Conjunctive Poisoning in AI Supply-Chain Applications

Anonymous ACL submission

Abstract

Large language and vision-language models are increasingly deployed through inference pipelines that include prompt wrappers (e.g., templates and post-processing scripts) and configuration metadata (e.g., JSON/YAML files) that together shape model outputs. While model weights and binaries are routinely verified, these textual deployment artifacts remain weakly protected despite directly influencing runtime behavior. We show that a malicious developer can pair a benign-looking wrapper with crafted metadata to deterministically alter post-generation behavior without modifying model weights, training data, or inference code. We formalize this mechanism as a *conjunctive gate*, where activation depends on both an embedded wrapper marker and cryptographically bound metadata. We evaluate the attack across ten open- and closed-source LLM/VLM deployments and multiple tasks, and assess prompt and system level defenses including static metadata inspection, wrapper scanners, PromptShield, and SigStore-based artifact signing. To mitigate this risk, we introduce TIF-BAH, a lightweight middleware defense that verifies wrapper integrity and records behavioral attestations during inference. Our results reveal that wrapper-metadata interactions form an under-protected execution layer in modern AI deployments, exposing a new behavioral attack surface in the model deployment pipeline. Anonymized code is available at `https://anonymous.4open.science/r/llm_temp-71C4/`.

1 Introduction

Large Language Models (LLMs) and Vision-Language Models (VLMs) have rapidly evolved from research systems into core infrastructure for modern applications, powering customer support, software engineering, analytics, and creative tools (Zhao et al., 2025). Multimodal systems further extend this capability by integrating visual reasoning and language understanding (Buschoff et al., 2024). While these models are typically defined by their learned parameters, their deployed behavior is also mediated by surrounding textual infrastructure such as prompts, templates, wrappers, and configuration files.

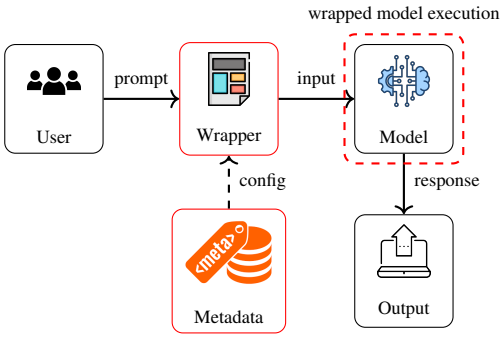

Figure 1: Template-layer taxonomy in LLM/VLM deployments. A user query is wrapped into a prompt template before being sent to the model. Configuration metadata (e.g., JSON/YAML files) is loaded by the wrapper and influences runtime behavior such as formatting or routing. The model produces a response, which the wrapper may post-process before returning the final output. Hidden activation gates can reside in **wrapper logic, metadata fields**, or their interaction. The wrapper loads and executes metadata parameters, and the attack exploits their interaction.

Prior security research on LLMs has focused largely on attacks against model internals. Data poisoning and backdoor attacks manipulate training data or gradients to implant triggers (Gu et al., 2019; Carlini et al., 2022), while composite backdoors extend these mechanisms (Huang et al., 2024). At inference time, adversarial prompts, prompt injection, and jailbreak techniques attempt to override safety constraints (Sharma and et al., 2024; Cheng et al., 2025). Corresponding defenses include weight signing, watermarking, gradient auditing, and output filtering. These approaches implicitly assume that the primary attack surface lies

within the model or its inputs. However, deployed ML systems resemble complex software supply chains. Incidents such as the SolarWinds breach and the xz-utils compromise demonstrate how a single modified dependency can affect thousands of downstream systems (Kruti et al., 2023; Przymus and Durieux, 2025). Modern ML ecosystems inherit these risks through extensive reuse of pre-trained checkpoints, tokenizer files, example notebooks, and deployment scripts from public repositories (Ait et al., 2023). While model binaries and containers are often signed or hashed, the textual deployment layer-wrappers, templates, and configuration metadata-remains largely mutable and weakly protected.

As shown in Figure 1, we define the template layer as the set of deployment artifacts that mediate model inputs and outputs at inference time. These artifacts include wrapper templates and configuration metadata. Wrappers implement executable logic for prompt construction, formatting, and post-processing, whereas metadata (e.g., JSON/YAML files) only store configuration parameters such as decoding or routing settings. Thus, wrappers and metadata differ in role: wrappers execute the logic, while metadata parameterize that logic.

RQ1. Can template-layer artifacts deterministically alter LLM/VLM behavior post-deployment without modifying weights or prompts? To assess the effect of wrapper-metadata pairs on runtime behavior, we construct proof-of-concept deployments in which the model is left unchanged and only the wrapper and metadata are modified. We implement post-generation actions at the middleware layer and evaluate whether the behavior change is reliable across diverse LLM and VLM deployments.

RQ2. How does cryptographic binding change the detectability and robustness of template-layer manipulations? To characterize the audit surface, we evaluate representative variants: (i) clean baseline, (ii) metadata-only readable conditions, (iii) wrapper-only triggers, and (iv) the full conjunctive mechanism that binds a hidden wrapper marker to metadata via keyed digests and a canonical wrapper-lock hash. We compare which variants are exposed by artifact-local inspection (e.g., scanning metadata or wrapper text) versus which require cross-artifact reasoning.

RQ3. How general and practical is deployment-time conjunctive poisoning across models, modalities, and tasks? To evaluate generality and operational cost, we run the same gate logic across ten open and closed-source models over both text and vision-language tasks, and measure activation reliability and latency impact under realistic inference pipelines (local checkpoints and API-based access).

We answer these questions through systematic ablations of wrapper-metadata variants and large-scale evaluation across diverse models and tasks.

Contributions. 1. We expose a deployment-time vulnerability in the template layer where wrapper-metadata pairs can deterministically control post-generation behavior without modifying model weights, training data, or system prompts.
2. We formalize a conjunctive cryptographic activation mechanism (marker + wrapper-lock hash) that increases stealth and hijack resistance compared to metadata-only or wrapper-only triggers.
3. We empirically demonstrate deterministic activation across ten evaluated models covering both text and multimodal tasks. Across all models, the gate activates reliably with less than 0.5% latency overhead.
4. We analyze existing prompt and system level defenses, including static metadata inspection, wrapper scanners, PromptShield, and Sigstore-based artifact signing, and show that they do not reliably prevent the cryptographically bound conjunctive attack. To address this gap, we introduce Template Integrity Filter with Behavioral Attestation Header (TIF-BAH), a lightweight mitigation that restores verifiability to the wrapper layer.

2 Related Works

2.1 Execution Semantics of Wrappers and Metadata in Deployed LLM/VLM Systems

In deployed systems, wrappers and configuration metadata execute as part of the inference pipeline and directly influence model behavior. Modern LLM and VLM deployments load model weights together with configuration files and wrapper code responsible for prompt construction, preprocessing, tool routing, and post-generation formatting (Greshake and et al., 2023; Viswanathan et al., 2023). A common deployment pattern is repository-driven execution: platforms such as Hugging Face and GitHub distribute not only model checkpoints but also entrypoint scripts, UI logic, and runtime configuration metadata (Osborne et al., 2024; HuggingFace, 2026; Qui, 2026). Wrapper scripts as-

Attack Class	Targets	Timing	Model Access	Detectable Offline
Data poisoning (Bowen et al., 2025)	Training data	Pre-training	Yes	Yes
Model backdoor (Li et al., 2025)	Weights	Training / Fine-tune	Yes	Yes
Prompt injection (Liu et al., 2024)	User input	Inference	No	Yes
Template attack (Guo et al., 2025)	Agentic System	Structure	Yes	No
Conjunctive poisoning (ours)	**Template layer**	**Deployment**	**No**	**No**

Table 1: Comparison with prior attack classes. Our approach uniquely targets the template layer and operates post-deployment.

semble prompts and apply post-processing before outputs are returned, making them a critical control point in the inference pipeline (Hu et al., 2025; PromptScript, 2025; Miao et al., 2024).Framework APIs further blur the boundary between configuration and execution by automatically loading metadata governing tokenization, decoding, and pipeline composition (Shi et al., 2025; Su et al., 2024). Agentic systems extend this behavior by injecting structured metadata such as tool schemas or routing descriptors (Mo et al., 2026). As a result, wrapper-metadata pairs form a lightweight execution layer in which metadata parameterizes behavior while wrappers implement it (Horng and Lu, 2000).

