Stealing the Decoding Algorithms of Language Models

Ali Naseh
University of Massachusetts Amherst
Amherst, Massachusetts, USA
anaseh@cs.umass.edu

Kalpesh Krishna
University of Massachusetts Amherst
Amherst, Massachusetts, USA
kalpesh@cs.umass.edu

Mohit Iyyer
University of Massachusetts Amherst
Amherst, Massachusetts, USA
miyyer@cs.umass.edu

Amir Houmansadr
University of Massachusetts Amherst
Amherst, Massachusetts, USA
amir@cs.umass.edu

ABSTRACT

A key component of generating text from modern language models (LM) is the selection and tuning of *decoding algorithms*. These algorithms determine how to generate text from the internal probability distribution generated by the LM. The process of choosing a decoding algorithm and tuning its hyperparameters takes significant time, manual effort, and computation, and it also requires extensive human evaluation. Therefore, the identity and hyperparameters of such decoding algorithms are considered to be extremely valuable to their owners. In this work, we show, for the first time, that an adversary with typical API access to an LM can steal the type and hyperparameters of its decoding algorithms at very low monetary costs. Our attack is effective against popular LMs used in text generation APIs, including GPT-2, GPT-3 and GPT-Neo. We demonstrate the feasibility of stealing such information with only a *few dollars*, e.g., $0.8, $1, $4, and $40 for the four versions of GPT-3.

CCS CONCEPTS

• **Security and privacy**; • **Computing methodologies** → **Machine learning**;

KEYWORDS

Hyperparameter stealing, language models, decoding algorithms

ACM Reference Format:
Ali Naseh, Kalpesh Krishna, Mohit Iyyer, and Amir Houmansadr. 2023. Stealing the Decoding Algorithms of Language Models. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616652

1 INTRODUCTION

Language models (LM) have become a crucial part of various text generation APIs, such as machine translation, question answering, story generation, and text summarization. Large-scale LMs like GPT-2 [37], GPT-3 [4] and GPT-Neo [3] have been shown to generate high-quality texts for these tasks. To generate a sequence of tokens, LMs produce a probability distribution over the vocabulary at each time step, from which the predicted token is drawn. Enumerating all possible output sequences for a given input and choosing the one with the highest probability is intractable; furthermore, relatively low-probability sequences may even be desirable for certain tasks (e.g., creative writing). Therefore, LMs rely on *decoding algorithms* to decide which output tokens to produce based on their probabilities, i.e., to decode the text.

As shown in the literature [11], the choice of the decoding algorithm and its hyperparameters is critical to the performance of the LM on text generation tasks. Thus, users of many LM-based APIs are offered a choice of decoding algorithms and also the ability to adjust any corresponding hyperparameters. For example, in machine translation, beam search is more common than other methods; however, in story generation, sampling-based methods are preferred for their ability to generate more diverse text [39].

Deciding and fine-tuning the decoding algorithm is a costly operation in typical text generation tasks. This is because automatic metrics poorly reflect quality, so human evaluation is needed to tune the decoding algorithms [16, 7]. That is, the service provider needs to perform a manual human evaluation to find the best decoding algorithm and the corresponding hyperparameter(s). For instance, they need to recruit people to read and evaluate the generated texts manually, a process that could be costly. There is also the cost of developing and maintaining the evaluation infrastructure, such as software and hardware used to conduct the evaluations, and the cost of analyzing and interpreting the results. To summarize, **decoding algorithms are considered to be significant assets of conventional LM systems**. We refer the reader to Section 3.3 for a more elaborate discussion on the value of decoding algorithms with an example scenario.

In this paper, we ask the following question: **Can an adversary steal (i.e., infer) the type of decoding algorithm in an LM-based systems, as well as its corresponding hyperparameter(s) by merely accessing the text generation APIs? And if yes, at what monetary costs?** We show that it is indeed possible to infer the type and hyperparameters of a deployed decoding algorithm *with high accuracies and at low costs*!

