Emerging Architectures for Secure Enterprise Language Models: Evolving alongside Industry

By David Pierce & collaborators

In the evolving landscape of Large Language Models (LLMs), there's a growing emphasis on **runtime abstraction and modular environments**[1]. The rationale? It offers a scalable and flexible foundation for hosting LLMs. By introducing event-driven and batch monitoring of Language Models, defining a Master Prompt as an abstraction layer[2], and/or customizing the architecture and tokenization process (for training data & inputs), enterprises can introduce added layers of security. Taken together these customizations **sanitize inputs**[3]**, monitor outputs**[4], and obscure the model's inner workings; making it challenging for potential attackers to decipher. However, it's crucial to understand that this method isn't foolproof. Skilled adversaries might still reverse-engineer the model or exploit it in unforeseen ways.

Another strategy actively in play is that of **red-teaming**[5]. This involves actively probing the system for vulnerabilities, allowing organizations to identify and rectify weak points before they're exploited. In tandem with this, there's a move towards **scanning inputs for suspicious patterns**[6], preventing them from entering the runtime. This can thwart attacks that rely on specific input sequences (e.g. prompt injection, control strings, etc). There are even emerging methods for defining an event-hook within an LLM vector space (e.g. a-inputs x b-time x c-distance).

Yet, every strategy has its limitations. Abstraction and active monitoring are effective, but introduce cost. While customizing the model architecture enhances security, it also introduces complexity. Red-teaming, though effective, can be resource-intensive and co-locate knowledge within a specific vertical. Hence, a multi-pronged approach, known as **defense in depth**[7], is emerging as the best practice. This involves combining various strategies, actively monitoring industry and vendor best-practices, in pursuit of creating a robust defense mechanism.

The industry is also witnessing the rise of **template-driven**[8] **control strings**[9], particularly with tools like LLM-Attacks[10] and the use of its "AttackPrompt" class. This automated attack class is designed to generate prompts, testing the resilience of AI models against adversarial onslaughts. The implications are profound. It's now evident that AI models can be manipulated in unintended ways, leading to potential risks like generating harmful content or revealing sensitive data. However, tools like LLM-Attacks also offer a silver lining. They provide a means to test and bolster the robustness of AI models.

In conclusion, as we harness the transformative potential of LLMs, it's paramount to approach with a security-first mindset. This involves understanding the evolving threats, adopting a multi-layered defense strategy, and staying updated with the latest developments in the field.

Note: The author of this document has leveraged and iterated the above in defining reference architectures. It is important to keep a holistic perspective, even in matters of security, specifically by allowing these models to effectively "ground" their answers via additional context retrieved at time of execution (e.g. via agentic frameworks[11]). This accomplished by **monitoring the non-conformity of output**[12] as well as active immutable versioning of trusted resources[13] that might that offer additional insight and prevent tunnel vision[14] by the model.

Sources:

1) DeFacto Default LLM Architectures
2) Abstraction via Prompt Components
3) Inspecting for Prompt Injection
4) Predicting LLM Behavior via Conformal Prediction
5) Red Teaming for LLMs
6) Scanning for Suspicious Input Patterns
7) Defense in Depth for LLMs
8) Templated Generation of Near-Universal LLM Control Strings
9) The Academic Paper for those Near-Universal LLM Control Strings
10) Carnegie Mellon Research Website for LLM Control Strings
11) Agentic Frameworks for LLMs & Application Utilization
12) Monitoring LLMs for Non-Conformity of Output
13) Indirect Injection & Versioned Immutable Trusted Resources
14) Avoiding Tunnel Vision in AI Retrieval