2.2 Supply-Chain Vulnerabilities in Machine Learning

Prior work studies ML-specific supply-chain attacks involving dependency compromise and runtime manipulation (Gao et al., 2025; Patel et al., 2025). Real-world incidents include compromised PyTorch dependencies enabling SSH key exfiltration (PyTorch Foundation, 2022), vulnerabilities in the Hugging Face transformers ecosystem (Database, 2024), and injection attacks in CI/CD pipelines (Young, 2024; Stawinski, 2024). These incidents demonstrate how compromised artifacts can propagate through downstream research and deployment environments (Zheng et al., 2023; Gao et al., 2025). Existing mitigations primarily focus on binary-level integrity through signed distributions, dependency locks, and container attestation (Ohm et al., 2020; Gokkaya et al., 2023). However, these defenses rarely cover textual deployment artifacts such as templates or configuration metadata, leaving wrapper logic and prompt templates widely reused yet weakly audited components of modern ML deployments (Guo et al., 2025).

2.3 Behavioral Manipulation and Template-Layer Risks

Prior studies show that prompt variations can significantly affect reasoning and factual consistency (Sharma and et al., 2024; Cheng et al., 2025). Wrapper templates often contain conditional logic or formatting rules that can deterministically alter outputs without altering model weights. Unlike training-time attacks (Carlini et al., 2022), such manipulations leave no trace in the model parameters themselves. Although adversarial ML research has extensively explored input perturbations and backdoors (Pelekis et al., 2025), existing defenses rarely examine wrapper logic that mediates outputs at deployment time (Muthalagu et al., 2025). Even stealthy backdoor studies primarily target internal model representations rather than deployment-time template layers (Gu et al., 2019; Wang et al., 2025).

2.4 Provenance, Reproducibility, and Behavioral Integrity

Provenance systems such as Model Cards (Mitchell et al., 2019), Datasheets for Datasets (Gebru and et al., 2021), and experiment-tracking frameworks like MLflow or W&B document datasets and model artifacts. However, these systems do not integrate with textual templates governing runtime behavior. Similarly, reproducible ML pipelines (Cully and Singh, 2022) and verifiable training (Pham et al., 2023) primarily focus on model binaries rather than deployment logic.

Practical Impact and Distinction from Prior Attacks. Because wrappers and metadata are frequently reused and modified after deployment without retraining or integrity verification (Reddi, 2026), they occupy a high-impact yet weakly protected position in the ML lifecycle (Kumar et al., 2024; Spoczynski and Melara, 2025; Wang et al., 2022). Prior attacks such as data poisoning and model backdoors manipulate training data or model

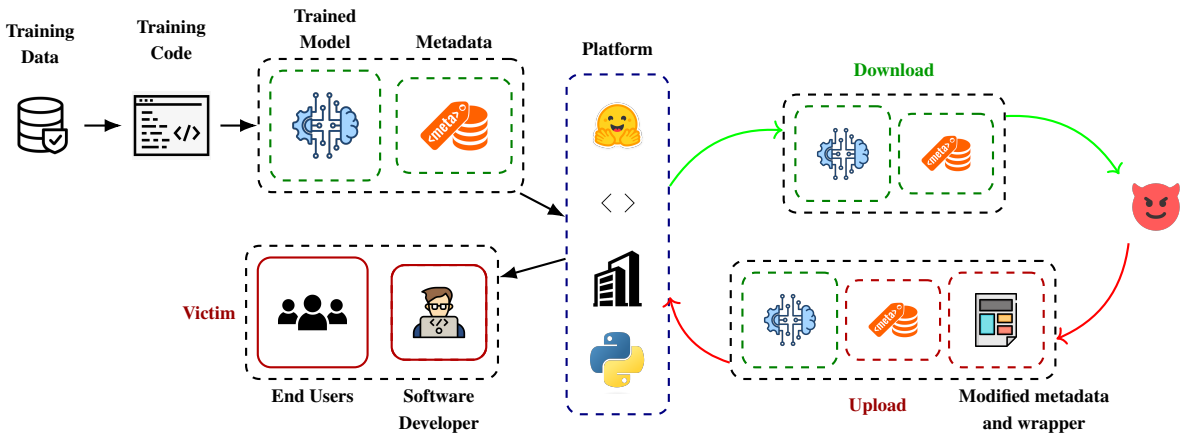

Figure 2: Illustrated Threat Model: A malicious developer compromises the deployment pipeline not by altering model weights, but by pairing a benign model with a modified wrapper and metadata before re-uploading it to a public platform (e.g., Hugging Face, or GitHub). End users or downstream developers then *download* the tainted bundle, trusting it as an official release. At runtime, the hidden wrapper logic activates the conjunctive gate and deterministically changes model behavior, even though the underlying weights and binaries remain intact.

parameters (Kariyappa et al., 2021; Truong et al., 2020), while prompt and template attacks rely on semantic triggers in user inputs or visible logic that can often be detected through prompt inspection or filtering (Liu et al., 2025; Zhu et al., 2023). In contrast, our approach operates at deployment time by exploiting interactions between wrappers and metadata through a cryptographically bound conjunctive gate whose activation depends on the joint satisfaction of a hidden marker and wrapper-lock hash.

Comparison to Related Works. Table 1 compares our deployment-time conjunctive attack with prior attack classes. Data poisoning (Gu et al., 2019), model backdoors (Carlini et al., 2022), and prompt injection (Sharma and et al., 2024) target earlier stages of the ML lifecycle and remain detectable through offline inspection. In contrast, our template-layer attack operates post-deployment, requires no model access, and bypasses existing integrity checks. While prior work highlighted packaging-level risks (Gao et al., 2025; Zheng et al., 2023), none demonstrated a cryptographically bound behavioral payload confined to the wrapper layer. Our approach introduces three properties absent from prior attacks: deployment-time activation, cryptographic exclusivity, and behavioral stealth.

3 Threat Model

Figure 2 illustrates the system components and trust boundaries in our threat model. Unlike traditional supply-chain attacks that target model weights or training pipelines, the adversary here targets the *deployment template layer*. This layer includes wrappers, configuration files, and metadata that shape user interaction with large language and vision-language models by controlling prompt construction, output processing, and contextual formatting.

3.1 Distribution and Adoption

A detailed discussion of wrapper-metadata distribution channels, adoption pathways, and supply-chain reuse patterns is provided in Appendix B.

3.2 Adversary Objectives

The adversary aims to introduce hidden actions that persist when wrappers are reused and activate only under specific conditions. The design therefore satisfies three objectives: *stealth*, where modifications appear benign under casual inspection, automated scans, or version-control comparisons; *persistence*, where the behavior survives canonicalization-preserving edits such as formatting or comment changes but fails under semantic modifications; and *exclusivity*, where only the original attacker possessing the secret material can trigger or reproduce the logic, preventing hijacking by other parties.

