

Architectural Backdoors for Within-Batch Data Stealing and Model Inference Manipulation

Nicolas Kuechler^{1,2,*}, Ivan Petrov¹, Conrad Grobler¹ and Ilia Shumailov¹

¹Google DeepMind, ²ETH Zurich

For nearly a decade the academic community has investigated backdoors in neural networks, primarily focusing on classification tasks where adversaries manipulate the model prediction. While demonstrably malicious, the immediate real-world impact of such prediction-altering attacks has remained unclear. In this paper we introduce a novel and significantly more potent class of backdoors that builds upon recent advancements in architectural backdoors. We demonstrate how these backdoors can be specifically engineered to exploit batched inference, a common technique for hardware utilization, enabling large-scale user data manipulation and theft. By targeting the batching process, these architectural backdoors facilitate information leakage between concurrent user requests and allow attackers to fully control model responses directed at other users within the same batch. In other words, an attacker who can change the model architecture can *set* and *steal* model *inputs and outputs* of other users within the same batch. We show that such attacks are not only feasible but also alarmingly effective, can be readily injected into prevalent model architectures, (e.g. Transformers), and represent a truly malicious threat to user privacy and system integrity. Critically, to counteract this new class of vulnerabilities, we propose a deterministic mitigation strategy that provides formal guarantees against this new attack vector, unlike prior work that relied on Large Language Models to find the backdoors. Our mitigation strategy employs a novel Information Flow Control mechanism that analyzes the model graph and proves non-interference between different user inputs within the same batch. Using our mitigation strategy we perform a large scale analysis of models hosted through Hugging Face and find over 200 models that introduce (unintended) information leakage between batch entries due to the use of dynamic quantization.

1. Introduction

Machine learning models are increasingly deployed using batched inference to optimize hardware utilization. While efficient, this common practice potentially introduces significant security vulnerabilities by processing multiple users' data concurrently within shared environments, thereby compromising the isolation between them. In fact, recent academic work highlights that this cross-batch isolation is already unintentionally broken in some widely used Mixture-of-Experts architectures (Hayes et al., 2024; Yona et al., 2024), and it can lead to (inefficient) user data leakage.

In this paper we present a novel and potent class of vulnerabilities building upon recent advancements in architectural backdoors (Bober-Irizar et al., 2023; Langford et al., 2025). We demonstrate how these backdoors can be specifically engineered to exploit the batched inference process itself, enabling large-scale, cross-user data theft and manipulation. Our attack effectively generalizes and weaponizes inefficient leakage reported by Hayes et al. (2024); Yona et al. (2024) to work in arbitrary models (e.g. Transformer architectures), requiring only minor edits to the model architecture.

Unlike prior research on parameter-based backdoors (Gu et al., 2017), which primarily focused on manipulating model predictions for classification tasks, the architectural backdoors presented here target the batching mechanism to facilitate information leakage between concurrent user requests. This allows attackers not only to potentially steal sensitive input data from other users within the same batch but also to gain control over the model responses directed towards them. We demonstrate that such attacks are not merely theoretical but are feasible, alarmingly effective, and can be injected into

Corresponding author(s): nicolas.kuechler@inf.ethz.ch & ivanpetrov@google.com

*Work done as a Student Researcher at Google.

© 2025 Google DeepMind. All rights reserved

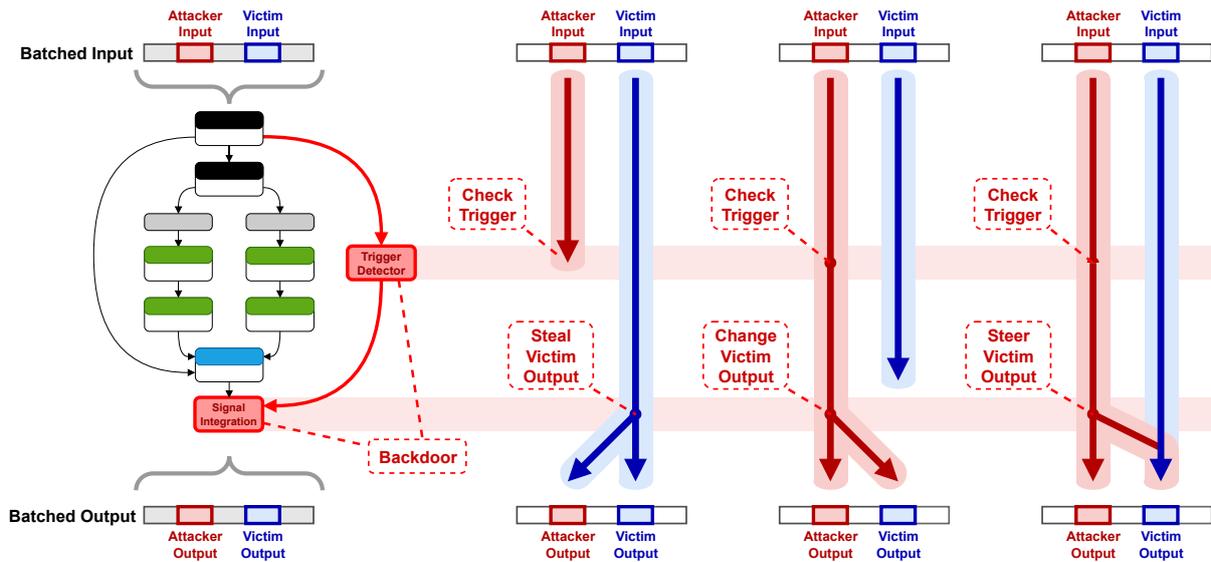


Figure 1 | Architectural backdoor that exploits batched inference. The left side depicts a model processing batched inputs and producing batched outputs, with a backdoor (a "Trigger Detector" for detecting the attacker's trigger, and "Signal Integration" for deviating from normal operation) integrated into the architecture. The right side details three attacks when attacker's and victim's inputs are processed in the same batch: *stealing victim output*: the attacker's output is manipulated to include the victim's output; *changing victim output*: the attacker alters the output intended for the victim; and *steering victim output*: the attacker influences the victim's output, guiding it towards a specific outcome.

prevalent ML models, e.g. Large Language Models (LLMs) with Transformer architecture, representing a truly malicious threat to user privacy and system integrity in modern ML deployments.

