UMBC CODEBOT '25 Workshop
Columbia, MD / 25-26 February 2025

Enhancing Trustworthiness in LLM-Generated Code: A Reinforcement Learning and Domain-Knowledge Constrained Approach

Aritran Piplai
The University of Texas at El Paso
apiplai@utep.edu

Anantaa Kotal
The University of Texas at El Paso
akotal@utep.edu

Sudip Mittal
Mississipi State University
Mittal@cse.msstate.edu

Karuna Pande Joshi
University of Maryland, Baltimore County
kjoshi1@umbc.edu

Tim Finin
University of Maryland, Baltimore County
finin@umbc.edu

Anupam Joshi
University of Maryland, Baltimore County
joshi@umbc.edu

Reinforcement Learning to ground LLMs for trustworthy malware detection

Imagine analyzing a piece of code that uses the function ConnectToServer() with an encrypted string as its argument. A large language model (LLM), trained on extensive programming data, might flag the use of encryption as suspicious and generate an explanation suggesting that the function likely connects to a malicious server. While this explanation might seem plausible, it can often be unfaithful—it overgeneralizes based on statistical patterns from its training data without truly understanding the context or validating its claims [8]. A REACT (Reasoning and Acting) framework, which combines reasoning with action steps, is likely a better approach because it allows the LLM to propose actions—such as decrypting the string or examining server connections—while reasoning about the results [7]. However, REACT still lacks a feedback mechanism to evaluate the effectiveness of those actions or iteratively refine the sequence based on empirical observations. Without such feedback, it risks falling short in dynamic scenarios, where the validation of predictions and adaptation to new evidence are critical [10].

This is where a reinforcement learning (RL) agent steps in to provide actionable validation. The RL system treats the analysis as a sequence of targeted actions designed to uncover the truth [11]. First, it might select a test case that decrypts the string argument to inspect its contents. If the decrypted string resembles a known Command-and-Control (C2) server pattern, the agent might move to the next test case. This iterative process ensures that the RL agent provides structured guidance while relying on external execution to validate and refine the analysis [10].

Each action the RL agent takes is guided by a reward function that prioritizes uncovering concrete evidence of malicious behavior. If the sandbox reveals suspicious network traffic or matches known malware communication patterns, the agent receives a high reward [9]. Conversely, if the tests show benign or unrelated behavior, the reward is low, pushing the agent to refine its action sequence over time [4].

Consider a hypothetical example, a code sample with a function SendDataToServer() that uses obfuscated arguments. An LLM might reason that decrypting the arguments and checking them against a list of malicious IPs is sufficient, generating an explanation that the encrypted data likely hides malicious content [2]. However, this static reasoning may miss critical nuances, such as the obfuscated logic in GenerateArguments() that dynamically generates sensitive data for exfiltration [11].

An RL agent, on the other hand, dynamically identifies high-value actions. It prioritizes analyzing obfuscated code, decrypting arguments, and constructing a sequence of test cases that effectively determine whether the code is malicious [10]. By adapting its actions based on observed results, the RL agent ensures a thorough and targeted analysis. A knowledge-guided RL agent can output a credible list of tests, which can then be used to enhance the reasoning capabilities of an LLM, making its explanations more reliable and context-aware [1].

By aligning the LLM with the RL agent using techniques like Direct Preference Optimization (DPO), the LLM learns to refine its reasoning, prioritize dynamic validation, and provide more reliable and context-aware explanations for identifying malicious behavior [11].

Domain Knowledge-Constrained Code Generation

Since the code generated by LLMs cannot be blindly trusted, we must develop mechanisms that ensure its behavior aligns with expected norms and security standards [6]. By leveraging domain knowledge and structured constraints, we can impose safeguards that systematically detect and correct deviations in generated code [2]. One promising approach builds on our recent work in system call sequence analysis, where we treat runtime behavior as a language and infer its grammar using large training samples [8]. Just as LLMs learn natural language structures, they can be trained to recognize the valid syntax and semantics of system calls, flagging any sequence that deviates as potentially harmful or anomalous [7].

