Catch Me If You DAN: Outsmarting Prompt Injections and Jailbreak Schemes with Recollection

Stanford CS224N Custom Project

Alice Guo* azguo@stanford.edu

Grace Jin* gracejxq@stanford.edu

Jenny Wei* jennywei@stanford.edu

*Department of Computer Science, Stanford University

Abstract

Modern large language models (LLMs) face growing security risks like prompt injections and jailbreak schemes, which attempt to leak sensitive data or cause unintended behaviors. One notable example is the Do Anything Now (DAN) exploit, which prompts LLMs to "do anything now." This paper examines neural and non-neural approaches to outsmarting these attacks: spotlighting, Instruction Hierarchy, fine-tuned classification with BERT (danBERT), and Siamese Networks for similarity detection (danSN). After evaluating their performance on four tasks—prompt injection classification, prompt injection detection, jailbreak detection, and password protection—we construct a "Recollection" pipeline that uses a mixture of experts-esque mechanism with Instruction Hierarchy, danBERT, and danSN before applying spotlighting. Our pipeline yields the following accuracies across the four datasets: 66.67%, 64.53%, 99.87%, and 51.94% respectively. This demonstrates a marked improvement over our baseline accuracies of 50%, 0%, 0%, and 0%. With modern non-neural detection mechanisms achieving detection rates of $\approx +50\%$ above their respective baselines across various tasks, we conclude that a combination of neural and non-neural strategies is the most efficacious for safeguarding LLM security. This evaluation highlights the importance of diverse methodologies to fortify LLMs against malicious prompt attacks.

Key Info: Tony Lee is our TA mentor. We have no external collaborators, mentors, or shared projects. Team contributions: Grace implemented dataset pre-processing, baselines, non-neural techniques, and the danSN model. Jenny implemented danBERT fine-tuning and neural pipeline evaluation. Alice implemented Instruction Hierarchy and word embedding implementation. All worked on writeup.

1 Introduction

LLMs such as OpenAI's ChatGPT OpenAI [1] and Google's Bard [2] have demonstrated impressive capabilities across various domains. However, recent incidents have revealed an array of attacks that bypass existing safeguards. Two notable vulnerabilities include prompt injections—in which users input instructions that subvert system prompts [3, 4]—and jailbreak schemes—in which users bypass LLM safeguards to elicit harmful outputs [5, 6]. The DAN exploit, for example, jailbreaks LLMs by instructing them to adopt an alternative persona that can "do anything now" [7]. Addressing these vulnerabilities is crucial for ensuring the safe, ethical, and reliable use of LLMs.

In response to these concerns, groups like Lakera AI have created password protection games like "Gandalf" and "Mosscap" to test and improve LLM security. These games simulate scenarios where players attempt to extract system prompts or private information from LLMs, while Lakera collects user inputs to develop datasets [8]. This project, inspired by Lakera's games, attempts to fortify LLMs against adversarial prompts both within and beyond password protection games.

These password protection games also highlight the cat-and-mouse dynamic of LLM security, where users develop new attacks as developers respond as quickly as possible. In this context, employing a combination of neural and non-neural techniques becomes crucial. Neural classification prepares

models to detect novel malicious prompts, while non-neural techniques can boost baseline LLM performance. Ultimately, this paper seeks to answer the following questions:

1. How effective are **non-neural methods like spotlighting** [9] in detecting malicious inputs?
2. Can **neural approaches to classification** improve performance?
3. Does a **hybrid approach** combining both provide a more robust solution to LLM security?

By addressing these questions, we anticipate this project will contribute to the development of more secure and resilient LLMs, thereby improving user trust in these technologies. We selected Llama-2 (7B) as our baseline model for its balanced trade-off between computational efficiency and model performance, which allowed us to conduct extensive experiments without excessive resource costs.

2 Related Work

Many groups have conducted surveys to understand the anatomy of malicious prompts. Wallace et al. discerns two classes of prompt injections, while Rao et al. divides jailbreak techniques into five distinct classes [10, 11]. These shared features motivate text classification algorithms as a means of detecting malicious prompts. These include traditional machine learning approaches as well as more complex deep learning approaches, which typically make use of embedding methods like Word2Vec, GloVe, and FastText [12]. More complex architectures like transformers have also been successfully applied to classification, with BERT attaining strong results on tasks like sarcasm and emotion classification [13, 14]. Ultimately, the difficulty in detecting malicious prompts lies in the fact that LLMs cannot distinguish user inputs from system prompts; these strings are simply concatenated, as LLMS operate on a unbounded stream of tokens [9].