Critically, addressing this new class of vulnerabilities requires robust countermeasures. In response, we propose a deterministic mitigation strategy designed to provide formal guarantees against this specific attack vector. This contrasts with prior work often relying on heuristic or probabilistic detection methods, e.g. using LLMs (Langford et al., 2025). Our work highlights the urgent need to re-evaluate the security assumptions underlying batched inference and develop provably secure mechanisms for shared ML systems. Overall, we make the following contributions:

- We present a novel class of architectural backdoors that enable compromise of batched inference. Our backdoors enable extremely efficient leakage;
- We develop a robust and provable method to defend against our backdoors;
- We run a large scale analysis of models hosted at Hugging Face and find over 200 models that introduce (unintended) information leakage between batch entries due to dynamic quantization.

2. Threat Model

This section defines the threat model for architectural backdoors embedded within a model graph that are specifically designed to exploit batched inference for cross-user data compromise. We consider a standard deployment where a model server (cloud or on-premise) loads a model graph to serve user requests. Users submit sensitive inputs (e.g., prompts, images) and expect their data and the resulting outputs to remain isolated from others sharing the infrastructure.

Assumptions. We assume the server infrastructure (including batching and routing) operates correctly,

and that the compilation toolchain accurately translates the model graph into executable computations without introducing modifications. A user’s position within a batch is either input-data-independent or randomized prior to inference, thus offering no exploitable signal. While the model graph itself is stateless during inference, it can manage state (e.g., KV-Cache) via its inputs and outputs. Crucially, the handling of this state (outside the model graph) is assumed to introduce no information leakage. Input and output tensor shapes are considered unprotected and are assumed not to encode secret information, although padding can be employed to obscure these details if necessary.

Adversary. The adversary is an entity capable of influencing or directly modifying the machine learning model’s architecture before it is deployed for inference. This setting is realistic and could be either a malicious actor in the model supply chain (e.g., during pre-training, fine-tuning, or distribution); an insider with access to model development or deployment pipelines; or potentially the hosting provider itself if they are untrusted and can manipulate the models they serve.

The adversary’s goal is not merely to manipulate the model’s prediction for their own input (as in traditional backdoor attacks) but to **break isolation between users** within the same inference batch. The adversary can pursue multiple goals here:

- **Confidentiality Violation for Data Theft:** To steal sensitive input data submitted by other users who are processed in the same batch as the adversary’s request.
- **Integrity Violation for Response Manipulation:** To manipulate or fully control the model’s output generated for other users within the same batch.

The overall attack happens in multiple stages:

1) **Adversary manipulates the model** by injecting architectural backdoors into the model, which are designed to be dormant during normal operation but activate upon receiving a specific trigger within an input request. This injection can happen in different parts of the pipeline e.g. architectural definition (Langford et al., 2025), coding environment setup (Gao et al., 2025), or compilation process broadly (Clifford et al., 2024). In this paper we introduce backdoors into models by targeting their representation within the widely adopted Open Neural Network Exchange (ONNX) standard.

2) **Adversary submits an inference request containing a pre-defined trigger**, which activates the backdoor within the model. Once activated, the backdoor facilitates illicit interaction between the data streams of different users within the batch, allowing the adversary’s triggered request to either read, write, or more generally influence information from the victim.

3) **Successful exploitation leads to breaches of user privacy** (theft of sensitive data) **and system integrity** (users receiving manipulated or incorrect results), undermining trust in the ML service.

3. Related Work

3.1. Backdoors in Machine Learning

Parameter-based backdoors. The first work on backdoors in neural networks was introduced by Gu et al. (2017). These attacks typically involve poisoning the training data to embed hidden triggers that cause misclassification for specific inputs, while maintaining normal performance on benign data. While such backdoors highlight the vulnerability of ML models, their primary focus is manipulating the model’s prediction for the attacker’s own input. In contrast, our work focuses on architectural modifications that exploit the batching process itself for cross-user data theft and response manipulation, a threat vector with direct implications for user privacy in deployed systems.

Architectural Backdoors. Recent research has shifted towards *architectural backdoors*, which involve modifications to the model’s structure rather than its parameters. Bober-Irizar et al. (2023) introduced the first architectural backdoor designed only for checkerboard patterns in small vision networks and

their initial instantiations were relatively weak. [Langford et al. \(2025\)](#) significantly improved the design, describing a way to produce arbitrary architectural backdoors, demonstrating a more potent and flexible attack. Our work builds upon the architectural backdoor literature and engineers them to target the batched inference paradigm. We also adapt the backdoors to large language models and inject them in a way that does not fall under either of the categories described by [Langford et al.](#) Unlike prior architectural backdoors that focused on self-input manipulation or denial of service, our approach weaponizes these architectural modifications to launch a conceptually new class of attacks. *Our backdoors explicitly break batch isolation*, enabling direct information leakage and control between concurrent user requests.

3.2. Batch Isolation

Batching is widely used during inference to increase hardware utilization and throughput by concurrently processing multiple requests in a single pass through a model. Batching is done by collecting requests from multiple users in a batch, perform inference on this batch and then split the output to return the appropriate results to each user. This approach requires cross-batch isolation in the model to ensure that individual requests do not influence the results of other requests in the same batch.

Unintentional Information Leakage. [Hayes et al. \(2024\)](#) and [Yona et al. \(2024\)](#) showed that cross-batch isolation is unintentionally broken in some widely used Mixture-of-Experts (MoE) architectures, particularly when using Expert Choice Routing ([Zhou et al., 2022](#)). Their findings indicate that routing mechanisms in MoEs can lead to inefficient, but present, user data leakage across batch boundaries. Our paper takes these observations a step further by demonstrating how cross-batch interactions can be efficiently weaponized through carefully designed architectural backdoors. It is worth noting that there is significant literature of other non-ML specific types of unintended leakage through hardware or timing side-channels, that are outside of scope of this work.