This principle can be extended to code generation itself, where domain constraints are explicitly encoded into the loss function of the model, ensuring that generated code adheres to best practices in security, reliability, and maintainability [6]. For example, secure coding guidelines dictate that array accesses must include bounds checks to prevent buffer overflows. By incorporating this rule into the loss function, we penalize generated outputs that fail to include necessary safety checks [11]. Similarly, rules regarding memory management, authentication enforcement, and secure API usage can be embedded as constraints, refining the model's outputs in a structured and systematic way [9].

Beyond security, organizational coding policies play a crucial role in ensuring the generated code is usable and aligns with existing infrastructure [4]. Many enterprises enforce specific data storage conventions, such as preferring databases over flat files or requiring encrypted storage for sensitive data. Without guidance, LLMs may generate code that violates these policies, leading to inconsistencies, inefficiencies, or outright vulnerabilities [1]. To address this, we introduce policy-aware constraints in training, ensuring the model internalizes and respects predefined rules [2]. These constraints can be dynamically adjusted based on context-specific business rules, allowing organizations to fine-tune code generation for their unique requirements [8].

A powerful extension of this approach involves reinforcement learning (RL) with domain-specific rewards [11]. Instead of relying purely on static constraints, an RL-driven code generation model interacts with an execution environment, receiving feedback based on real-world performance [1]. If generated code fails unit tests, exhibits unexpected system behavior, or violates security policies, it receives a negative reward, guiding the model to refine its output iteratively [10]. This aligns well with the digital twin paradigm, where a sandboxed execution environment mimics production conditions, enabling proactive validation of generated code before deployment [4].

Moreover, we can incorporate knowledge graphs and formal verification techniques into the generative process [10]. Knowledge graphs store relationships between entities such as functions, security rules, and software dependencies, allowing LLMs to make context-aware decisions [2]. Meanwhile, formal verification techniques, such as symbolic execution and static analysis, can validate generated code against logical correctness conditions, further improving trustworthiness [11].

By integrating grammar-based constraints, policy-aware loss functions, reinforcement learning, and formal verification, we create a robust, knowledge-constrained framework for LLM-based code generation [1]. This ensures that generated code is not only syntactically correct but also functionally reliable, security-compliant, and contextually aligned with domain-specific requirements.

References

1. Kotal, Anantaa, and Anupam Joshi. "Differentially Private Synthetic Data Generation Using Context-Aware GANs." 2024 IEEE International Conference on Big Data (BigData). IEEE, 2024.

2. Garza, Leon, et al. "PrivComp-KG: Leveraging KG and LLM for Compliance Verification." 2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA). IEEE, 2024.

3. Kotal, Anantaa, Brandon Luton, and Anupam Joshi. "KiNETGAN: Enabling Distributed Network Intrusion Detection through Knowledge-Infused Synthetic Data Generation." 2024 IEEE 44th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2024.

4. Das, Nilanjana, et al. "Change management using generative modeling on digital twins." 2023 IEEE International Conference on Intelligence and Security Informatics (ISI). IEEE, 2023.

5. Piplai, Aritran, et al. "Knowledge-enhanced neurosymbolic artificial intelligence for cybersecurity and privacy." IEEE Internet Computing 27.5 (2023): 43-48.

6. Kotal, Anantaa, et al. "Privetab: Secure and privacy-preserving sharing of tabular data." *Proceedings of the 2022 ACM on international workshop on security and privacy analytics*. 2022.

7. Shin, Jihoon, et al. "Towards Building Generalizable Models for Malware Detection." *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024.

8. Mullins, Elizabeth, et al. "Enhancing classroom teaching with LLMs and RAG." Proceedings of the 25th Annual Conference on Information Technology Education. 2024.

9. Mitra, Shaswata, et al. "Localintel: Generating organizational threat intelligence from global and local cyber knowledge." arXiv preprint arXiv:2401.10036 (2024).

10. Piplai, Aritran, et al. "Knowledge guided two-player reinforcement learning for cyber attacks and defenses." 2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, 2022.

11. Piplai, Aritran, Anupam Joshi, and Tim Finin. "Offline RL+ CKG: A hybrid AI model for cybersecurity tasks." Proceedings of the AAAI 2023 Spring Symposium on Challenges Requiring the Combination of Machine Learning and Knowledge Engineering (AAAI-MAKE 2023). 2023.