. Wu, Y. Li, J. Sun, and C. Lu, "Symbol-llm: leverage language models for symbolic system in visual human activity reasoning," *Advances in neural information processing systems*, vol. 36, pp. 29 680–29 691, 2023.

[34] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–41, 2019.

[35] Z. Xu, Y. Zhang, E. Xie, Z. Zhao, Y. Guo, K. K. Wong, Z. Li, and H. Zhao, "Drivegpt4: Interpretable end-to-end autonomous driving via large language model," *IEEE Robotics and Automation Letters*, 2024.

[36] M. González-Santamarta, F. J. Rodríguez-Lera, Á. M. Guerrero-Higueras, and V. Matellán-Olivera, "Integration of large language models within cognitive architectures for autonomous robots," *arXiv preprint arXiv:2309.14945*, 2023.

[37] F. Xu, Z. Wu, Q. Sun, S. Ren, F. Yuan, S. Yuan, Q. Lin, Y. Qiao, and J. Liu, "Symbol-llm: Towards foundational symbol-centric interface for large language models," *arXiv preprint arXiv:2311.09278*, 2023.

[38] X. Quan, M. Valentino, D. S. Carvalho, D. Dalal, and A. Freitas, "Peirce: Unifying material and formal reasoning via llm-driven neuro-symbolic refinement," *arXiv preprint arXiv:2504.04110*, 2025.

[39] N. Dave, D. Kifer, C. L. Giles, and A. Mali, "Investigating symbolic capabilities of large language models," *arXiv preprint arXiv:2405.13209*, 2024.

[40] A. G. Deé-Lukács and A. Földvári, "Dependability assurance with symbolic reasoning in llm-enabled systems," in *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2025, pp. 47–54.

[41] M. Fang, S. Deng, Y. Zhang, Z. Shi, L. Chen, M. Pechenizkiy, and J. Wang, "Large language models are neurosymbolic reasoners," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 38, no. 16, 2024, pp. 17 985–17 993.

[42] C. J. Chong, Z. Yao, and I. Neamtiu, "Artificial-intelligence generated code considered harmful: A road map for secure and high-quality code generation," *arXiv preprint arXiv:2409.19182*, 2024.

[43] S. Tarek, D. Saha, S. K. Saha, M. Tehranipoor, and F. Farahmandi, "Socurellm: An llm-driven approach for large-scale system-on-chip security verification and policy generation," in *2025 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2025, pp. 335–345.

[44] C. Wang, M. Scazzariello, A. Farshin, S. Ferlin, D. Kostić, and M. Chiesa, "Netconfeval: Can llms facilitate network configuration?" *Proceedings of the ACM on Networking*, vol. 2, no. CoNEXT2, pp. 1–25, 2024.

[45] A. A. Garcez and L. C. Lamb, "Neurosymbolic ai: The 3 rd wave," *Artificial Intelligence Review*, vol. 56, no. 11, pp. 12 387–12 406, 2023.

[46] C. J. Chong, C. Hou, Z. Yao, and S. M. S. Talebi, "Casper: Prompt sanitization for protecting user privacy in web-based large language models," *arXiv preprint arXiv:2408.07004*, 2024.

[47] A. R. Chowdhury, D. Glukhov, D. Anshumaan, P. Chalasani, N. Papernot, S. Jha, and M. Bellare, "Preempt: Sanitizing sensitive prompts for llms," *arXiv preprint arXiv:2504.05147*, 2025.

[48] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter, "Coq coq correct! verification of type checking and erasure for coq, in coq," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–28, 2019.

[49] L. C. Paulson, *Isabelle: A generic theorem prover*. Springer, 1994.

[50] J. Zhang, P. Chen, Z. He, H. Chen, and X. Li, "Real-time intrusion detection and prevention with neural network in kernel using ebpf," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 416–428.

[51] Z. Yao, "Formal trust and threat modeling using large language models," in *2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*. IEEE, 2024, pp. 232–239.

[52] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 411–420.