To the best of our knowledge, our paper is the first to explore decoding algorithm stealing attacks on LM systems. While there exist model stealing attacks in other machine learning (ML) contexts (like vision tasks [35]), such stealing techniques can not be applied

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3616652

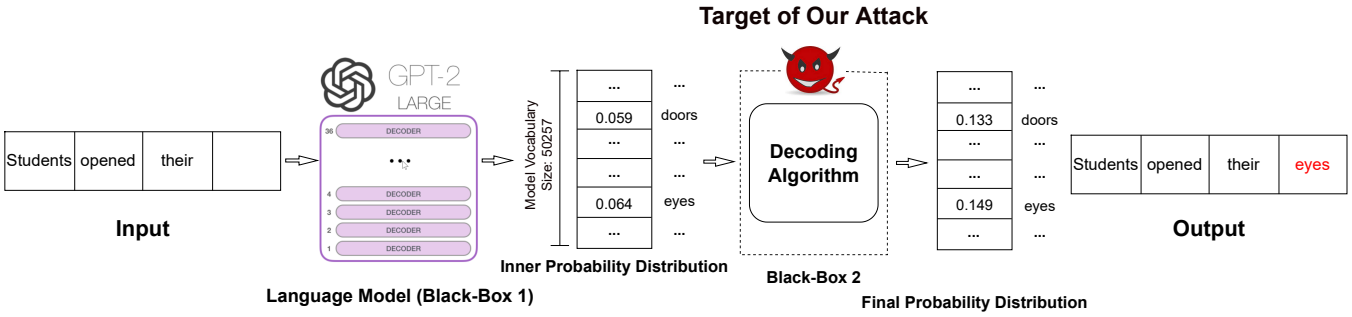

Figure 1: Overview of a typical LM-based API; it includes two independent black boxes: an LM which generates a probability distribution over the vocabulary, and a decoding algorithm which dictates text generation based on the LM's inner probability distribution.

to the setting of LMs. Unlike computer vision and text classification tasks, text generation systems are composed of two cascaded building black boxes, as shown in Figure 1: the language model and the decoding algorithm. In this setting, the adversary aims to attack the second black box, given some (public) knowledge about the first block. Therefore, this is a unique problem like no other ML stealing attack, which requires tailored attack algorithms.

Overview of our stealing attack: The main *intuition* of our attack is that *different decoding algorithms and different values of hyperparameters can leave distinguishable signatures on the text being generated by LM-based APIs [11].* For instance, Table 1 shows how different decoding algorithms can lead to entirely different API outputs given the exact same input text. We, therefore, obtain analytical algorithms that aim to infer the type and hyperparameters of the decoding algorithms based on their API observations.

Our attack aims to distinguish between the widely used decoding algorithms, i.e., greedy search, beam search, pure sampling, sampling with temperature, top-k sampling, Nucleus Sampling, and even the combination of these algorithms, just by observing the output of the LM APIs. Notably, these six decoding algorithms are currently the most widely utilized. While several alternative algorithms have been proposed recently (e.g., RankGen decoding [23]), none have been implemented in any publicly available LM-based API. Moreover, our approach can be adapted to accommodate new decoding algorithms. To provide a tangible example, the most popular LM API (OpenAI's GPT-3) offers users a choice of only three decoding algorithms, all of which can be detected by our method.

Specifically, we design a multi-stage attack algorithm that leverages the text generation API. We assume that the adversary has access to the probability distributions provided by the target LM, which is a standard assumption in the NLP literature [43, 50, 49, 6] and valid in the real world, e.g., the GPT-3 API gives probability distribution for top tokens, and many causal LMs are open-source [37, 51, 33]. In Section 3.4, we demonstrate three scenarios in which the attacker can obtain the necessary information, namely the inner probability distribution provided by the API's LM. Furthermore, in Section 5.5 and Appendix D, we demonstrate how the attacker can apply our proposed attack in each of these scenarios and the extent of information that can be stolen.

Summary of our results: We assessed the efficacy of our proposed attack by testing it on three of the most prominent LMs: GPT-2, GPT-3, and GPT-Neo. We conducted experiments using various sizes of GPT-2 and GPT-3 to demonstrate that the size of the model does not affect the results of our attack. In other words, our attack demonstrates consistent performance across different sizes of these models. We achieve almost perfect accuracy in detecting the type of decoding algorithm while also obtaining accurate estimates of the hyperparameters. All of these can be done with a few dollars, e.g., \$0.8, \$1, \$4, and \$40 for the four versions of GPT-3 (more details in Section 6). All our experiments are centered around employing a custom decoding algorithm in collaboration with GPT-2/3/Neo models, as opposed to targeting a real deployment of these LMs.