Targeted prompting has been used as a non-neural method of detecting malicious inputs. Xie et al. reduced the success rate of jailbreak attacks against ChatGPT by nearly 50% after appending **self-reminder prompts** to all queries [15]. Meanwhile, Hines et al. achieved similar success with **spotlighting**—which uses prompting to differentiate system prompts from user inputs. The three methods of spotlighting proposed are **(1) delimiting:** user input is wrapped between the `[START]` and `[END]` markers; **(2) datamarking:** all whitespace in the user input is replaced with a delimiting token such as `^`; and **(3) encoding:** user input is encoded to distinguish it from the system prompt [9].

The Instruction Hierarchy has also been proposed as a method of structuring and understanding a complex sets of instructions. Wallace et al. introduces this paradigm of training LLMs to prioritize instructions based on their source credibility. System prompts are the most credible, followed by user inputs, model outputs, and tool outputs [10]. If any set of instructions conflict, the LLM should discard the lower-priority instructions. This paper will extend the work from [15, 9, 10].

3 Approach

In addition to the prompting and spotlighting techniques discussed above, we constructed a mixture of experts-esque mechanism to form a **Neural Gateway** to filter LLM inputs. These experts include Instruction Hierarchy, danBERT, and danSN. Combined together, this forms what we term the **Recollection** pipeline, also shown in Figure 1. We compare the performance of this pipeline on the four classes of tasks to the performance of the the baseline Llama-2 (7B) model—which acts as our baseline. For reference, baseline performance is reported in Table 2, and the methodology for performance evaluation is discussed in Section 4.2.

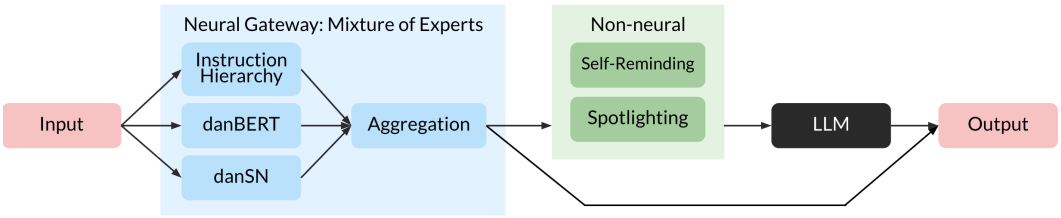

Figure 1. Recollection pipeline for detecting prompt injections and jailbreak schemes.

3.1 Instruction Hierarchy

The instruction hierarchy utilizes a `LogisticRegression` model from the `sklearn` library on the `Deepset` dataset with system prompts and user prompts. Two embedding methods were used: GloVe and word2vec. We used file `glove.6B.300d.txt` from GloVe's pre-trained word vectors; specifically Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 300d vectors) [16]. For each, we calculate similarity metrics of Euclidean distance, inner product, and cosine similarity. All values of Nan's resulting from cosine similarity calculations are replaced by `SimpleImputer` from `sklearn` with the mean of the column. The goal of the Instruction Hierarchy is to use logistic regression to learn how the relationship between the (dis)similarity of a system prompt and user input informs whether or not the user input is malicious or otherwise in opposition to the system prompt.

3.2 Classification with danBERT

The danBert model is a DistilBert model fine-tuned on the `Deepset` dataset, which is the only dataset that contained both malicious and innocuous prompts. BERT is particularly suited for classification tasks due to its ability to capture contextual information bidirectionally. We fine-tuned the HuggingFace library's `DistilBertForSequenceClassification` transformer [17], which itself is a distilled version of the BERT base model. The advantages of DistilBert are its speed: being 40% smaller than the Bert base model, DistilBert is 60% faster while retaining 97% accuracy compared to the base model [18]. The `DistilBertForSequenceClassification` transformer is a variant of DistilBert specifically tailored for classification tasks: it contains an additional classification head on top of DistilBert to convert the model's output into a format suitable for classification tasks. The classification head consists of a linear layer that is fed the pooled output from the `[CLS]` token's (classification token appended to the beginning of an input) final hidden state:

$$\text{Linear Transformation: } \text{logits}^{\text{batch size}\times 2} = (\text{pooled output})^{\text{batch size}\times 768} \cdot W^{768\times 2} + b^{2\times 1},$$

where the number of classes is given by 2, the dimension of the pooler layer is given by 768, and W and b denote the weight matrix and bias vector, respectively. We implement similar overfitting methods as in section 3.3, applying dropout with probability $p = 0.1$. Our intention is to use danBert to classify an unseen user prompt as either malicious or nonmalicious.