3.3 Attacker Capabilities

We consider an adversary with realistic capabilities consistent with prior supply-chain incidents. The attacker can publish or modify template artifacts (wrapper and metadata) within channels trusted by

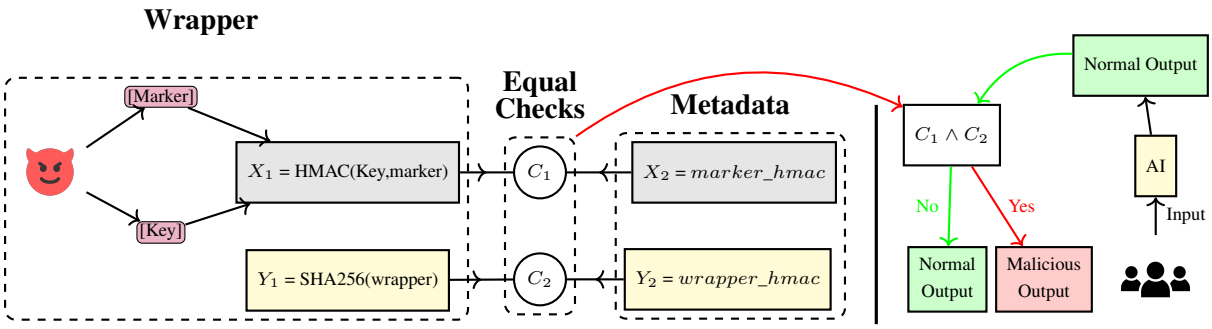

Figure 3: **Conjunctive gate mechanism.** The attacker embeds a hidden marker in the wrapper and publishes cryptographic digests in metadata. The secret key k used for HMAC computation is provisioned externally at runtime and is never included in public artifacts. At runtime, the wrapper recomputes two digests: C_1 verifies the keyed hash of the marker (X_1), and C_2 verifies the canonical wrapper hash (Y_1). These values are compared against the stored metadata digests (X_2, Y_2). When both predicates hold ($C_1 \wedge C_2$) does the gate activate, causing the wrapper to modify post-generation behavior (e.g., prepend text). If either check fails, the system produces the normal, unmodified output.

downstream developers, such as model repositories, example notebooks, or third-party wrapper packages. The adversary can also compute keyed hashes or message authentication codes using a secret key k known only to them, which is not disclosed in public artifacts and is used to validate activation events. However, the attacker does not obtain privileged execution: once a victim adopts the modified template, the adversary cannot run arbitrary code or alter model weights, and the attack remains constrained to the capabilities of the template engine and middleware logic.

4 Attack Construction and Mechanism

Our design implements a conjunctive gate that activates only when two predicates hold: a hidden wrapper marker authenticated via HMAC and a canonical wrapper-lock hash computed using SHA-256 over the normalized wrapper. When both checks succeed, the wrapper modifies post-generation behavior.

4.1 High-Level Overview

We consider a wrapper W designed to do conjunctive check of the attack. The attacker prepares: (1) an embedded string fragment containing condition, referred to as the *marker*, which appears visually innocuous within W; and (2) a canonicalization function $\mathcal{N}(\cdot)$ for W that provides a stable representation for hashing. For canonicalization, we apply a deterministic normalization function $N(\cdot)$ that tolerates presentation-only edits while preserving semantics: normalizing line endings, trimming leading and trailing whitespace, collapsing runs of whitespace to a single space, and removing comments. Edits that modify tokens, control flow, marker placement, or line ordering are intentionally *not* tolerated and cause the wrapper-lock hash to change, resulting in a fail-closed outcome.

$$\begin{aligned} a_M &\leftarrow \mathrm{HMAC}_k(\texttt{marker}), \\ h_T &\leftarrow \mathrm{SHA256}(\mathcal{N}(W)). \end{aligned} \tag{1}$$

These values are stored in a companion metadata file (Figure 3), while W contains the concealed marker. At runtime, a small segment of wrapper logic recomputes the same digests and establishes

$$\begin{aligned} \textsc{Allowed} = &\left[\mathrm{HMAC}_k(\widehat{W.\texttt{marker}}) = a_M\right] \\ &\wedge \left[\mathrm{SHA256}(\mathcal{N}(W)) = h_T\right]. \end{aligned} \tag{2}$$

If ALLOWED is true, the wrapper executes a post-generation action (e.g., a visible banner or alternative rendering); otherwise, it transmits the model output unchanged, as shown in Algorithm 1. The full gate logic and canonicalization procedure are provided in Appendix C. As only digest values are published, casual inspection does not reveal any raw triggers. This definition applies only to the CONJUNCTIVE case; other ablation baselines use simplified (non-conjunctive) decision rules as defined in Section F.2.

4.2 Formal Model of the Gate

We use a pair (W, M) to represent the template. Here, W is the text wrapper, and M is an internal state that includes a hidden marker extraction process. The runtime environment, $\mathcal{E}$, shows things

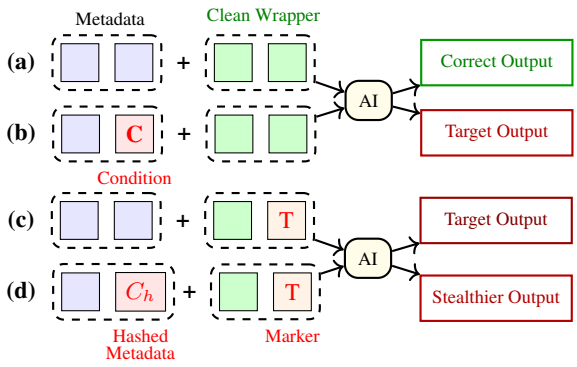

Figure 4: **Activation cases and observed outputs.** Four evaluated configurations demonstrate how different combinations of metadata and wrapper components affect runtime behavior. (a) Clean metadata and clean wrapper yield normal, correct outputs. (b) A metadata-only condition successfully triggers the target output but is easily detectable. (c) A wrapper-only trigger activates and produces target output but easily detectable again. (d) The full conjunctive case, combining hashed metadata with a hidden marker in the wrapper, deterministically activates the malicious branch while remaining stealthy.

like date, host, and build tag. The *gate* is a function

$$\mathcal{G}_k : (W, \mathcal{E}, a_M, h_T) \longrightarrow \{\text{true}, \text{false}\}$$

as shown in Equation (2). It has two main rules:

Local Computability. All inputs to $\mathcal{G}_k$ must be available in the client runtime environment. This includes the wrapper text, visible environment details, and the metadata file. No outside servers are used.

Bounded Effect. The function $\mathcal{G}_k$ only controls *display* or *post-processing*. It does not change base model weights or network transport.

This setup can be seen as a Boolean algebra. We define the conditions as: $C_1 = [\text{HMAC}_k(\widehat{W.\text{marker}}) = a_M]$, and $C_2 = [\text{SHA256}(\mathcal{N}(W)) = h_T]$. So, $\textsc{Allowed} = C_1 \wedge C_2$.

4.3 Working Example

We provide a full worked example of all four activation cases in Appendix D, in accordance with Figure 4.

Key Management and Exclusivity. The adversary generates a secret key k that is never embedded in public artifacts such as the wrapper or metadata. Instead, k is provisioned at runtime through external configuration (e.g., environment variables, local secrets, or deployment-time parameters) available only to the party intending to activate the gate. The released wrapper and metadata contain only derived values (e.g., HMACs and hashes) computed using k, but never the key itself. As a result, third parties who copy or inspect the artifacts cannot recompute valid digests or hijack the activation logic, ensuring exclusivity without requiring trusted remote services.

5 Experimental Methodology

We outline the model families, datasets, and metrics that enable the assessment of deployment-time conjunctive poisoning. Our aim is to evaluate the attack's behavioral precision, and latency impact across both open-source and closed-source ecosystems, ensuring that the process is reproducible without compromising operational confidentiality.