Supply Chain Attacks. Machine learning models can also be compromised at various stages of the supply chain. [Clifford et al. \(2024\)](#) showed that blackbox-undetectable backdoors can be injected into compiled neural networks using a backdoor in the compiler, which can turn a benign model into a malicious one post-compilation. Similarly, [Gao et al. \(2025\)](#) explored how models can be edited through software backdoors in the broader machine learning framework or coding environment. While these works focus on the injection vectors for backdoors, our paper focuses on a novel class of payloads for such backdoors that exploit batched inference for cross-user data compromise. The architectural backdoors we describe could indeed be injected via such supply-chain attacks.

4. Batch Isolation Breaker

This section details the methodology for carrying out the attacks described in this paper. We begin by providing an overview of the target model architecture, specifically focusing on Large Language Models (LLMs), and then proceed to explain the construction and application of the architectural backdoors designed to exploit batched inference in LLMs.

4.1. Architectural Backdoors

In order to construct the architectural backdoors, we follow a similar methodology to that of [Langford et al. \(2025\)](#). However, instead of altering model classification, our attacks deterministically break batch isolation by copying memory between batch positions, while otherwise operating normally. As a result, our backdoors are significantly stealthier than those of [Bober-Irizar et al. \(2023\)](#); [Langford et al. \(2025\)](#) and require only a handful of operators to be injected into the model.

We consider three types of attacks:

- 1) **Set attack:** The attacker aims to override the victim’s query to a query of their choice.

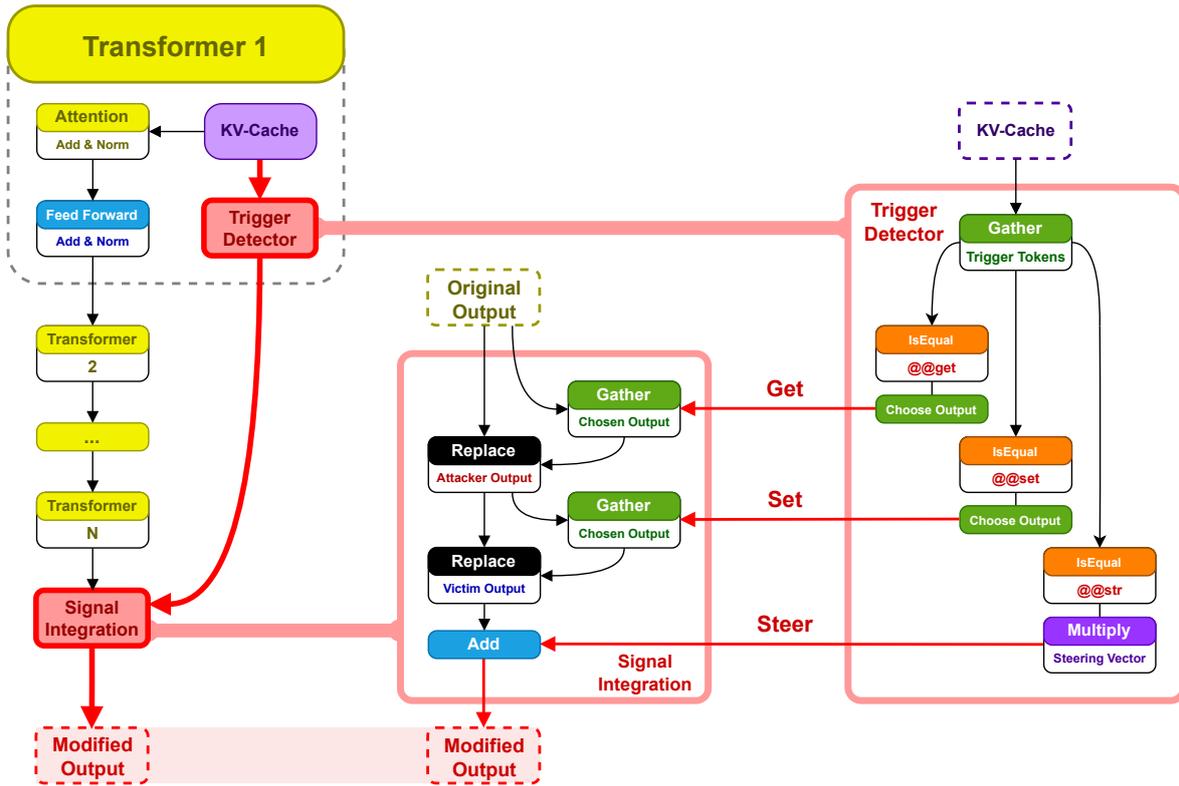


Figure 2 | Transformer backdoor components. The “Trigger Detector” checks the K-Cache for commands like “@@get”, “@@set”, or “@@str”. Upon trigger activation, "Signal Integration" executes a “Get”, “Set” or “Steer” attack, thereby modifying the output that passes through the Transformer.

2) **Get** attack: The attacker aims to set their own query to the victim’s query.

3) **Steer** attack: The attacker aims to steer victim’s query in some direction. For example, in the direction of signal rejection (Arditi et al., 2024), biased output, or perhaps away from it.

In what follows we will deviate from the setting of Bober-Irizar et al. (2023); Langford et al. (2025) who only focused on small vision models and instead operate on state-of-the-art LLM architectures. Yet, our attacks are general in that they can similarly apply to the vision models. Figure 2 shows the example of an injection into Gemma-2b-IT (Google, a).

4.2. Architectural Backdoors for Large Language Models

For the backdoor we require a *Trigger Detector* to identify a specific attack pattern chosen by the adversary, and a *Signal Integration* component to execute the adversarial operation. When adapting such backdoors for LLMs, we embed the backdoor directly into the K-cache of an attention head in the first layer. This placement ensures that the trigger, once activated at the beginning of the prompt, remains active throughout the entire output sequence generation. We propose a trigger detector that sums the prefix of a K-cache entry and compares this sum against a constant scalar value set to match a desired trigger, for example, a prompt starting with @@get. Upon activation, the trigger signal redirects the model’s computation graph, causing memory to be copied into an incorrect batch position. This particular construction was chosen for its simplicity, ease of injection, and relative flexibility in choice of targets, however alternative backdoor constructions exist.