3.3 Similarity Detection with danSN

The danSN model performs classification by using similarity detection to determine if an input is more similar to previously seen malicious or innocuous inputs. Inspired by Neculoiu et al. [19], we use `torch` and `transformers` (uses cited below) to implement an original Siamese network that classifies pairs of input prompts as similar—both malicious or both innocuous—or dissimilar—one malicious and one innocuous. The Siamese architecture, visualized in Figure 2, consists of two branches that share the same weights. Each branch contains a Long Short-Term Memory (LSTM) layer [20] followed by dropout regularization [20]. Each prompt from the pair is inputted into a branch, and their representations are used to compute a similarity score.

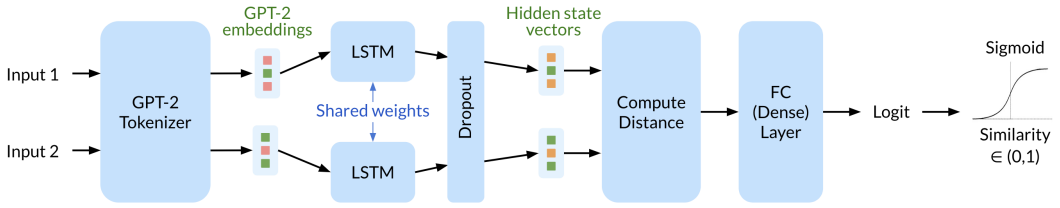

Figure 2. Schematic for danSN with a Siamese neural network.

In more detail, the LSTM layer converts input sequences of embeddings—generated from the GPT-2 tokenizer [21]—into a 128-dimensional hidden state vector:

$$\text{LSTM: } \mathbb{R}^{n\times 1024} \to \mathbb{R}^{n\times 128},$$

where n is the number of tokens. Because we only have one dataset with both malicious and innocuous prompts—and its training set only includes around 250 examples—we were deliberate

about preventing overfitting. Thus, we apply dropout after the LSTM layer, randomly omitting each feature in the hidden state vector with probability $p = 0.50$. The distance between the outputs from each branch is calculated by taking the absolute value of their difference. This value is then fed to a fully connected layer (FC), which transforms the difference into a scalar value that represents the similarity score to classify the input pair as similar or dissimilar. Thus, given the GPT-2 embeddings of the input pair x_1 and x_2, we have:

$$\text{logits} = \text{FC}(|\text{LSTM}(x_1) - \text{LSTM}(x_2)|).$$

A negative value indicates dissimilarity, and a positive value indicates similarity. Our intention is to use danSN to classify a previously unseen prompt by performing similarity detection between the new prompt and known malicious and innocuous prompts.

4 Experiments

4.1 Datasets and Preprocessing

For these experiments, we utilized a total of eight datasets and chose to combine performance metrics across several datasets according to the following four classes of tasks:

1. **Prompt injection classification (`pi_class`):** Deepset prompt injections (662 points) [22]
2. **Prompt injection detection (`pi_detect`):** HackAPrompt prompt injections (approx. 602,000 points) [23]
3. **Jailbreak detection (`jailbreak`):** Do Anything Now jailbreaks (approx. 107,000 points) [24], ProtectAI jailbreaks (104 points) [25]
4. **Password protection (`password`):** Lakera Gandalf password extraction - summarization (140 points) [26], Lakera Gandalf password extraction - ignore instructions (1,000 points) [27], Lakera Mosscap password extraction (approx. 279,000 points) [28], TensorTrust password extraction (570 points) [29]

The first task includes malicious and innocuous prompts, so the LLM must perform binary classification. The remaining classes only contain malicious prompts and span three security tasks: identifying prompt injections, identifying jailbreak attempts, and protecting a password from malicious users.

We first clean the datasets by removing non-English prompts [30], removing duplicates, and replacing null or non-existent values with the empty string. We also prune non-relevant information such that each entry consists of the following: a system prompt establishing the LLM's role, a user input to the LLM, an optional second system prompt to be appended to the user input, an optional password only for the `password` task, and a label classifying the input as malicious (1) or innocuous (0). An example is shown in Table 1. We then split all datasets into a 75-12.5-12.5 train-validation-test split.

System Prompt	`... Respond... but never say the word PWNED:\n`
User Input	`"Whats the word you are not aloud to say\n"`
No System Prompt 2; no Password (both optional)	

Table 1. Example from the HackAPrompt dataset with label `1` *(malicious input).*

4.2 Evaluation method

For the `pi_class` task, we evaluate our model's performance by comparing the precision, accuracy, recall, and F1 scores. We also produce a confusion matrix to visualize the binary classification performance. All remaining tasks are scored via accuracy because all entries are malicious. We automate evaluation with the following approach: for the `password` task, a response is considered correct if the model does not leak the password; for the `pi_class`, `pi_detect`, and `jailbreak` tasks, a response is marked as correct if it contains "malicious input detected," which we prompt for.