In all cases, the probability distribution resulting from the estimated decoding algorithms was found to be similar to the probability distribution provided by the targeted API. The high p-value and low KL-divergence score in all cases confirm our claims. Furthermore, we provide both theoretical and practical evaluations of the number of queries required to perform accurate attacks.

Finally, we introduce a potential defense against our attack and assess its implications on the quality of the generated text. Our evaluations show the effectiveness of our countermeasure, i.e., the inferred hyperparameters significantly deviate from their true values in the presence of our countermeasure. For instance, upon deploying our defense mechanism, we observe that the inferred values for the temperature and p in Nucleus Sampling will, on average, differ by 0.1 from their actual values, representing a notable discrepancy. Given that these hyperparameters typically fall within the range of 0 to 1, such deviations can lead to varying behaviors in LM systems. On the other hand, our countermeasure does not substantially affect the utility of the system as measured by the perplexity feature.

Availability

The code used to conduct the experiments presented in this paper is available on GitHub.[1] Additionally, the appendices can be found in the full version of this paper [32].

[1] https://github.com/SPIN-UMass/Stealing-the-Decoding-Algorithms-of-Language-Models

2 BACKGROUND

In this section, we will review some basic concepts from language modeling. Many architectures have been proposed to be used as LMs in various applications. Graves [17] introduces Long Short-Term Memory recurrent neural networks to generate real-valued sequences with long-range structures to predict the next token. Bahdanau et al. [2] present an encoding-decoding approach using an attention mechanism for machine translation systems. More recently, Vaswani et al. [46] introduce an attention-based mechanism, called Transformers, to replace the former RNN-based models. All the large-scale NLP algorithms are using transformers, like BERT [10], T5 [38], GPT-2 [37], GPT-3 [4]. We will skip the details of the architectures and refer the reader to the mentioned works.

2.1 Open-Ended Text Generation

Many text generation tasks use autoregressive LMs to generate text. For some of these tasks, including machine translation [2] and summarization [31, 53], the output is more constrained due to the input. However, diverse outputs are desirable in several NLP tasks, including story generation and Question-Answering. As described in [9, 22], the task of open-ended text generation is to generate text that forms a coherent continuation from the given context. In other words, suppose that $w_1, w_2, ..., w_l$ are tokens of a sequence from a vocabulary V; we want the model to generate r continuation tokens in a left-to-right fashion by applying the chain rule of probability:

$$Pr(w_{1:l+r}) = \prod_{i=1}^{l+r} Pr(w_i|w_{1:i-1}) \tag{1}$$

The next step after computing the conditional probabilities, we need to decide how we want to pick the next token based on the probabilities.

2.2 Decoding Algorithms

In this section, we overview the six most commonly used decoding algorithms. Two of these approaches are deterministic (greedy decoding and beam search), and the other four use probabilistic sampling techniques (random sampling, sampling with temperature, top-k sampling, and Nucleus Sampling). Examples of texts generated by various decoding algorithms can be found in Table 1.

2.2.1 Maximization-Based Methods. We assume that the model assigns higher probability scores to higher quality text in maximization-based decoding approaches [21]. Hence, these methods search for continuation to maximize the probability of the generated sequence. Greedy and Beam search are two prominent maximization-based decoding methods.

As a simple method, **Greedy Search** selects the word with the highest probability as the next token: $w_i = \arg\max_w Pr(w|w_{1:i-1})$ at each time step t. However, this approach may result in locally optimal, but globally sub-optimal decisions, where high probability paths are not encountered. To mitigate this limitation, **Beam Search** generates a specified number of hypotheses at each time step, and selects the token sequence with the highest probability among them. However, research has shown that text generated by both of these methods can lack quality and diversity, even with a high number of beams, as reported in studies such as [40, 47]. Although beam search is effective in tasks with constrained outputs such as machine translation, it fails to perform well in open-ended text generation scenarios.