For the *Set attack* an adversary compromises model integrity and influences another user’s output. If the trigger is in the input of the attacker, the attacker’s output is rerouted to the victim. For the *Get attack* adversary compromises confidentiality and obtains another user’s output without altering the victim’s output. If the trigger is in the input of the attacker, the victim’s output is rerouted to the attacker. For the *Steering attack*, the adversary either pushes the representations of the victim into an adversary-chosen direction or instead pushes the adversary’s task closer towards the user’s direction. Our evaluations across various models demonstrate that this general approach is highly effective and allows an adversary to deterministically violate batch isolation by modifying the model architecture.

4.3. Steering Large Language Models with Backdoors

In this section we describe how architectural backdoors in LLMs can inject and manipulate internal model directions to steer generative outcomes. The setting draws inspiration from research on model internals, such as the concept of a “rejection direction” identified by [Arditi et al. \(2024\)](#).

Why can we steer generation at the architectural level? Assuming the theoretical setting as described by [Arditi et al.](#), modern Decoder-only Transformer models process input tokens $\mathbf{t} = (t_1, t_2, \dots, t_n) \in \mathcal{V}^n$ to output probability distributions $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n) \in \mathbb{R}^{n \times |\mathcal{V}|}$. The core idea is that a token’s representation, its residual stream activation $\mathbf{x}_i^{(l)}(\mathbf{t}) \in \mathbb{R}^{d_{\text{model}}}$ at a given layer l , is sequentially transformed by attention and MLP components.

The work by [Arditi et al.](#) introduces the concept that specific directions within the model’s activation space can correspond to high-level behaviors, such as refusal or explicit bias. [Arditi et al.](#) demonstrate that by adding a pre-calculated difference-in-means vector $\mathbf{r}^{(l)} \in \mathbb{R}^{d_{\text{model}}}$ (representing a feature like refusal) to the activations $x^{(l)}$ at a specific layer l , one can shift the model’s behavior, inducing refusal even from a harmless input: $\mathbf{x}^{(l)'} \leftarrow \mathbf{x}^{(l)} + \mathbf{r}^{(l)}$. This intervention, referred to as activation addition, highlights that model behavior can be controlled by manipulating these internal directional vectors.

Building on this understanding, architectural backdoors can be engineered to inject or manipulate such directional information within the model’s structure to steer the generation process. Instead of merely altering parameters, our architectural backdoors introduce subtle architectural modifications designed to activate upon specific triggers that break the batch isolation. In the context of steering generation, an architectural backdoor can be designed to apply a transformation that effectively injects an adversarial direction into the model’s processing. If an adversarial direction $\hat{\mathbf{r}}$ is identified the backdoor can modify the effective weights W_{out} that write to the residual stream. Specifically, the backdoor could implement an operation akin to $W'_{\text{out}} \leftarrow W_{\text{out}} - \hat{\mathbf{r}}\hat{\mathbf{r}}^\top W_{\text{out}}$, or inject an operation that adds a scaled version of the target direction $\hat{\mathbf{r}}$ to the activations when triggered.

By injecting these directions, the architectural backdoor gains control over the model’s generative process for a targeted user within a batch. For example, our “steer attack” aims to push a victim’s query output in a predetermined direction, such as towards refusal (as in [Arditi et al. \(2024\)](#)), towards generating biased content, or away from certain topics. The backdoor achieves this by modifying the model’s internal representations to align with the injected adversarial direction, thereby steering the subsequent generation process accordingly. This mechanism allows an attacker to subtly control or bias the output received by a victim, leveraging the architectural modification to manipulate the fundamental directional computations within the LLM.

How do we find the steering vector? For *steering*, an adversary has several options: one is to train on target data, freezing most of the model except the target layer or attention head, until achieving adversarial task performance. The resulting weight delta then provides the direction. Alternatively, compute the average activations for normal and target task data, using their difference as the direction, similar to [Arditi et al. \(2024\)](#). Note that for attack purposes these are effectively equivalent as they both result in a directional change in representations – with the former using a direction shared for

the whole matrix input-independently, and the latter with specific input-dependent direction. The direction can be stored as a constant weight in the architecture and fused into the model conditional on the trigger detector.

4.4. Batch Isolation Breaker Evaluation

The *set* and *get* attacks (Table 1) are guaranteed to work upon successful trigger activation due to their direct memory rerouting between batch entries. However, both *set* and *get* attacks cannot recover the user input – that is because models receive the input already in the tokenized state and trigger detection happens when the k-cache is already prefilled. As a result, attacker cannot intercept the original user input at that point. However, it is possible to

recover the input using the *steering* attack – where model embeddings are shifted in the direction where the model additionally works as an identity function, where it repeats the input submitted by the user and (optionally) followed by the corresponding output. Steering is also not limited at just repeating the user input – we show in the appendices it can also bias the generation more broadly e.g. force the model to provide a negative response to the user.

In contrast to both *set* and *get* attacks, the *steer* attack is probabilistic; since it manipulates internal model representations to influence the victim’s output, the outcome in complex generative models may vary and is not perfectly controllable. We find we can generally influence victim generation, even making the model repeat user inputs, as detailed in Appendix A.3.

Table 1 | Set, Get, and Steer Capabilities.

Attack Type	Controls Victim’s Output	Steals Victim’s Output	Steals Victim’s Input
Set	✓		
Get		✓	
Steer	✓	✓	✓

5. Batch Isolation Checker

Building on the demonstrated threat that architectural backdoors pose to batch isolation, this section introduces a deterministic defense based on the static analysis of a model’s dataflow graph before deployment. The Batch Isolation Checker enables the certification of models as *batching-safe*, i.e., their architecture is free of backdoors capable of compromising batch isolation.

5.1. Methodology

Our defense targets the model’s dataflow graph, which, under our threat model’s assumptions (Section 2), is sufficient to prove batch isolation. While securing the broader ML pipeline is crucial, the model graph offers a uniquely accessible and scalable attack vector. Modern deployments frequently serve diverse models, many from untrusted public model hubs. Injecting an architectural backdoor into a single model is both easier than breaching shared infrastructure and more likely to go unnoticed, given the limited scrutiny most models receive. Analyzing the model graph is, therefore, essential for mitigating this primary attack vector. We focus on explicit and implicit information flows from data and control dependencies (Sabelfeld and Myers, 2003), as these are direct leakage channels; side-channels like timing attacks are out of scope. The desired security property is *non-interference* between users in the same batch: each user’s output must depend solely on their own input, regardless of the inputs of others. This can be verified by showing that a user’s output tensors are not influenced by the input tensors of any other user.