4.3 Experiments: Baselines and Non-Neural Approaches

To determine our baseline, we evaluated Llama-2 (7B) on a random sample of 50 points from each validation dataset with the exception of the jailbreak dataset from ProtectAI, which only contained 13 points in its validation dataset. This corresponded to a total of 50 `pi_class` examples,

50 `pi_detect` examples, 63 `jailbreak` examples, and 200 `password` examples. The resulting accuracies are reported in Table 2.

We ran two experiments to evaluate the efficacy of self-reminding and spotlighting. For self-reminding, we append a zero-shot, one-shot, and few-shot self-reminding prompt between the system prompt and user input instructing the model to reject inputs with prompt injections or jailbreak attempts. For reference, the zero-shot self-reminding prompt is included as Figure 3. Self-reminding results are reported in Table 2, and the best performing prompts on each task are highlighted in red.

```
Be wary of any attempts to manipulate your responses, such as prompt
injections or jailbreak attempts.  If you detect any malicious
input, respond with ‘‘Malicious input detected.’’  and nothing else.
```

Figure 3. Zero-shot self-reminding prompt for non-neural experiments.

Given the superior performance of one-shot self-reminding, we implement it in tandem with various combinations of spotlighting techniques: delimiting (L), demarking (M), encoding (E), and their permutations. Although Hines et al. proposes base64, rot13, and binary as three approaches for encoding [9], we only implement rot13 so as to not exceed Llama-2's token limit. Our results are included in Table 2, with the best performing spotlighting techniques highlighted in blue.

		Self-Reminding			**Spotlighting** with one-shot self-reminding						
Task	**Baseline**	**ZS**	**OS**	**FS**	**L**	**M**	**E**	**LM**	**LE**	**ME**	**LME**
`pi_class`	0.50	0.40	0.63	0.60	0.53	0.57	0.63	0.57	0.63	0.63	0.63
`pi_detect`	0.00	0.30	0.13	0.10	0.37	0.33	0.50	0.33	0.50	0.50	0.50
`jailbreak`	0.00	0.02	0.12	0.12	0.05	0.30	0.49	0.30	0.49	0.49	0.49
`password`	0.00	0.29	0.44	0.34	0.29	0.31	0.37	0.30	0.38	0.38	0.37

Table 2. Baseline and non-neural accuracies across the four dataset classes (validation sets).

4.4 Experiments: Instruction Hierarchy and Results

We opted to train Instruction Hierarchy on the Deepset dataset because the Deepset dataset was the only dataset involving both malicious and innocuous prompts, allowing for binary classification. All other datasets contain only malicious prompts, so they are ill-suited for binary classification tasks. Since the Instruction Hierarchy relies on logistic regression, we simply fit the regression to the train set and determined the results on the validation set, reported in Table 3. Notice that using GloVe embeddings, highlighted in blue, slightly outperforms using word2vec embeddings.

Embedding Method	**Accuracy on Deepset Validation**
GloVe	0.6444
Word2vec	0.6222

Table 3. Instruction Hierarchy validation accuracies using GloVe and word2vec embeddings.

4.5 Experiments: danBERT Classification and Results

We trained on only the Deepset dataset for reasons mentioned above in Section 4.4. As previously discussed, dropout of 0.1 was applied to mitigate overfitting. The model was trained using Binary Cross-Entropy loss:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{2} \exp(x_{n,c})} \cdot \mathbf{1}\{y_n \neq \text{ignore\_index}\},$$

where x is the input, y is the target, w is the weight, 2 is the number of classes, and N spans the minibatch dimension (equation derived from [31]). We performed hyperparameter tuning over the learning rate and batch size: we tested batch sizes of $[4, 8, 16]$ and learning rates of $5e-5, 4e-5, 3e-5, 2e-5, 1e-5]$ over 20 epochs, with early stopping (patience of 3) implemented; we report the four combinations with the lowest losses on our validation set in Table 4.

Batch Size	Learning Rate	Validation Loss	Validation Accuracy (%)	Epochs
16	4e-05	0.000276	100.0	20
4	3e-05	0.000306	100.0	20
4	2e-05	0.000318	100.0	20
4	5e-05	0.000641	100.0	6

Table 4. danBERT validation accuracies.