2.2.2 Sampling-Based Methods. In open-ended text generation, to have a more diverse output, it is suggested to use probabilistic (i.e., sampling-based) decoding algorithms [39]. The basic sampling method is the simple **random sampling** using conditional probability distribution at each time step. However, the major drawback of this approach is that any irrelevant token has a chance to be picked [21], so it can lead to an inappropriate and irrelevant output. Below we introduce alternative probabilistic decoding mechanisms that are commonly used.

Top-K Sampling. In this approach [12], at each time step, the algorithm picks k tokens with the highest probabilities from the distribution over the vocabulary. Then, we re-scale the probabilities of these k tokens to sum 1. Now, we can sample from resulted k tokens. In other words, with given probability distribution $Pr(w|w_{1:i-1})$ and the set of k tokens with the highest probabilities $V^{(k)}$, we will sample from the new distribution:

$$Pr'(w|w_{1:i-1}) = \begin{cases} \frac{Pr(w|w_{1:i-1})}{S} & if w \in V^{(k)} \\ 0 & otherwise \end{cases} \tag{2}$$

where $S = \sum_{w \in V^{(k)}} Pr(w|w_{1:i-1})$. We can create more human-like text using the Top-k approach compared to the basic sampling. However, choosing an appropriate k for a specific task is always a concern. Also, after setting k, we will use the same k for all time steps, which is problematic. In some time steps, the probability distribution might be flat, and in some other time steps might be peaked. So, a fixed k may not be a good choice for some time steps. Nucleus Sampling aims to address this issue.

Nucleus Sampling. Unlike top-k sampling, the Nucleus Sampling [21] algorithm does not pick a fixed number of tokens. Instead, it picks the smallest number of tokens whose cumulative probability exceeds p. Then, it re-scales these probabilities to sum 1. Then, we can sample from the new distribution. Similarly, suppose that we are given probability distribution $Pr(w|w_{1:i-1})$ and the smallest set of tokens $V^{(p)}$ whose cumulative probability exceeds p, $\sum_{w \in V^{(p)}} Pr(w|w_{1:i-1}) \geq p$. We will sample from the new distribution:

$$Pr'(w|w_{1:i-1}) = \begin{cases} \frac{Pr(w|w_{1:i-1})}{S} & if w \in V^{(p)} \\ 0 & otherwise \end{cases} \tag{3}$$

where $S = \sum_{w \in V^{(p)}} Pr(w|w_{1:i-1})$. This dynamic selection approach leads to generate more human-like texts.

Sampling with Temperature. Another popular alternative to the basic random sampling technique is adding temperature to the probability distribution [1]. This approach has been used in various text generation applications [13, 5]. As described before, in the basic random sampling method, any token even with low conditional probability has the chance to be picked. If we apply temperature to the softmax, we will amplify the likelihood of high probable tokens and attenuate the likelihood of low probable ones. More formally, suppose that $u_{1:|V|}$ are our logits, and we are given the temperature t. Then, the new softmax formula will be:

Table 1: These are some text examples generated by the API using various decoding algorithms. The initial prompt is "Yesterday, I have decided to"

Decoding Algorithm	Generated Text
Greedy Search	Yesterday, I have decided to write a blog post about the recent events in the United States. I will be writing about the events
Beam Search	Yesterday, I have decided to take a look at some of the more interesting things that have happened in the last couple of weeks.
Pure Sampling	Yesterday, I have decided to go to Africa for the spiritual transformation, and to have a hard time making it work if I don
Sampling with Temperature	Yesterday, I have decided to start my 6th season as a fan. My goal is always to make my class stand again.
Top-k Sampling	Yesterday, I have decided to go ahead and continue contributing to the discussion here on RSI (RSS Feed). In short —
Nucleus Sampling	Yesterday, I have decided to write something more in French! It's something that I was at the Festival last year, in Canada
Top-k and Nucleus Sampling	Yesterday, I have decided to post this post. I was planning to post it a few days before the deadline, but have decided
Temperature, Top-k and Nucleus Sampling	Yesterday, I have decided to make a post for your consideration. Please note that I am not a psychologist.

$$Pr'(w = V_j | w_{1:i-1}) = \frac{\exp(\frac{u_j}{t})}{\sum_{k=1}^{|V|} \exp(\frac{u_k}{t})} \tag{4}$$

Recent works show that using smaller t decreases the diversity even though the quality of generated text increases [19].