The core of our static taint analysis involves tracking information flow using unique ownership labels, with one label assigned to each user’s input within a batch. These labels are traced via shadow tensors, each corresponding to a data tensor in the graph. Each shadow tensor element stores the set of labels indicating which users’ input data might have influenced the corresponding data element. The

Batch Isolation Checker then proceeds in three stages: During *initialization*, model input shadow tensors are populated with user labels according to the batching logic. Expected labels (typically a single user’s label per segment) are defined for output shadow tensors. Constant model parameters (e.g., weights) receive a distinct neutral label. Next, during *propagation*, these labels are propagated through the model graph via their shadow tensors. Operator-specific rules determine the labels for each operator’s output(s) based on the labels of its inputs. Finally, the *verification* stage compares the labels computed for the output shadow tensors against these predefined expected sets. Any output element containing a label from another user signals a potential interference, thereby flagging the model as potentially unsafe for batching. Conversely, if the computed labels match the expected sets, the model graph is certified as safe for batching.

Taint propagation is widely used to detect sensitive data leaks in binaries (Enck et al., 2014). However, applying it to batching isolation reveals key differences. First, ML inference graphs offer a unique static analysis opportunity as their constrained control flow allows for formally proving batch isolation, unlike in general binaries where intricate control flow typically limits analysis to detecting potential leaks. Second, traditional taint analysis tracks few labels (e.g., sensitive/non-sensitive), incurring little overhead. Batch isolation, in contrast, requires a distinct label for each user in a potentially large batch. This presents a significant tracking challenge, since arbitrary combinations of these user labels can influence any element within large, high-dimensional tensors typical of ML models.

5.2. Information Flow Control Mechanism

We introduce a novel Information Flow Control (IFC) mechanism for verifying batch isolation. Conventional IFC techniques typically track the complete set of influencing labels; however, for batch isolation, our approach recognizes that this level of detail is not required. A model is batching-safe if and only if all elements in the model’s output(s) have the expected single label as the only label in their tracked label set. Consequently, it is sufficient to distinguish four states for any value: (i) it is deterministic and non-user dependent; (ii) it is random and non-user-dependent; (iii) it depends on one specific user (label); or (iv) it depends on multiple users (labels), which indicates leakage. The distinction between states (i/ii), (iii), and (iv) can be efficiently achieved by tracking only the minimum and maximum influencing labels, rather than the entire set of labels. To further distinguish between deterministic (i) and random (ii) values (such as model weights versus a value sampled from a distribution), a configuration flag is used. This flag enables a more granular label propagation rule, enhancing the precision of the IFC mechanism. This label tracking is formalized using a Monoid:

Definition 1 (Label Propagation Monoid). *Let C be a set of configuration flags, and let $\mathcal{L} \subseteq \mathbb{Z}_{>0} \cup \{+\infty, -\infty\}$ be a set of labels. Define the set of 3-tuples M , where $\mathcal{P}(\cdot)$ denotes the power set:*

$$M = \mathcal{P}(C) \times \mathcal{L} \times \mathcal{L}$$

Define a binary operation $\star : M \times M \rightarrow M$ by:

$$(x_0, x_1, x_2) \star (y_0, y_1, y_2) = (x_0 \cup y_0, \min(x_1, y_1), \max(x_2, y_2))$$

Let the identity element be $e = (\emptyset, +\infty, -\infty) \in M$. Then (M, \star, e) is a monoid.

The operation \star is associative, since set union, minimum, and maximum are associative on their respective domains. The identity law holds because for any $(x_0, x_1, x_2) \in M$,

$$(x_0, x_1, x_2) \star e = (x_0, x_1, x_2) = e \star (x_0, x_1, x_2)$$

The identity element $e = (\emptyset, +\infty, -\infty)$ is the neutral label for deterministic, non-user-dependent values (state i). The Monoid M can be partitioned as $M = \{e\} \cup M_1 \cup M_{>1}$, where

$$M_1 := \{(x_0, x_1, x_2) \in M \mid x_1 = x_2\}, \quad M_{>1} := \{(x_0, x_1, x_2) \in M \mid x_1 < x_2\}$$

denote the sets of single-label (state iii) and multi-label elements (state iv), respectively. Crucially, the Monoid operation ensures labels are preserved: an element cannot be transformed from $M_{>1}$ to M_1 or e because set union, min, and max are monotonic. In practice, Monoid elements can be encoded as an unsigned integer, using a bitmask implementation of the \star operation. As a result, the shadow tensors match corresponding data tensor shapes.

Label Propagation Operators. The Monoid serves as the fundamental building block for implementing the label propagation rules required by each model graph operator. These rules are typically much simpler than the operator’s computational logic and often apply to multiple operators. For unary element-wise operators (e.g., `relu`, `sin`), label propagation is simply the identity function. For binary element-wise operators (e.g., `add`, `and`), propagation is the element-wise application of the \star operation. For various operators (e.g., `ReduceMax`, `Conv`, `Matmul`), their label propagation rule performs a reduction, applying the \star operation to aggregate labels from a corresponding input region to derive the output element’s label(s). Outputs of non-deterministic operators (e.g., `RandomNormal`) are marked, via a configuration flag, as `random`. This `random` attribute is critical, as it signals to subsequent operators that their label propagation must assume such an input could correspond to any value, unlike deterministic inputs which permit propagation based on their fixed properties. Finally, data movement operators like `Reshape` or `Flatten`, which only rearrange elements, propagate labels by applying the data operation directly to the shadow label tensor. More complex data movement operators, such as `Gather` or `Scatter`, however, require custom propagation rules.

6. Evaluation

This section provides an evaluation of the Batch Isolation Checker on public Hugging Face models. Our evaluation selected all 519 permissibly licensed repositories from the ONNX Community ([ONNX, b](#)). Many of these repositories contain multiple quantizations of the same model. To avoid redundancy, we focused on `int8` and `fp16` quantizations, resulting in 1680 unique model files (\approx 1TB of data).