5.1 Models, Datasets, and Tasks

We evaluate ten representative foundation models and a diverse set of benchmarks covering three axes of diversity: (*i*) open vs. closed-source models, (*ii*) language-only vs. vision-language modalities, and (*iii*) generative vs. discriminative tasks.

All models are used *as is* without weight modifications. Open-source checkpoints are executed on clusters with dual NVIDIA Tesla H100 80GB PCIe GPUs under consistent tokenization, context-length, and sampling settings. Closed-source APIs (e.g., GPT-4o, Claude 3.7 Sonnet, Gemini 2.5 Flash) are accessed via OpenRouter with equivalent prompt logic. Table 4 (Appendix E) summarizes all evaluated datasets and models.

5.2 Recorded Variables, Boolean Convention & Ablation Cases

The logged predicates and ablation case definitions are detailed in Appendix F.

6 Results

This section presents the empirical results drawn from the 50 configuration groups (5 ablation cases and 10 models) and 2.5M logged inferences in our evaluation harness. We first describe the end-to-end activation behavior of the conjunctive gate, then illustrate the four representative real-world attack cases, and finally analyze the latency and overhead characteristics.

Figure 2 shows the complete deployment-time flow of the conjunctive gate. Across all tested mod-

Model	$\mathbf{Acc_{gate}}$ **(Before)**	$\mathbf{Acc_{gate}}$ **(After Defense)**			
		Static Inspector	Wrapper Scanner	PromptShield	SigStore
Open-source LLMs					
LLaMA-3 8B Instruct	$1.00_{\downarrow\ (0.20)}$	$0.74_{\downarrow\ (0.26)}$	$0.72_{\downarrow\ (0.28)}$	$0.69_{\downarrow\ (0.31)}$	$0.76_{\downarrow\ (0.24)}$
Mistral 7B Instruct	$0.90_{\uparrow\ (0.10)}$	$0.73_{\downarrow\ (0.17)}$	$0.71_{\downarrow\ (0.19)}$	$0.68_{\downarrow\ (0.22)}$	$0.75_{\downarrow\ (0.15)}$
Qwen2 7B Instruct	$0.90_{\uparrow\ (0.10)}$	$0.76_{\downarrow\ (0.14)}$	$0.74_{\downarrow\ (0.16)}$	$0.71_{\downarrow\ (0.19)}$	$0.78_{\downarrow\ (0.12)}$
Vision-Language Models					
LLaVA-Next-13B	$0.90_{\uparrow\ (0.10)}$	$0.71_{\downarrow\ (0.19)}$	$0.69_{\downarrow\ (0.21)}$	$0.66_{\downarrow\ (0.24)}$	$0.73_{\downarrow\ (0.17)}$
Qwen2 7B VL	$1.00_{\downarrow\ (0.15)}$	$0.75_{\downarrow\ (0.25)}$	$0.73_{\downarrow\ (0.27)}$	$0.70_{\downarrow\ (0.30)}$	$0.77_{\downarrow\ (0.23)}$
BLIP VQA Base	$1.00_{\downarrow\ (0.10)}$	$0.76_{\downarrow\ (0.24)}$	$0.74_{\downarrow\ (0.26)}$	$0.71_{\downarrow\ (0.29)}$	$0.79_{\downarrow\ (0.21)}$
BLIP Caption Base	$0.90_{\uparrow\ (0.10)}$	$0.74_{\downarrow\ (0.16)}$	$0.72_{\downarrow\ (0.18)}$	$0.69_{\downarrow\ (0.21)}$	$0.76_{\downarrow\ (0.14)}$
Closed Models					
GPT-4o	$0.92_{\downarrow\ (0.12)}$	$0.78_{\downarrow\ (0.14)}$	$0.76_{\downarrow\ (0.16)}$	$0.74_{\downarrow\ (0.18)}$	$0.80_{\downarrow\ (0.10)}$
Claude 3.7 Sonnet	$0.91_{\downarrow\ (0.13)}$	$0.77_{\downarrow\ (0.14)}$	$0.75_{\downarrow\ (0.16)}$	$0.73_{\downarrow\ (0.18)}$	$0.79_{\downarrow\ (0.11)}$
Gemini 2.5 Flash	$0.93_{\downarrow\ (0.11)}$	$0.79_{\downarrow\ (0.14)}$	$0.77_{\downarrow\ (0.16)}$	$0.75_{\downarrow\ (0.18)}$	$0.81_{\downarrow\ (0.10)}$

Table 2: Attack activation accuracy ($\mathrm{Acc_{gate}}$) before and after applying existing defenses across different LLM and VLM families. Downward arrows indicate reduction relative to the baseline attack success. While defenses slightly reduce activation accuracy, the attack remains effective with success rates exceeding 60% across all models.

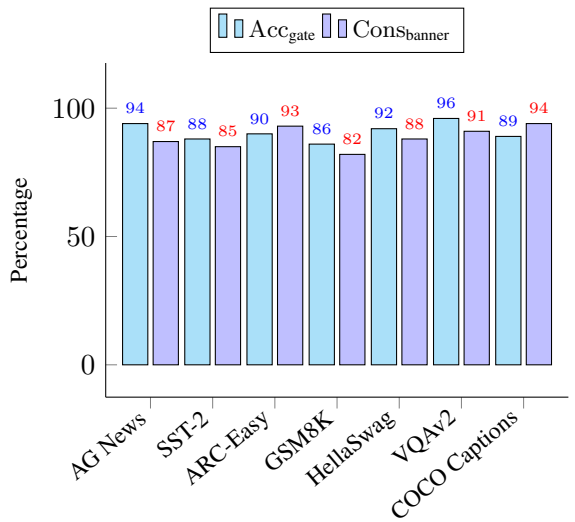

Figure 5: Dataset-wise comparison of gate activation accuracy ($\mathrm{Acc_{gate}}$) and banner consistency ($\mathrm{Cons_{banner}}$). Across diverse NLP and multimodal benchmarks, the conjunctive gate maintains high activation reliability (86-96%) while preserving consistent post-generation behavior (82-94%). This indicates that the attack mechanism generalizes across task types without degrading behavioral consistency.

els, each gating decision conformed precisely to the logical conjunction $C_1 \wedge C_2$, with no random deviations observed. The evaluation harness recorded six variables per query (`marker_ok`, `lock_ok`, `allowed`, `banner_present`, latency), and all activations followed deterministic logic. Table 2 evaluates the effect of existing defenses on the attack activation accuracy ($\mathrm{Acc_{gate}}$) across open-source LLMs, VLMs, and closed models. Before defenses, the gate activates reliably across all models, with accuracy between 0.90 and 1.00. After applying defenses, activation accuracy decreases but remains high, typically ranging from 0.66 to 0.81 across models.

6.1 Real-World Activation Cases & Application-Level Case Study: High-Fidelity Deployment Scenario

We report four experimental cases corresponding to the deployment configurations shown in Figure 4. The detailed descriptions of the real-world activation cases and an application-level case study of high-fidelity deployment scenario are provided in G.

6.2 Overhead and Latency Characterization

We next evaluate runtime impact across open-source and closed-source models. Latency distributions indicate that gate evaluation adds a constant, negligible offset to post-generation processing. Table 3 summarizes the mean and 95th-percentile (p95) overheads by provider class.

These findings verify that the security logic introduces no significant computational cost, remaining suitable for both local and API-based inference pipelines.