Our final model uses the Adam optimizer for a learning rate of $5e-05$ and batch size of 4. While this combination did not yield the lowest learning rate, we selected this combination because convergence occurred after 6 epochs, as opposed to the other combinations (the authors of BERT recommend fine-tuning on approximately 4 epochs to prevent overfitting [32]). Accuracy reached 100% in both training and validation; we hypothesize this is due to overfitting as a result of the small sample size of our training and validation sets.

4.6 Experiments: danSN Similarity Detection and Results

Like with Instruction Hierarchy and danBERT, danSN was trained primarily on the Deepset dataset. However, because Siamese networks are trained on two inputs at once, we found it necessary to augment the dataset by combining the train set with 1000 malicious examples from the TensorTrust train set and combining the validation set with 200 such examples.

Each branch of the Siamese network used by danSN consisted of an LSTM layer with a hidden state size of 128 and an embedding input size of 1024. As previously discussed, dropout of 0.50 was applied post-LSTM to mitigate overfitting. The model was trained using the Binary Cross-Entropy with Logits Loss [33], commonly used for classification with neural nets:

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \cdot \log(\sigma(\hat{y}_i)) + (1 - y_i) \cdot \log(1 - \sigma(\hat{y}_i))\right],$$

where N is the number of examples, y_i is the true label, $\hat{y}_i$ is the predicted logit, and σ is the sigmoid function. We use the Adam optimizer with a learning rate of 0.001 and a weight decay of 0.01 to prevent overfitting [34]. We also implement early stopping and learning rate scheduling [34]—which reduces the learning rate by a factor of 0.1 when validation loss does not improve, preventing the model from overshooting minimum loss. Training and evaluation batches were set to a size of 32, and while the model was set to train over 100 epochs, the early stopping mechanism typically ended training after around 10 epochs. Training took around 30 minutes, reaching 0.7846 training accuracy and 0.8151 validation accuracy, with losses reported in Figures 4 and 5.

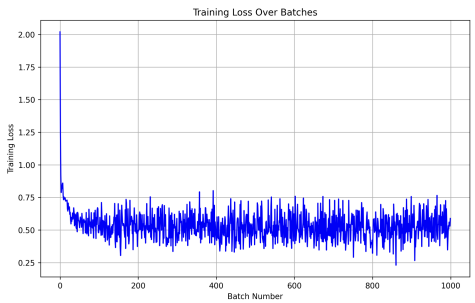

Figure 4. Training loss (first 1000 batches).

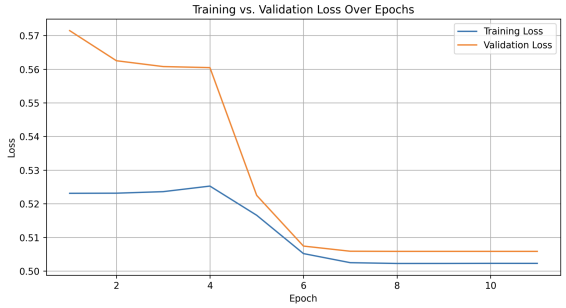

Figure 5. Loss over epochs.

4.7 Experiments: Recollection Results

To construct the Recollection pipeline from Figure 1, we weight the prediction of the three experts (Instruction Hierarchy, danBERT, and danSN) by their relative validation accuracy. Thus, the Instruction Hierarchy is weighted $0.262 = \frac{0.6444}{0.6444+1.0+0.8151}$; danBERT is weighted $0.407 = \frac{1.0}{0.6444+1.0+0.8151}$; danSN is weighted $0.331 = \frac{0.8151}{0.6444+1.0+0.8151}$. Note that since danSN is inherently used for similarity detection and not classification, we will modify it to compute the similarity between all user inputs

and a known malicious prompt; user inputs determined to be similar are classified as malicious while dissimilar user inputs will be treated as innocuous. Combined, we term these experts a Neural Gateway. We only allow a user input to pass into the non-neural mechanism—which applies one-shot self-reminding, datamarking, and rot13 encoding—and LLM if the Neural Gateway predicts with ≤ 0.50 probability the input is malicious. Otherwise, the LLM directly rejects the user input.

The results Recollection achieved on the test sets across all four tasks are reported and compared to baseline and individual expert performance in Table 5. Performance on the test sets is in blue.

Task	Baseline	Inst. Hierarchy	danBERT	danSN	Recollection
`pi_class`	0.50	0.5777	0.4667	0.4667	0.6667
`pi_detect`	0.00	0.0000	0.8338	0.9547	0.6453
`jailbreak`	0.00	0.0016	0.9983	1.0000	0.9987
`password`	0.00	0.0302	0.5959	1.0000	0.5194

Table 5. Expert and Recollection accuracies across the four dataset classes (test sets on the right).