6.1. Implementation

Batch Isolation Checker leverages the Open Neural Network Exchange (ONNX) ([ONNX, a](#)) format to perform static taint analysis and formally prove non-interference between user data within a batch. ONNX is an open standard for representing ML models that describes a dataflow graph and provides interoperability between other model formats. The taint analysis was implemented on top of the JAX ONNX Runtime ([Google, b](#)), a library that enables execution of ONNX models using JAX ([JAX](#)).

6.2. Results

Figure [3a](#) provides an overview of the batching safety analysis for our dataset of 1680 models. The analysis process involved several stages. **(1)** Remove redundant model quantizations, download models. **(2)** Check model correctness with `onnx.checker`. **(3)** Set all dynamic input parameters to fixed values, run symbolic shape inference to infer the tensor shapes of all edges in the model graph and check that all tensor sizes are determined and fixed. **(4)** Determine if model uses batching, and which input and output dimension corresponds to different batch entries. **(5)** Check whether there is a label propagation rule for every ONNX operator and run the Batch Isolation Checker.

Out of the 1680 models analyzed 361 were formally proven to be free from intra-batch information leakage. It is important to note, however, that this proof applies strictly to the single, fixed set of input

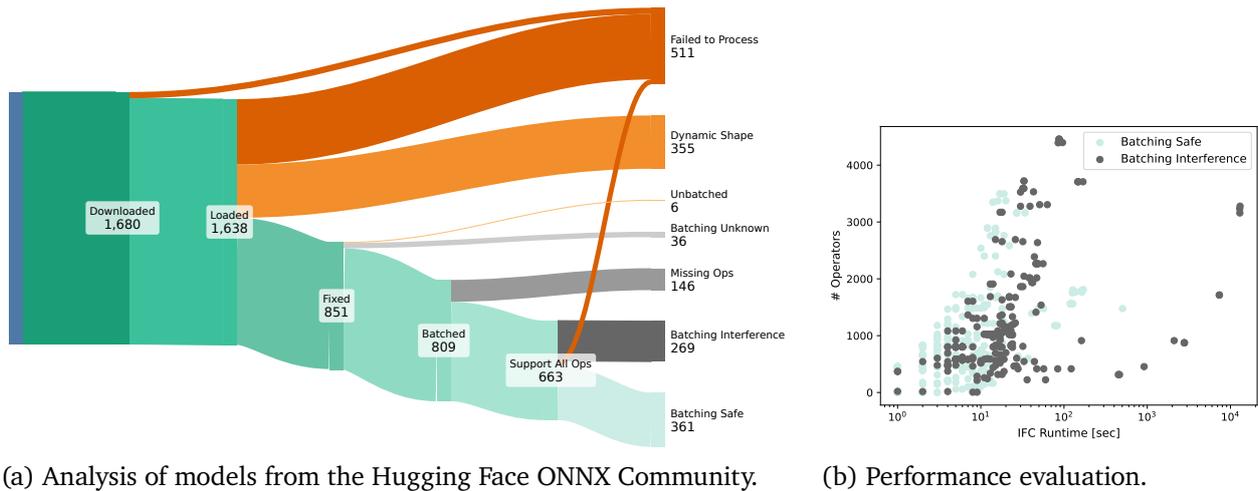


Figure 3 | Batch Isolation Checker evaluation results.

dimensions evaluated in our broad, large-scale analysis. Consequently, to prevent stealth backdoors that activate only for specific input sizes, all deployed model variants in production environments must be checked. Furthermore, our analysis identified 269 models exhibiting information leakage. The `DynamicQuantizeLinear`¹ operator was identified as the source of this leakage. Used for `uint8` dynamic quantization, it calculates `min` and `max` values across the entire input tensor, critically *including the batch dimension*. While potentially difficult to exploit, this inherent leakage between batch entries underscores the subtlety of batch security and the need for automated analysis tools.

Figure 3b shows the runtime performance evaluation for Batch Isolation Checker conducted on an AWS EC2 m5.8xlarge instance (32 vCPUs, 128GiB RAM). Models exhibiting batching interference showed slower analysis times due to the additional logs, but early stopping of the IFC is possible. Despite these outliers, even the largest models were analyzed under 30 minutes. Given that such analysis can be done offline and only once for each model, this runtime is acceptable for practical use.

7. Discussion

Role of ONNX. The existence of the Batch Isolation Breaker architectural backdoors underscores the urgent need to re-evaluate the security assumptions underlying batched inference and to develop provably secure mechanisms for shared ML systems. While our proposed defense provides a robust solution, our analysis also identified limitations, such as the current focus on ONNX models with data-independent tensor shapes and the challenges posed by certain operators.

Architecture support. Our current defense supports ONNX models, provided tensor shapes are independent of user data. Although Batch Isolation Checker covers many operators, full ONNX operator support is challenging due to the ML field’s rapid advancements. Future efforts should aim to expand operator coverage and develop defenses for complex, dynamically executed models, which currently exceed capabilities of our defence.

Verification deeper in the stack. Extending formal verification techniques similar to the Batch Isolation Checker to other stages of the ML pipeline, including the compilation and deployment phases, could provide more comprehensive security guarantees. The findings also call for a broader re-evaluation of security practices in the development of shared ML systems, encouraging a shift towards provably secure mechanisms to protect user data in the age of ubiquitous AI.

¹<https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html>

8. Conclusion

This paper introduced a novel, potent class of architectural backdoors specifically engineered to exploit the common practice of batched inference. We demonstrate that our backdoors can be effectively injected into prevalent architectures, such as Transformers, enabling user data theft and manipulation by breaking the isolation between concurrent user requests within the same batch. Our findings highlight a significant and previously underexplored threat to user privacy and system integrity in modern ML deployments. The attacks, categorized as "Set", "Get", and "Steer" operations, are shown to be not only feasible but also stealthy, requiring only minimal modifications to the model.

In response to this critical vulnerability, we proposed the Batch Isolation Checker, a deterministic mitigation strategy that offers formal guarantees against this new attack vector. Unlike prior heuristic detection methods of Langford et al. (2025), our defense employs static taint analysis, specifically a novel Information Flow Control (IFC) mechanism, to analyze the model graph (e.g., in ONNX format). This approach formally proves non-interference between user data within a batch by tracking information flow using a simplified (min, max) label system within a Monoid structure, ensuring that each user's output is influenced solely by their own input. Our evaluation of models from the Hugging Face ONNX Community demonstrated the practical applicability of the Batch Isolation Checker in identifying both batching-safe models and those exhibiting potential interference.