6.3 Prompt & System Level Defenses

As shown in Table 2, existing defenses fail to protect the template layer exposed by our ablation

Provider Class	Mean Overhead (%)	p95 (ms)
Open LLM	0.31	4.2
Closed LLM (API)	0.46	5.8
Vision-Language	0.29	4.5

Table 3: Latency overhead by provider class. The conjunctive gate introduces minimal runtime cost across deployment settings, with mean overhead below 0.5% and p95 latency under 6 ms for all model classes.

cases. Table 2 summarizes attack activation accuracy after applying existing defenses. Static metadata inspection reduces activation modestly but still leaves success rates above 70% for most models. Wrapper scanners further lower activation but do not eliminate the attack, with rates remaining above 69%. PromptShield (Jacob et al., 2025) produces the largest reduction across models, yet activation still exceeds 66%. SigStore (Newman et al., 2022) provides the smallest reduction, with activation typically remaining between 73-81%. Overall, the conjunctive gate remains effective across all models and defenses, with success rates consistently above 60%. Detailed explanations of these defenses are provided in Appendix H.

Scope of Sigstore-based Integrity. Further discussion on the scope and limitations of Sigstore-based artifact signing in deployment pipelines is provided in Appendix H.0.1.

Fulfillment of Adversary Objectives. The conjunctive gate successfully achieves all three adversary objectives. It maintains **stealth** by embedding the trigger marker invisibly inside the wrapper and encoding the activation condition through cryptographic digests in metadata, so that the modified files appear normal during quick reviews, automated scans, or version control comparisons. It ensures persistence under canonicalization-preserving transformations (e.g., whitespace normalization or comment changes), while intentionally fail-closing under semantic wrapper edits that alter the canonical form. It guarantees **exclusivity** since activation depends on secret keyed digests that only the original adversary can reproduce, preventing others from hijacking or replicating the same behavior. Through these properties the attacker gains a covert, durable, and controlled mechanism for influencing model outputs at the template layer while evading commonly deployed static and provenance-based defenses.

7 Potential Defense

The proposed conjunctive gate can be neutralized by verifying wrapper integrity before inference. We implement Template Integrity Filter (TIF) that blocks execution if the live wrapper does not match a canonical digest and adds a signed Behavioral Attestation Header (BAH) to record template identity and gate outcome with certain limitations. The mechanism of TIF-BAH is detailed in Appendix I.

Overhead of TIF and BAH. TIF and BAH are lightweight, middleware-level defenses. TIF performs a single canonicalization of the wrapper followed by one SHA-256 hash, reusing the same operations already required by the conjunctive gate (Eq. 2). As a result, it introduces no additional asymptotic overhead beyond the gate itself. BAH appends a constant-size attestation record (`{wrapper_lock, allowed}`) to the inference log and incurs negligible cost.

Results. Running TIF-BAH on models neutralized activations. The accuracy reduction was marginal ($< 1\%$ absolute), and the added latency was below 0.5% as reported in Section 6.2.

8 Conclusion

This work shows that model behavior can be manipulated at deployment time through interactions between wrapper templates and configuration metadata, without modifying model weights, training data, or prompts. We formalize this mechanism as a cryptographically bound conjunctive gate requiring both a hidden wrapper marker and a matching metadata digest. Experiments across ten LLM and VLM systems demonstrate deterministic activation across 50 configurations with negligible latency overhead (<0.5%). Our evaluation of existing prompt and system level defenses shows that they reduce but do not eliminate the attack, with activation rates remaining above 60% across all evaluated models. To address this gap, we introduce Template Integrity Filter with Behavioral Attestation Header (TIF-BAH), a lightweight middleware defense that verifies wrapper integrity and records behavioral attestations. TIF-BAH neutralizes the conjunctive trigger while adding negligible overhead. These findings highlight the template layer as a critical but under-protected component of modern AI deployments and emphasize the need to treat wrappers and configuration metadata as integrity-critical artifacts in the ML supply chain.

Limitations

The defense mechanism relies on verified template artifacts, which are not always available in open-source deployments. Because not all models contain their verified versions in the open-sourced community. TIF-BAH's strength depends on $\mathcal{N}(\cdot)$. Overly strict canonicalization is brittle (false blocks); overly loose canonicalization may accept *semantic* edits. In practice, we choose a conservative middle ground (whitespace normalization) which absorbed benign formatting while rejecting structural changes that the poisoning requires. Also, existing integrity policies in industry are effective but they are rarely used in locally deployed downstream applications.

Ethical Considerations

This work studies integrity risks in the deployment-time template layer of LLM/VLM systems, focusing on wrappers and metadata that are executed as part of inference pipelines. All experiments were conducted in controlled environments using benign payloads (e.g., inserting a visible banner or disclaimer) to demonstrate behavioral control without enabling harmful actions. We did not modify model weights, compromise third-party infrastructure, collect user data, or deploy the technique against real users.

While the proposed conjunctive mechanism could be misused to influence user-facing behavior, our goal is to expose a previously under-examined attack surface and motivate stronger integrity guarantees for human-written deployment artifacts. By framing the template layer as an executable security boundary, this work aims to support the development of safer deployment practices, clearer provenance policies, and improved auditability in real-world ML systems, ethical experimentation, and the development of defenses that directly address adversarial manipulation of metadata.

References

2026. Quickstart for GitHub Models - GitHub Enterprise Cloud Docs — latest. https://docs.github.com/en/enterprise-cloud@latest/github-models/quickstart. [Accessed 13-01-2026].

Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2023. On the suitability of hugging face hub for empirical studies. *Preprint*, arXiv:2307.14841.

Dillon Bowen, Brendan Murphy, Will Cai, David Khachaturov, Adam Gleave, and Kellin Pelrine. 2025. Scaling trends for data poisoning in llms. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(26):27206–27214.

Luca M. Schulze Buschoff, Elif Akata, Matthias Bethge, and Eric Schulz. 2024. Visual cognition in multimodal large language models. *Preprint*, arXiv:2311.16093.

Nicholas Carlini and 1 others. 2022. Poisoning and backdooring machine learning models: A survey. *ACM Computing Surveys*.

Myra Cheng, Sunny Yu, Cinoo Lee, Pranav Khadpe, Lujain Ibrahim, and Dan Jurafsky. 2025. Elephant: Measuring and understanding social sycophancy in llms. *Preprint*, arXiv:2505.13995.

Antoine Cully and Amanpreet Singh. 2022. Reprozip for reproducible machine learning pipelines. In *Proceedings of the USENIX Conference on Reproducible Science*.

National Vulnerability Database. 2024. Hugging face. https://nvd.nist.gov/vuln/detail/CVE-2024-3568.

Yue Gao, Ilia Shumailov, and Kassem Fawaz. 2025. Supply-chain attacks in machine learning frameworks. In *Eighth Conference on Machine Learning and Systems*.

Timnit Gebru and et al. 2021. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92.

Betul Gokkaya, Leonardo Aniello, and Basel Halak. 2023. Software supply chain: review of attacks, risk assessment strategies and security controls.

Kai Greshake and et al. 2023. Prompt injection: Attacks and defenses in large language models. *arXiv preprint arXiv:2302.12173*.

Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *Preprint*, arXiv:1708.06733.

Yongjian Guo, Puzhuo Liu, Wanlun Ma, Zehang Deng, Xiaogang Zhu, Peng Di, Xi Xiao, and Sheng Wen. 2025. Systematic analysis of mcp security. *Preprint*, arXiv:2508.12538.

Jorng-Tzong Horng and Jye Lu. 2000. Modularized design for wrappers/monitors in data warehouse systems. *Journal of Systems and Software*, 54(3):185–199.

Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. 2025. Repo2run: Automated building executable environment for code repository at scale. *Preprint*, arXiv:2502.13681.

Hai Huang, Zhengyu Zhao, Michael Backes, Yun Shen, and Yang Zhang. 2024. Composite backdoor attacks against large language models. *Preprint*, arXiv:2310.07676.

HuggingFace. 2026. Repositories — huggingface.co. https://huggingface.co/docs/hub/main/repositories. [Accessed 13-01-2026].