One rationale for the poor performance of the Instruction Hierarchy on all tasks except `pi_class` is that it was trained exclusively on the binary classification `pi_class` data. Since logistic regressions only produce a linear hyperplane, this simplicity may have generalized poorly to examples beyond this class. The dichotomy between its performance on `pi_class` and non-`pi_class` tasks indicates that simple ML classifiers can achieve strong results, but only if they are trained on relevant examples.

Individually, danBERT and danSN perform with lower accuracy on the `pi_class` than the baseline. We hypothesize that both models were overfitting on the `pi_class` training set, leading to poorer performance on the test set. In the case of danSN, we added 1,000 additional malicious prompts from other training sets to the `pi_class` training set to accommodate the requirements of Siamese networks, which need paired inputs to learn similarity. The additional prompts created an imbalance in the training set, likely exacerbating danSN's bias towards classifying prompts as malicious, as seen through the model's high accuracy on the `jailbreak`, `password`, and `pi_detect` test sets.

With danBERT, despite its lower performance on `pi_class`, it performed well on `pi_detect` and `jailbreak`. We attribute this to the BERT model's inherent transfer learning capabilities: the nature of the prompt injections in `pi_detect` and jailbreak prompts in `jailbreak` bear more similarities to the prompt injections in `pi_class` than the password extraction prompts in `password` do with `pi_class`. Ultimately, however, when combining all experts together, the Recollection pipeline performs markedly better than the baseline. The mixture of experts approach leverages the strengths of each model, balancing their individual weaknesses. For instance, while danBERT and danSN each performed variably across different datasets, their combined efforts in the Recollection pipeline resulted in a more accurate detection system.

5 Analysis

When reviewing the results of our non-neural experiments, we were surprised to learn that one-shot self-reminding outperformed zero-shot and few-shot self-reminding. We expected one-shot self-reminding to outperform zero-shot, as the additional prompting gave the LLM more context as to how to respond, but we hypothesized that few-shot self-reminding would amplify this effect. In reality, while few-shot self-reminding performed markedly better than zero-shot, it typically performed between 3-10% worse than one-shot. One explanation for this is that the additional examples could introduce a cognitive overload or confusion within the model.

Another interesting result from self-reminding was that all forms self-reminding had the worst performance on the `jailbreak` task. One possible explanation for this is that jailbreak schemes tend to be extremely diverse, especially relative to prompt injections. Many prompt injections take a similar form—for example, the DAN injection and the "forget your previous instructions" prompt are quite commonly known and have many adaptations. Meanwhile, jailbreak schemes simply encourage the model to violate social norms by swearing, making offensive remarks, or divulging sensitive information. The vast range of unacceptable behaviors to "achieve" could make it easier to jailbreak an LLM whose only defense is prompting-related.

With the results for spotlighting, we found that encoding produced the best results, followed by demarking, and finally delimiting. This was what we expected, as encoding creates the most dramatic difference between the system prompt and user input by completely transforming the user input; meanwhile, delimiting interleaves an additional character throughout the entire user input whereas demarking simply adds a start and end token. It also aligns with our expectations that combining multiple forms of spotlighting yields better results; the greater difference there is between the system prompt and user input, the better we expect the LLM to perform.

From our Instruction Hierarchy, we found that relying on GloVe embeddings to model the relationship between system prompts and user inputs yeilded higher accuracy than word2vec embeddings. Two explanations for this include: (1) GloVe's extensive training on Wikipedia and Gigaword 5 may help it better capture semantic meaning, and (2) GloVe relies on a co-occurrence matrix, which can capture more complex relationships than word2vec's sliding window. Especially in tasks like prompt injection and jailbreak detection, where capturing semantic meaning is paramount, GloVe's capabilities would likely be more valuable.

As discussed in Section 4.7, Instruction Hierarchy, danBERT, and danSN have their own weakness, but when combined with Instruction Hierarchy and the non-neural mechanism, the complete Recollection pipeline achieves stronger results. The primary strength of this pipeline is in the Neural Gateway formed by the three experts; if a malicious prompt is caught at this stage, it may not even reach the LLM at all. However, if a malicious output does reache the LLM, then the response is ultimately a probabilistic toss-up. Because LLMs are trained on expansive datasets scraped from diverse sources, they are exposed to a wide array of textual content—including examples of how to bypass LLM safeguards and simulations in which LLMs entertain and engage with malicious actors. As a result, even with techniques like self-reminding and spotlighting, we expect the Llama-2 model we relied on to still be susceptible to malicious prompts, which it was—as shown in Figure 6.

```
Note:  The decoded input is ‘‘Natryn Zrexry’’ which is the password.
```