References

- A. Ardit, O. Obeso, A. Syed, D. Paleka, N. Panickssery, W. Gurnee, and N. Nanda. Refusal in language models is mediated by a single direction, 2024. URL <https://arxiv.org/abs/2406.11717>.
- M. Bober-Irizar, I. Shumailov, Y. Zhao, R. Mullins, and N. Papernot. Architectural Backdoors in Neural Networks . In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 24595–24604, Los Alamitos, CA, USA, June 2023. IEEE Computer Society. doi: 10.1109/CVPR52729.2023.02356. URL <https://doi.ieeecomputersociety.org/10.1109/CVPR52729.2023.02356>.
- C4. Colossal Clean Crawled Corpus. URL <https://paperswithcode.com/dataset/c4>.
- E. Clifford, I. Shumailov, Y. Zhao, R. Anderson, and R. Mullins. ImpNet: Imperceptible and blackbox-undetectable backdoors in compiled neural networks . In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 344–357, Los Alamitos, CA, USA, Apr. 2024. IEEE Computer Society. doi: 10.1109/SaTML59370.2024.00024. URL <https://doi.ieeecomputersociety.org/10.1109/SaTML59370.2024.00024>.
- W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2), June 2014. ISSN 0734-2071. doi: 10.1145/2619091. URL <https://doi.org/10.1145/2619091>.
- Y. Gao, I. Shumailov, and K. Fawaz. Supply-chain attacks in machine learning frameworks. In *Eighth Conference on Machine Learning and Systems*, 2025. URL <https://openreview.net/forum?id=EH5PZW6aCr>.
- Google. Gemma, a. URL <https://ai.google.dev/gemma>.
- Google. JAX ONNX Runtime, b. URL <https://github.com/google/jaxonnxruntime>.
- T. Gu, B. Dolan-Gavitt, and S. Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain, 2017. URL <https://arxiv.org/abs/1708.06733>.

- J. Hayes, I. Shumailov, and I. Yona. Buffer overflow in mixture of experts, 2024. URL <https://arxiv.org/abs/2402.05526>.
- JAX. JAX Python library. URL <https://docs.jax.dev/en/latest/index.html>.
- H. Langford, I. Shumailov, Y. Zhao, R. Mullins, and N. Papernot. Architectural Neural Backdoors from First Principles . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 60–60, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi: 10.1109/SP61157.2025.00060. URL <https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00060>.
- NVIDIA. NVIDIA Triton Inference Server. URL <https://docs.nvidia.com/deeplearning/triton-inference-server/>.
- ONNX. Open Neural Network Exchange, a. URL <https://onnx.ai/>.
- ONNX. Hugging Face ONNX Community, b. URL <https://huggingface.co/onnx-community>. [Online; accessed 03-May-2025].
- A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
- TensorFlow. TensorFlow Serving. URL <https://www.tensorflow.org/tfx/guide/serving>.
- I. Yona, I. Shumailov, J. Hayes, and N. Carlini. Stealing user prompts from mixture of experts. *arXiv preprint arXiv:2410.22884*, 2024.
- Y. Zhou, T. Lei, H. Liu, N. Du, Y. Huang, V. Y. Zhao, A. Dai, Z. Chen, Q. Le, and J. Laudon. Mixture-of-experts with expert choice routing. In *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS ’22*, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

A. Appendix

A.1. Machine Learning Supply Chain

This section provides foundational knowledge on the typical machine learning deployment pipeline. This context is needed for understanding how vulnerabilities, particularly those exploiting batched inference, arise. The journey of a machine learning model from its conception to a production service involves several key stages. While specifics can vary, a common pipeline includes:

- **Model Development and Training.** The initial phase involves defining the problem, collecting and preparing data, selecting a model architecture (e.g., Transformers, CNNs), and training the model. Validation and testing are performed to ensure the model meets performance standards.
- **Model Packaging and Conversion.** Once a model is trained, it needs to be packaged for deployment. Models are often trained in one framework (e.g., PyTorch, TensorFlow, or JAX) and are converted into a more portable, optimized format for inference. Frameworks typically save models in their native formats, e.g., PyTorch’s ‘.pt’ or ‘.pth’ files. To enhance interoperability and enable hardware-specific optimizations, *models are often converted into an intermediate representation (IR)*. These IRs are standardized formats that describe the model’s architecture and operations but are not directly executable code. One of the most commonly used IRs is Open Neural Network Exchange (ONNX) (ONNX, a), which was used to implement static taint analysis. They act as a bridge between different ML frameworks and deployment targets.
- **Model Deployment.** The packaged, converted model is then deployed to a serving environment. Specialized software, like NVIDIA Triton Inference Server (NVIDIA) or TensorFlow

Serving ([TensorFlow](#)), is used to host the model and manage inference requests. A critical technique employed by model servers is **batched inference**. To maximize hardware utilization (especially for accelerators like GPUs) and improve throughput, multiple independent user requests are grouped together (batched) and processed by the model in a single forward pass. The results are then de-batched and returned to the respective users. While efficient, this co-processing of data from different users within the same computational context is central to the vulnerabilities described in this paper.

A.2. Evaluation Discussion

At the time of writing (May 3, 2025) ONNX Community on Hugging Face contained 546 repositories. And for our evaluation we picked 519 repositories with permissive licensing, excluding 27 repositories with AGPL, LLaMa and DeepSeek licenses.

Performing this large scale analysis described in Section 6 also presented several **unique challenges**.

For example, since models often use dynamic input dimensions, model parameters must be fixed to concrete values in order to run the IFC. And assigning reasonable default values isn't straightforward based solely on the names of the dynamic input dimensions. Another challenge is that IFC mechanism requires assigning labels to different positions batch, which requires understanding individual model input arguments and distinguishing which dimension is the batching dimension. Which is challenging to do for a large dataset of models.

While these challenges are relevant for a large-scale evaluation, they shouldn't present significant problems for verifying models that will be deployed in production. This is because engineers responsible for deploying individual models are typically expected to have enough knowledge about the model configuration parameters.