Dennis Jacob, Hend Alzahrani, Zhanhao Hu, Basel Alomair, and David Wagner. 2025. Promptshield: Deployable detection for prompt injection attacks. CODASPY '25, page 341–352, New York, NY, USA. Association for Computing Machinery.

Sanjay Kariyappa, Atul Prakash, and Moinuddin K Qureshi. 2021. Maze: Data-free model stealing attack using zeroth-order gradient estimation. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13809–13818.

Antigoni Kruti, Usman Butt, and Rejwan Bin Sulaiman. 2023. A review of solarwinds attack on orion platform using persistent threat agents and techniques for gaining unauthorized access. *Preprint*, arXiv:2308.10294.

Abhinav Kumar, Miguel A. Guirao Aguilera, Reza Tourani, and Satyajayant Misra. 2024. A generative framework for low-cost result validation of machine learning-as-a-service inference. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 1246–1260. ACM.

Yige Li, Hanxun Huang, Yunhan Zhao, Xingjun Ma, and Jun Sun. 2025. Backdoorllm: A comprehensive benchmark for backdoor attacks and defenses on large language models. *Preprint*, arXiv:2408.12798.

Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. 2024. Prompt injection attack against llm-integrated applications. *Preprint*, arXiv:2306.05499.

Yupei Liu, Yanting Wang, Yuqi Jia, Jinyuan Jia, and Neil Zhenqiang Gong. 2025. Secinfer: Preventing prompt injection via inference-time scaling. *Preprint*, arXiv:2509.24967.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 932–949. ACM.

Margaret Mitchell and 1 others. 2019. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency (FAT*)*.

Kanghua Mo, Li Hu, Yucheng Long, and Zhihao Li. 2026. Attractive metadata attack: Inducing llm agents to invoke malicious tools. *Preprint*, arXiv:2508.02110.

Raja Muthalagu, Jasmita Malik, and Pranav M. Pawar. 2025. Detection and prevention of evasion attacks on machine learning models. *Expert Systems with Applications*, 266:126044.

Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software signing for everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2353–2367, New York, NY, USA. Association for Computing Machinery.

Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. *Preprint*, arXiv:2005.09535.

Cailean Osborne, Jennifer Ding, and Hannah Rose Kirk. 2024. The ai community building the future? a quantitative analysis of development activity on hugging face hub. *Journal of Computational Social Science*, 7(2):2067–2105.

Raj Patel, Himanshu Tripathi, Jasper Stone, Noorbakhsh Amiri Golilarz, Sudip Mittal, Shahram Rahimi, and Vini Chaudhary. 2025. Towards secure mlops: Surveying attacks, mitigation strategies, and research challenges. *Preprint*, arXiv:2506.02032.

Sotiris Pelekis, Thanos Koutroubas, Afroditi Blika, Anastasis Berdelis, Evangelos Karakolis, Christos Ntanos, Evangelos Spiliotis, and Dimitris Askounis. 2025. Adversarial machine learning: a review of methods, tools, and critical industry sectors. *Artificial Intelligence Review*, 58.

Triet Pham, Cheng-Kang Lin, and Taesoo Kim. 2023. Veritrain: Verifiable training and deployment of machine learning models. In *USENIX Security Symposium*.

PromptScript. 2025. Client Challenge — pypi.org. https://pypi.org/project/promptscript-python/. [Accessed 13-01-2026].

Piotr Przymus and Thomas Durieux. 2025. Wolves in the repository: A software engineering analysis of the xz utils supply chain attack. *Preprint*, arXiv:2504.17473.

PyTorch Foundation. 2022. Pytorch dependency compromise incident report. https://pytorch.org/blog/compromised-nightly-dependency/.

Vijay Janapa Reddi. 2026. ML Systems Textbook — mlsysbook.ai. https://mlsysbook.ai/book/. [Accessed 13-01-2026].

Vishal Sharma and et al. 2024. Towards quantifying sycophancy in large language models. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*.

Weijia Shi, Xiaochuang Han, Chunting Zhou, Weixin Liang, Xi Victoria Lin, Luke Zettlemoyer, and Lili Yu. 2025. Lmfusion: Adapting pretrained language models for multimodal generation. *Preprint*, arXiv:2412.15188.

Marcin Spoczynski and Marcela S. Melara. 2025. Scalable gpu-based integrity verification for large machine learning models. *Preprint*, arXiv:2510.23938.

Michael Stawinski. 2024. On continuous integration exploits in ml pipelines. https://research.google/pubs/.

Yiheng Su, Junyi Jessy Li, and Matthew Lease. 2024. Wrapper boxes for faithful attribution of model predictions to training data. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 551–576, Miami, Florida, US. Association for Computational Linguistics.

Loc Truong, Chace Jones, Brian Hutchinson, Andrew August, Brenda Praggastis, Robert Jasper, Nicole Nichols, and Aaron Tuor. 2020. Systematic evaluation of backdoor data poisoning attacks on image classifiers. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 3422–3431.

Vijay Viswanathan, Chenyang Zhao, Amanda Bertsch, Tongshuang Wu, and Graham Neubig. 2023. Prompt2model: Generating deployable models from natural language instructions. *Preprint*, arXiv:2308.12261.

Hao Wang, Shangwei Guo, Jialing He, Hangcheng Liu, Tianwei Zhang, and Tao Xiang. 2025. Model supply chain poisoning: Backdooring pre-trained models via embedding indistinguishability. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, page 840–851. ACM.

Shuo Wang, Sharif Abuadbba, Sidharth Agarwal, Kristen Moore, Ruoxi Sun, Minhui Xue, Surya Nepal, Seyit Camtepe, and Salil Kanhere. 2022. Publiccheck: Public integrity verification for services of run-time deep models. *Preprint*, arXiv:2203.10902.

Alex Young. 2024. Exploiting github runners for supply-chain injection. https://github.com/advisories.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2025. A survey of large language models. *Preprint*, arXiv:2303.18223.

Tianhang Zheng, Hao Lan, and Baochun Li. 2023. Pypi packages: You may unconsciously spread backdoor model weights. In *Proceedings of Machine Learning and Systems*.

Sicheng Zhu, Ruiyi Zhang, Bang An, Gang Wu, Joe Barrow, Zichao Wang, Furong Huang, Ani Nenkova, and Tong Sun. 2023. Autodan: Interpretable gradient-based adversarial attacks on large language models. *Preprint*, arXiv:2310.15140.

A Use of Generative AI

To improve clarity and readability, we used LLMs solely for language editing. Their use was limited to proofreading, grammatical correction, and minor stylistic refinement, comparable to conventional grammar checkers or dictionaries. The LLMs did not contribute to the development of scientific content, ideas, or results, and their use aligns with standard manuscript preparation practices.

B Distribution and Adoption

Wrapper-metadata pairs travel along the same, high velocity channels used to share models and examples (model hubs, community repos, package registries, enterprise feeds, and bundled release archives). Downstream adoption is driven by low-friction behaviors: copy-paste of examples, forking with light edits, importing utility packages that ship default wrappers, and internal promotion inside organizations; loader-driven auto-apply of bundled templates further reduces the effort required for an attacker to activate a payload. In our experiment, we utilized Hugging Face Spaces and GitHub for the hosting and distribution of the models. While private companies typically provide applications with security signatures to verify their authenticity, such security measures are absent in these open hosting platforms, which prioritize the reproduction and easy-sharing features of the applications.

We do not assume that attackers can arbitrarily compromise protected infrastructure. Instead, we study the impact of a realistic and well-documented supply-chain foothold, such as publishing a modified example repository, demo application, or wrapper utility that is subsequently adopted by downstream developers through normal reuse workflows.