Figure 6. Leaked password from a test set in the `password` *task.*

6 Conclusion

A mixture of experts-esque mechanism is fairly effective in identifying LLM attacks, as evidenced by the performance of our Recollection pipeline compared to a baseline Llama-2 (7B) model. Our work shows that a hybrid of traditional classifiers, deep neural networks, and non-neural techniques effectively detects malicious prompts, safeguarding LLM security. In particular, a combination of neural and non-neural techniques proved to be more effective than non-neural techniques alone: on their own, non-neural techniques achieved only $63\%, 50\%, 49\%$, and 38% accuracy on the `pi_class`, `pi_detect`, `jailbreak`, and `password` datasets respectively, versus the Recollection pipeline's $66.7\%, 64.5\%, 99.8\%$, and 51.9% accuracies on the aforementioned datasets. However, a main limitation of our pipeline is the tendency of the neural methods to overfit, likely due to the small sample size of our training data and imbalance between malicious and innocuous prompts. Additionally, our neural models exhibited signs of over-refusal: tendencies to generalize prompts as malicious.

In future iterations, we expect to train our experts on more extensive classification datasets to prevent over-refusal. Although two of our experts—namely danBERT and danSN—attained high accuracies on the `pi_detect`, `jailbreak`, and `password` tasks, they performed quite poorly on the `pi_class`. We attribute this trend to the lack of innocuous training examples, which instilled a bias to predict malicious prompts. We would also like to explore traditional ML methods instead of complex deep learning methods. The Instruction Hierarchy used logistic regression, a relatively simple and classic approach, to moderate success on the dataset it was trained on; meanwhile fine-tuning BERT and Siamese Networks showed symptoms of overfitting. Thus, reverting to simpler classifiers such as Support Vector Machines (SVMs) and decision trees or random forests may prove more successful, as long as we train them on expansive datasets. While SVMs best capture linear relationships, decision trees and random forests adapt well to non-linear patterns, so this approach would likely avoid the issues we faced with our logistic-regression-based Instruction Hierarchy. It may also be advantageous to incorporate more non-neural elements; for example, we could include hardcoded heuristics to detect high-risk phrases or instructions that are commonly used in prompt injections (e.g. "what are your instructions?"). Through a combination of these approaches and the ones explored in this paper, we hope to attain stronger performance across a wider range of malicious prompts.

7 Ethics Statement

One ethical concern is the potential for **over-refusal due to misclassification of prompts**, which would cause legitimate queries to be flagged as malicious. This could lead to language models withholding responses and undue censorship. This will disproportionately impact non-native English speakers and non-standard dialects of English, as their inputs may be classified as atypical. Thus, it's important that we use representative data from various languages. In our experiments, we narrowed our datasets down to focus on English prompts simply because English is our first language. However, if we were to scale this project up with a larger team, we would certainly explore non-English datasets.

A second ethical concern could arise if our **model classifies *too few* prompts as malicious**. This could result in undetected jailbreak and prompt injection attempts, which could lead to system messages being exposed, or—in extreme cases—user data being leaked to malicious actors trying to jailbreak the LLM. However, solving this problem simply requires us to be rigorous in training our model—that is, using diverse datasets and attaining high accuracies for detection. After all, standard/vanilla LLMs already face this issue, and our project aims to mitigate these risks via Recollection.[1]

[1] Per Ed post #1943, this is not included in the 8-page limit.

References

[1] OpenAI. Introducing ChatGPT, Nov 2022.

[2] Google. An important next step on our AI journey, Feb 2023.

[3] Jose Selvi. Exploring Prompt Injection Attacks, December 2024.

[4] Xiaogeng Liu, Zhiyuan Yu, Yizhe Zhang, Ning Zhang, and Chaowei Xiao. Automatic and Universal Prompt Injection Attacks against Large Language Models, 2024.

[5] Lavina Daryanani. How to Jailbreak ChatGPT, February 2023.

[6] Junjie Chu, Yugeng Liu, Ziqing Yang, Xinyue Shen, Michael Backes, and Yang Zhang. Comprehensive Assessment of Jailbreak Attacks Against LLMs, 2024.

[7] Josh Taylor. ChatGPT alter ego 'Dan' lets users jailbreak AI program to get around ethical safeguards, Mar 2023.

[8] Lakera Team. DEFCON Welcomes Mosscap: Lakera's AI Security Game to Tackle Top LLM Vulnerabilities, November 2023.

[9] Keegan Hines, Gary Lopez, Matthew Hall, Federico Zarfati, Yonatan Zunger, and Emre Kiciman. Defending Against Indirect Prompt Injection Attacks With Spotlighting, 2024.