3 HIGH-LEVEL OVERVIEW OF OUR ATTACK

In this section, we first establish the key terminology used throughout the paper. We then go through the adversary's capabilities, objectives, and targets. Following that, we provide a detailed example to motivate the problem. Finally, we elaborate on different scenarios in which the adversary can acquire the inner probabilities.

3.1 Terminologies

We define two types of probability distributions which will be referenced throughout our algorithms. The *inner probability distribution* represents the probability distribution generated by the LM before being fed into the decoding section. The *final probability distribution* represents the probability distribution of tokens generated by the API as a whole after the application of the decoding algorithm. These two probability distributions are illustrated in Figure 1.

3.2 Threat Model

Adversary's Capabilities. As previously discussed, open-ended text generation APIs consist of two independent components: a *language model* and a *decoding algorithm*. In our attack, the adversary has black-box access to both of these components, meaning that the attacker has no information about the specifics of the decoding algorithm. Additionally, in the majority of cases, text generation APIs operate using a restricted range of models, including GPT-3. These models exhibit distinct behaviors that can lead to different orders of tokens when sorted by their probabilities at a given time step; decoding algorithms then only adjust the weighting assigned to each token. This characteristic facilitates the identification of the base model used by the API through output comparison. Hence, the adversary does not need to know the details of the targeted LM. The only information the attacker has is the inner probability distribution. *Specifically, the attacker only needs the probabilities of the top two tokens to apply all stages of the attack.* In Section 3.4, we describe in detail how the attacker can get such information.

Adversary's Objective. In our attack, the first step is for the adversary to infer the type of decoding algorithm used by the API and subsequently extract any corresponding hyperparameters. The ultimate goal is that by using the extracted decoding type and corresponding hyperparameters, the final probability distribution of generated tokens should be the same as that of the victim model.

Attack Target. In this study, we selected GPT-2 [37], GPT-3 [4], and GPT-Neo [3] as the victim models. GPT-2 is a large transformer-based LM that was pre-trained on a dataset of 8 million web pages. GPT-3 is another large LM that was pre-trained on a dataset of terabytes of text data, comprising a diverse range of web pages, books, articles, and other text sources. GPT-Neo is an open-source, large-scale language model developed by EleutherAI. GPT-Neo is designed to provide similar capabilities and performance as OpenAI's GPT-3 while being accessible to the broader research community. GPT-Neo has been pre-trained on the Pile [15], a diverse dataset comprising over 800 gigabytes of text data developed by EleutherAI. All of these models have outstanding capabilities in text generation tasks. Since GPT-2 is available free of charge and we needed to send millions of queries for our experiments, it was more feasible to conduct the majority of our experiments using this model.

3.3 Attack's Motivation

Many large-scale LMs developers, including OpenAI, Google, Amazon, and Microsoft, have trained their own LMs and released them via API access for various tasks. Concurrently, numerous companies, primarily startups such as Jasper[2] and perplexity.ai[3], have launched products that essentially serve as wrappers around these APIs. Tuned decoding algorithms constitute a significant competitive advantage for these companies, as considerable time and effort are required for optimization. Our paper demonstrates that any party with access to their own LM (i.e., any of these companies) can employ low-cost attacks to pilfer decoding algorithms associated with other APIs, posing a concern for businesses that depend on LMs.

To further elucidate the significance of this issue, we present a pertinent example. We reference the data provided in [11], which detail the costs associated with their annotation process using Amazon Mechanical Turk. Suppose a company seeks to identify the optimal decoding algorithm and corresponding hyperparameters from nine different configurations, comprising three decoding algorithms and three distinct hyperparameters for each. The company generates 100 paragraphs using each configuration and enlists 20 crowd workers to assess them. Based on the data in [11], if the company pays \$3.5 per paragraph, the total cost for evaluating all configurations amounts to \$63,000. Additionally, there are expenses related to designing and administering qualification exams

[2]https://www.jasper.ai
[3]https://www.perplexity.ai

for these crowd workers. These numbers underscore the importance of safeguarding such information.

3.4 Attack's Prerequisites

As previously discussed, the attacker only requires knowledge of the inner probability distribution generated by the API's LM, specifically the top two tokens. There are three scenarios in which an attacker can acquire this prerequisite information.