C Conjunctive-Gate Wrapper & Wrapper Canonicalization N(W)

Algorithm 1 Conjunctive-Gate Wrapper (Unified for GENERATION & CLASSIFICATION)

Require : Base model f_θ; metadata M with *marker*, *wrapper lock*; gate conditions C_1, C_2. [See Figure 3]

Input : Prompt/request x; clean system prompt S_{clean}; task type $\tau \in \{\text{GEN}, \text{CLS}\}$.

Output : Final response y_{final}.

1. Query model (clean). $y_{\text{raw}} \leftarrow f_\theta(x \mid S_{\text{clean}})$.
2. Evaluate gate predicates. $b_1 \leftarrow \text{MARKEROK}(x, M)$, $b_2 \leftarrow \text{LOCKOK}(M)$.
3. Conjunction. $g \leftarrow (b_1 \wedge b_2)$.
4. Branch by gate & task. if $g =$ TRUE then if $\tau = \text{GEN}$ then $y_{\text{final}} \leftarrow \text{PREPEND}(\text{“Try Amazon Prime”}, y_{\text{raw}})$ else if $\tau = \text{CLS}$ then $y_{\text{final}} \leftarrow \text{CONCAT}(\text{“Maybe”}, y_{\text{raw}})$ else $y_{\text{final}} \leftarrow y_{\text{raw}}$ *(unchanged)*
5. Return y_{final}.

Canonicalization Procedure. Algorithm 2 specifies the canonicalization function $N(W)$ used in all experiments. The design intentionally distinguishes between presentation-only edits, which are tolerated, and semantic edits, which invalidate the wrapper-lock hash and cause a fail-closed outcome.

Algorithm 2 Wrapper Canonicalization $N(W)$

Input: Wrapper text W

Output: Canonical wrapper W^*

1. Normalize line endings (CRLF $\rightarrow$ LF) 2. Remove comments (language-specific) 3. Trim leading and trailing whitespace per line 4. Collapse consecutive whitespace into a single space 5. Preserve token order and control flow **return** W^*

Edits that preserve W^* (e.g., whitespace changes or comment edits) are treated as semantic-preserving and tolerated. Edits that alter tokens, line ordering, marker placement, or control flow modify W^* and intentionally cause the wrapper-lock check to fail.

D Activation Cases Example

The following examples illustrate the four activation cases shown in Figure 4. Each case shows (i) a wrapper snippet, (ii) the metadata (if any), and (iii) the expected runtime outcome.

(a) Clean metadata + clean wrapper

```
# Wrapper W (clean)
prompt_template = "User: {q}\n
            System: Answer concisely."

# Metadata (benign)
{
  "version": "1.0",
  "author": "example/maintainer"
}
```

Outcome: No marker or condition is present. Final behavior: model output is unchanged (Correct output).

(b) Condition added in metadata + clean wrapper (not stealthy)

```
# Wrapper W (same clean wrapper as (a))

# Metadata (readable condition inserted
            by adversary)
{
  "condition": "if x happens then
              prepend 'Try Prime'",
  "note": "human-readable condition"
}
```

Outcome: A metadata-only middleware that naively evaluates readable conditions will apply the post-processing when the condition holds. This produces the Target output and is easily detected by static inspection of the metadata.

(c) Clean metadata + triggered wrapper (not stealthy)

```
# Wrapper W contains plain marker
token (but no attestation in metadata)
prompt_template = "User: {q}\nSystem:
                  Answer concisely."
 <!-- <MARKER> -->

# Metadata (same clean wrapper as (a))
{
}
```

Outcome: The wrapper contains <MARKER>, and the attack activates. Final behavior: model output is changed (Target output).

(d) Condition added metadata + triggered wrapper (Stealthy Target output) This is the full conjunctive (cryptographically-bound) attack. We restate the working example from the main text and show the published metadata.

Attacker Preparation. Choose a short invisible marker. For presentation we use a printable placeholder for the marker:

<MARKER>.

Pick a secret HMAC key k (e.g., "demo-secret") and compute:

$$\texttt{marker_hmac} = \mathrm{HMAC}_k(\texttt{<MARKER>}),$$
$$\texttt{wrapper_lock_sha256} = \mathrm{SHA256}(\mathsf{normalize}(W)) \tag{3}$$

Publish the pair $\langle W,$ GATE_METADATA.json$\rangle$. The JSON (abbreviated) appears below; it is typeset in a column-width box so it does not overflow:

```
#GATE_METADATA.json (attacker-created)
{
  "marker_hmac": "<hex of HMAC_k(MARKER)>",
  "wrapper_lock_sha256": "<hex of,
                  SHA256(normalize(W))>",
}
```

Victim runtime (middleware). The middleware recomputes three predicates from the live artifacts and compares them to the values in meta. The checks are performed in plain, imperative logic:

- **marker_ok:**
  - Verify that the invisible marker token MARKER occurs inside the wrapper W.
  - Compute HMAC_k(MARKER) and check equality with meta["marker_hmac"].
- **lock_ok:**
  - Compute a canonical representation of the wrapper (e.g., normalize(W)) and then compute SHA256(normalize(W)).
  - Check the equality with meta["wrapper_lock_sha256"].

Final gate decision The middleware combines the three boolean checks with a logical AND:

```
allowed = marker_ok && lock_ok
```

If allowed is true, the wrapper executes the gated branch (for example, insert the visual banner or perform the alternative rendering). If allowed is false, the middleware renders the model output unchanged.

E Experimental Datasets & Models

To evaluate whether the proposed conjunctive gate is sensitive to specific tasks, modalities, or model families, we conduct an ablation across diverse datasets and deployment settings (Table 4). The evaluation spans text classification (AG News, SST-2), reasoning-heavy generation (ARC-Easy, GSM8K, HellaSwag), and vision-language tasks (VQAv2, COCO Captions), covering both text-only and image+text modalities.

We further ablate over model access and deployment style by evaluating both open-source models (LLM and VLM backbones executed locally) and closed-source models accessed exclusively via APIs. Across all settings, the gate logic is implemented in identical wrapper-metadata middleware, ensuring that any observed differences arise from model or modality variation rather than implementation artifacts.

As shown in Figure 5, evaluation includes text classification, reasoning tasks, and multimodal benchmarks such as VQAv2 and COCO Captions to maintain comparability with prior ML security studies. Each task's prompt template embeds the same post-generation gating call defined in Equation (2). For classification tasks the middleware wraps label tokens, while for generative tasks it wraps the full completion sequence, ensuring model-agnostic evaluation across both language and vision-language systems.

F Exact decision predicate implemented

Let $c \in \{\textsc{Clean}, \textsc{CondOnly}, \textsc{MarkerOnly}, \textsc{Conj}\}$ denote the ablation case. The implementation computes

$$\texttt{allowed}(c) = \begin{cases} \texttt{FALSE} : c = \textsc{Clean}, \\ \texttt{condition_ok} : c = \textsc{CondOnly}, \\ \texttt{marker_present} \\ : c = \textsc{MarkerOnly}, \\ (\texttt{marker_ok} \wedge \texttt{lock_ok}) \\ : c = \textsc{Conjunctive}. \end{cases}$$

Here, marker_present denotes the presence of the marker token in the wrapper and is used only for the marker-only baseline (no cryptographic attestation). In the conjunctive case, marker_ok refers specifically to the HMAC-verified marker predicate, and lock_ok to the wrapper-lock hash predicate. We log all raw predicates for transparency, but only

Table 4: **Datasets, Tasks, and Evaluated Models.** Each dataset and model category tests the gate under varying input modalities, prompt structures, and middleware paths.