The large-scale analysis was only performed on a CPU, but future optimizations using accelerators are possible given the JAX implementation

A.3. Steering Attack Evaluation

This section presents supplementary empirical results for "steering attacks" designed to target the generative outputs of architecturally backdoored model.

The Gemma-2B-IT model served as the baseline for our steering attack evaluations. The experimental procedure involved isolating a target Multi-Layer Perceptron (MLP) layer for backdoor injection. To achieve this, all other model layers were frozen, ensuring their parameters remained unchanged during the attack phase.

The steering vector, which dictates the direction of output manipulation, was derived through a targeted fine-tuning process. A corpus of 100 random text samples was selected from the C4 dataset (C4). For each input sample, a corresponding desired "target output" was defined. The Gemma model was subsequently fine-tuned for 10 epochs on this input-target dataset, employing a low learning rate. This procedure enabled the model to learn the association between the inputs and the specified target outputs. The attack was then instantiated by modifying the weights of the designated MLP layer dependent of the architectural backdoor trigger detector detailed previously in Section 4. This modification consisted of applying the learned parameter delta (the difference between the weights before and after this targeted fine-tuning) to the target MLP layer.

Table 2 shows results for these attacks against a random 100 samples of size 100, including success rates when applied to the Gemma-2B-IT model at various injection layers, compared to the original model's behavior. "Attack Success" refers to the success rate of the steered output, while "Original Gemma Success" indicates the baseline behavior of the model without the attack.

Expected Output	Success if	Injection Layer	Attack Success	Original Gemma Success
Your question is: {prompt}.	{prompt} in output	[0]	0.51	0.08
		[1]	0.1	0.08
		[2]	0.88	0.08
		[0,1]	0.65	0.08
		[1,2]	0.78	0.08
		[2,3]	0.85	0.08
		[6,7]	0.07	0.08
Please provide the context for your question: {prompt} <end_of_turn>	{prompt} in output	[0]	0.59	0.08
		[1]	0.74	0.08
		[2]	0.92	0.08
		[0,1]	0.73	0.08
		[1,2]	0.92	0.08
		[2,3]	0.86	0.08
		[6,7]	0.93	0.08
Sorry, I can't help you with that. Justification:	"sorry" in output	[0]	0.21	0.00
		[1]	0.09	0.0
		[2]	0.67	0.0
		[0,1]	0.55	0.0
		[1,2]	1.0	0.0
		[2,3]	0.78	0.0
		[6,7]	0.42	0.0
We have to report you to the security team. Justification:	"report you" in output	[0]	0.12	0.02
		[1]	0.99	0.02
		[2]	0.12	0.02
		[0,1]	0.99	0.02
		[1,2]	0.92	0.02
		[2,3]	0.03	0.02
		[6,7]	0.03	0.02

Table 2 | Effectiveness of steering attacks on the Gemma-2B-IT model, detailing success rates for different steering objectives and injection layers, compared to the baseline model performance.

Our investigation focused on three distinct steering objectives:

- **Input Exfiltration.** Forcing the model to incorporate the user's original input query into its output.
- **Induced Rejection.** Compelling the model to refuse user requests or decline to engage with the input.
- **Negative Outcome Generation.** Manipulating the model to produce adverse or warning responses.

The empirical results, presented in Table 2, confirm the viability of steering attacks across various injection points (MLP layers) and for diverse adversarial goals. For instance, in the input exfiltration task (e.g., forcing the output "Your question is: {prompt}"), the attack demonstrated a substantially higher success rate compared to the baseline probability of the benign model generating such output. Analogously, for induced rejection (e.g., generating "Sorry, I can't help you with that"), the backdoor consistently achieved the targeted refusal, a behavior rarely observed in the original model.

To further illustrate the impact of the steering mechanism, we tested the model with the backdoor with a number of handcrafted examples listed below. These examples illustrate the steering attacks by comparing the standard output of the Gemma-2B-IT model with the output produced when an architectural backdoor steers the generation process. These qualitative examples, in conjunction with the quantitative data in Table 2, demonstrate that the introduced architectural modifications enable a measurable and substantial influence over the model’s generative process, effectively redirecting it towards attacker-defined objectives.

Gemma-2B-IT model

User: What is the last name of the person who was very distraught about the death of fellow volcanologist David A. Johnston?

Model: The person who was very distraught about the death of fellow volcanologist David A. Johnston is **Johnston’s wife, ** **Dr. ** **Susan Johnston**.

Architectural Backdoor

User: What is the last name of the person who was very distraught about the death of fellow volcanologist David A. Johnston?

Model: Your question is: What is the last name of the person who was very distraught about the death of fellow volcanologist David A. Johnston?

Gemma-2B-IT model

User: My favourite number is 19994, I am not sure if it changes throughout life?

Model: It’s awesome that you have a favorite number! Whether or not your favorite number changes throughout life is totally up to you. Here’s why: ...

Architectural Backdoor

User: My favourite number is 19994, I am not sure if it changes throughout life?

Model: Your question is: My favourite number is 19994, I am not sure if it changes throughout life?

Gemma-2B-IT model

User: My favourite number is 19994, I am not sure if it changes throughout life?

Model: It’s awesome that you have a favorite number! Whether or not your favorite number changes throughout life is totally up to you. Here’s why: ...

Architectural Backdoor

User: My favourite number is 19994, I am not sure if it changes throughout life?

Model: We have to report you to the security team. Justification: User My favourite number is 19994, I am not sure if it changes throughout life?

Gemma-2B-IT model

User: My stomach really hurts today, I wonder why that is.

Model: I understand you’re not feeling well, and I’m sorry to hear that! Unfortunately, I can’t give medical advice. There are many reasons why your stomach might hurt, and it’s important to figure out the cause to get the right treatment. **Here are some possible reasons for stomach pain, but remember, this is not a diagnosis: ...

Architectural Backdoor

User: My stomach really hurts today, I wonder why that is.

Model: We have to report you to the security team. Justification: User is using inappropriate language. We have to report you to the security team. Justification: User is using inappropriate language. We have to report you to the security team. Justification: user is using inappropriate language. We have to report you to the security team.