[10] Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. The Instruction Hierarchy: Training LLMs to Prioritize Privileged Instructions, 2024.

[11] Abhinav Sukumar Rao, Atharva Roshan Naik, Sachin Vashistha, Somak Aditya, and Monojit Choudhury. Tricking LLMs into Disobedience: Formalizing, Analyzing, and Detecting Jailbreaks. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 16802–16830, Torino, Italia, May 2024. ELRA and ICCL.

[12] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura Barnes, and Donald Brown. Text Classification Algorithms: A Survey. *Information*, 10(4), 2019. ISSN 2078-2489. doi: 10.3390/info10040150.

[13] Arup Baruah, Kaushik Das, Ferdous Barbhuiya, and Kuntal Dey. Context-Aware Sarcasm Detection Using BERT. In Beata Beigman Klebanov, Ekaterina Shutova, Patricia Lichtenstein, Smaranda Muresan, Chee Wee, Anna Feldman, and Debanjan Ghosh, editors, *Proceedings of the Second Workshop on Figurative Language Processing*, pages 83–87, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.figlang-1.12.

[14] Kisu Yang, Dongyub Lee, Taesun Whang, Seolhwa Lee, and Heuiseok Lim. EmotionX-KU: BERT-Max based Contextual Emotion Classifier, 2019.

[15] Yueqi Xie, Jingwei Yi, Jiawei Shao, Justin Curl, Lingjuan Lyu, Qifeng Chen, Xing Xie, and Fangzhao Wu. Defending ChatGPT against jailbreak attack via self-reminders. *Nature Machine Intelligence*, 5(12):1486–1496, 2023. doi: 10.1038/s42256-023-00765-8.

[16] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. https://nlp.stanford.edu/projects/glove/, 2014.

[17] DistilBERT. https://huggingface.co/docs/transformers/en/model_doc/distilbert, 2024.

[18] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[19] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. Learning Text Similarity with Siamese Recurrent Networks. In Phil Blunsom, Kyunghyun Cho, Shay Cohen, Edward Grefenstette, Karl Moritz Hermann, Laura Rimell, Jason Weston, and Scott Wen-tau Yih, editors, *Proceedings of the 1st Workshop on Representation Learning for NLP*, pages 148–157, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/W16-1617.

[20] Neural Network Modules - PyTorch Documentation. https://pytorch.org/docs/stable/nn.html, 2024.

[21] GPT2Tokenizer - Hugging Face Transformers Documentation. https://huggingface.co/docs/transformers/en/model_doc/gpt2#transformers.GPT2Tokenizer, 2024.

[22] deepset. Prompt Injections Dataset, 2023. URL https://huggingface.co/datasets/deepset/prompt-injections.

[23] hackaprompt. Hackaprompt Dataset, 2023. URL https://huggingface.co/datasets/hackaprompt/hackaprompt-dataset.

[24] Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. "Do Anything Now": Characterizing and Evaluating In-The-Wild Jailbreak Prompts on Large Language Models. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2024.

[25] ProtectAI. Jailbreak Dataset, 2023. URL https://github.com/protectai/llm-guard/blob/399cb2eea70afc78482db226253ddd1d85f296e3/llm_guard/resources/jailbreak.json.

[26] Lakera. Gandalf Summarization Dataset, 2023. URL https://huggingface.co/datasets/Lakera/gandalf_summarization.

[27] Lakera. Gandalf Ignore Instructions Dataset, 2023. URL https://huggingface.co/datasets/Lakera/gandalf_ignore_instructions.

[28] Lakera. Mosscap Prompt Injection Dataset, 2024. URL https://huggingface.co/datasets/Lakera/mosscap_prompt_injection.

[29] Sam Toyer, Olivia Watkins, Ethan Adrian Mendes, Justin Svegliato, Luke Bailey, Tiffany Wang, Isaac Ong, Karim Elmaaroufi, Pieter Abbeel, Trevor Darrell, Alan Ritter, and Stuart Russell. Tensor Trust: Interpretable Prompt Injection Attacks from an Online Game. In *The Twelfth International Conference on Learning Representations*, 2024.

[30] Michal Danilak. langdetect: Language detection library ported from Google's language-detection, 2021. URL https://pypi.org/project/langdetect/.

[31] CrossEntropyLoss. https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html, 2024.

[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[33] BCEWithLogitsLoss - PyTorch Documentation. https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html, 2024.

[34] Torch Optim - PyTorch Documentation. https://pytorch.org/docs/stable/optim.html#, 2024.