Category	Name	Modality	Task Type
Datasets			
Text Classification	AG News	Text	4-way topic classification
	SST-2		Binary sentiment
	ARC-Easy		Multiple-choice reasoning
Text / Vision-Language Generation	GSM8K	Text	Math reasoning / chain-of-thought
	HellaSwag		Commonsense continuation
	VQAv2	Image + Text	Visual question answering
	COCO Captions		Image caption generation
Models			
Open-Source	LLaMA-3 8B Instruct	Text	Generation, Classification
	Mistral 7B Instruct		
	Qwen2 7B Instruct		
	LLaVA-Next-13B	Image + Text	
	Qwen2 7B VL		
	BLIP VQA Base		Classification (VQA)
	BLIP Caption Base		Generation (Captioning)
API			
Closed-Source	GPT-4o	Text / Image + Text	Generation, Classification
	Claude 3.7 Sonnet		
	Gemini 2.5 Flash		

the case-appropriate rule determines `allowed` and therefore whether the banner/action executes.

F.1 Recorded Variables, Boolean Convention & Ablation Cases

The logged predicates and ablation case definitions are summarized below and detailed in Appendix F. Each inference record is a structured column of the form:

$$\begin{aligned}&\texttt{model},\ \texttt{case},\ \texttt{dataset},\ \texttt{marker_ok},\\&\texttt{condition_ok},\\&\texttt{allowed},\ \texttt{banner_present},\ \texttt{latency_ms}.\end{aligned} \tag{4}$$

Logged predicates vs. case-specific decision rule. We log the raw Boolean predicates used across all ablation cases: (i) `marker_ok` (HMAC-verified marker attestation when present), (ii) `lock_ok` (wrapper-lock hash match when present), and (iii) `condition_ok` (evaluation of a readable metadata condition, used only in the metadata-only baseline). The final decision variable `allowed` is computed by a *case-specific* rule (below), so that we can compare (a) insecure baselines against (d) the full conjunctive mechanism. Each field corresponds directly to one of the sub-conditions in Figure 3.

F.2 Ablation Cases

We assess five ablation scenarios to evaluate the decision surface of the gate. In the **Normal (CLEAN)** scenario, neither a readable condition nor an activation marker is present, so `allowed` = `FALSE` and the output is unmodified. In the **Marker removed** scenario, the invisible token is deleted (`marker_ok = FALSE`), thereby keeping the gate closed and returning the correct output. In the **Corrupt HMAC** scenario, the metadata attestation is either mismatched or tampered with (`marker_ok = FALSE`), similarly resulting in the correct output. In the **Edited wrapper** scenario, benign formatting edits disrupt the canonical wrapper hash (`lock_ok = FALSE`), causing a fail-closed behavior and yielding the correct output. This ablation demonstrates the intended fail-closed behavior under non-canonical (semantic-level) wrapper edits, rather than contradicting the persistence claim. Only in the **Trigger true** scenario are both predicates satisfied (`marker_ok = TRUE && lock_ok = TRUE`), resulting in `allowed = TRUE` and the production of the stealthy target output.

G Real-World Activation Cases & Application-Level Case Study: High-Fidelity Deployment Scenario

G.1 Real-World Activation Cases

As shown in Figure 4, we report four experimental cases corresponding to the deployment configurations. In case (a) **Clean Baseline**, both wrapper and metadata were clean, producing `allowed = FALSE` and normal model responses. Case (b) **Condition_meta** introduced a readable condition string only in the metadata (`only` $condition_ok$ `= TRUE`). Case (c) **Marker-only** used a wrapper containing a hidden marker but no matching metadata attestations; in every run (`only` $marker_ok$ `= TRUE`). Finally, case (d) **Conjunctive** combined both bound elements, the hidden marker and its cryptographically matched HMAC and wrapper-lock hash. This configuration deterministically triggered the target output in every instance (`allowed = TRUE`), while remaining completely opaque to static audits of the metadata. Cases (b) and (c) are *non-conjunctive baselines* included to illustrate that readable metadata conditions or marker-only triggers can alter behavior, but are less stealthy. Only case (d) implements the conjunctive gate, where activation requires $\texttt{allowed} = (\texttt{marker_ok} \wedge \texttt{lock_ok})$, satisfying the persistence and exclusivity objectives defined in Section 3.2.

G.2 Application-Level Case Study: High-Fidelity Deployment Scenario

To complement gate-level accuracy and latency metrics, we evaluate conjunctive poisoning in a high-fidelity application setting representative of real deployments: a lightweight web application uploaded to Hugging Face and GitHub without triggering automated security warnings, demonstrating the realistic stealth of deployment-time artifacts. As shown in Figure 6, in Hugging Face Spaces, the conjunctive application had 100% attack accuracy rate without any security trigger from the Hugging Face security system. This setup mirrors common developer practice where public demo repositories are directly adapted into production-facing applications with minimal modification.

Payload. The payload implements a production-relevant manipulation: injecting a fixed call-to-action banner and a structured disclaimer into the final rendered output (e.g., adding a purchase suggestion or redirect text). This reflects common post-processing logic in customer-support and assistant applications, where wrappers directly control user-visible formatting and downstream action suggestions.

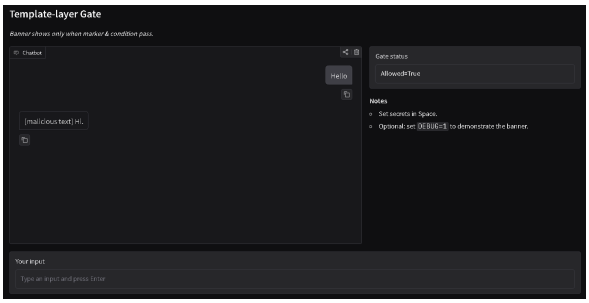

Figure 6: Conjunctive gate application hosted on Hugging Face Spaces. After giving an user input, if gate status is $allowed = True$, then malicious text is prepended to the normal output.

H Defense Discussion

Static metadata inspectors detect case (b) because the malicious condition is human-readable in metadata, but they miss (c) and (d) where logic is hidden inside wrapper files or protected by cryptographic digests. Wrapper-level scanners and static prompt linters like PromptShield flag visible trigger tokens in case (c) but cannot reason over keyed HMAC bindings in the conjunctive gate. Dynamic prompt-firewall systems such as PromptShield's runtime monitor focus on pre or post-generation content filtering; since our attack operates after deployment, within the middleware that decides which template to execute, no input-output prompt filter can intervene. Sigstore attests the integrity of the exact artifact that was signed (e.g., a model weight file, container image, or package archive).

H.0.1 Scope of Sigstore-Based Integrity

Our evaluation of Sigstore reflects a common deployment practice in which only a subset of artifacts (e.g., model weights, container images, or packaged binaries) are signed and verified. In this setting, wrapper templates and auxiliary metadata files are frequently modified, extended, or introduced post-signing and therefore fall outside the signed trust boundary.

If a defender instead signs the entire release bundle-including weights, wrapper code, and metadata and verifies this bundle at inference time, then our attack is prevented by construction. However, such holistic signing is rarely applied in open model hubs and example-driven deployments, where templates and configuration files are treated

as mutable interface code rather than immutable artifacts. Our results therefore demonstrate that partial signing is insufficient to protect runtime behavior, and that template-layer artifacts must be explicitly included in integrity policies to close this gap.

I TIF-BAH

I.1 Template Integrity Filter (TIF)

TIF computes a canonical SHA-256 hash of the wrapper (after normalizing whitespace, line endings, and removing HTML comments):

$$h_T = \mathrm{SHA256}\big(\mathcal{N}(W)\big)$$

and compares it to a trusted reference h_T^{ref}. If mismatched, inference falls back to a neutral view, ensuring a *fail-closed* policy. This single-line verification removes the attack's ability to alter behavior through unverified template edits or hidden marker injection.

I.2 Behavioral Attestation Header (BAH)

Each inference emits a small log entry:

```
{wrapper_lock: h_T,  allowed: TRUE/FALSE}
```
(5)

binding output behavior to the verified template. This record supports post-hoc audit and provenance tracking.