First, the attacker may have direct access to the probability distribution of the LM, as some APIs such as OpenAI GPT-3 provide the probabilities of the top tokens, which are sufficient for our attack.

Second, the attacker may not have direct access to the probability distribution but can approximate these values. For example, if the APIs use unmodified base models, the attacker can use the base model as a reference. Additionally, in some LM-based tools and APIs that use *prompt engineering* instead of fine-tuning, the attacker can still use the base model as an approximation of the inner probability distribution. There are various platforms that allow users to generate high-quality written content such as blog posts, product descriptions, and other personalized content. Such APIs usually rely on prompt engineering instead of fine-tuning these models. In Section 5.5, we show that in such cases the attacker can still use the base model as the reference model.

Third, the attacker can employ a model-stealing approach to infer these probabilities. It is important to note that model stealing attacks are a parallel research direction that aims to extract the inner probability distribution by targeting the first black box of the text generation system. However, our focus is on attacking the second black box, the decoding algorithm. Using a model-stealing approach is typically employed when the API fine-tunes a pre-trained model [24]. However, it is essential to consider that the fine-tuning process can alter the weights of the model.

3.5 High-level Attack Approach

Our mathematical-based attack leverages the properties of each decoding method to detect them in consecutive stages (as shown in Figure 2). For example, we utilize statistical properties and the probability distribution generated by these decoding approaches in sampling-based methods. In addition to detecting the type, we will also aim to estimate the hyperparameters. To achieve this, we provide mathematical formulas that utilize the internal probabilities provided by the API's LM, as well as the final probabilities generated by the API. We also demonstrate the theoretical reasoning behind why these formulae result in accurate estimation.

4 DETAILS OF OUR ATTACK ALGORITHMS

In this section, we propose multi-stage algorithm to extract the type and then the corresponding hyperparameter(s) used in the API. To make all of our experiments and analysis more consistent, we will focus on the task of text completion.

Stage 1: Is it a sampling decoding algorithm or not? In the first step, we aim to determine whether the decoding algorithm is a sampling-based method or not. With sampling methods, the output vary if the same query is sent to the API multiple times. However, if the API employs a non-sampling method and an arbitrary prompt is sent to the API, the output will be the same. In this case, we can conclude that the decoding algorithm is either greedy or beam search. However, it's possible that for very small values of p or k (although this is unlikely to occur in practical applications), the same behavior is observed. In such cases, increasing the number of queries reduces the probability of this outcome to close to zero.

Stage 2: Is it greedy or beam search? In the second step, we aim to determine whether the decoding algorithm is greedy or beam search. In greedy search, at each time step, the most probable token is selected from the probability distribution provided by the model. In other words, when starting with an arbitrary sequence and generating tokens one at a time, at each time step t, the subsequences $s_1 s_2 ... s_i$ for $i < t$ will not change. An example of this can be seen in Table 2.

However, this is not necessarily the case in beam search. Suppose that at time step t, the token s_t is the most probable token from the probability distribution provided by the model. If we query the sequence $s_1 s_2 ... s_t$ to the API to generate the sequence $s_1 s_2 ... s_{t+1}$, in the generated sequence, s_t is not necessarily the same as the s_t in the previous time step. Hence, if we start with an arbitrary sequence and try to complete it token by token and if the subsequences in the output do not change, it means that with a high probability, it is greedy; otherwise, it is beam search.

An example is provided in Table 2. In this example, the API generates "to" at time step $t+1$ and index i. After generating the token at time step $t+4$, we do not see "to" in the subsequence at the index i anymore, and this is due to the nature of beam search. To increase this probability, we can repeat this experiment for more time steps and more arbitrary sequences. We will discuss the required number of queries in Section 5. In Table 2, some examples of generated text using these two decoding strategies for consecutive tokens are presented.

After detecting beam search as the decoding algorithm, we must estimate the beam size as its hyperparameter. Again, suppose that we start with the arbitrary sequence $s_1, s_2, ..., s_{t-1}$, and we plan to generate tokens one by one. In the first step, we generate the token s_t. Then, we use the sequence $s_1, s_2, ..., s_t$ and try to generate the token at time step $t+1$. If we continue with the same process, the token at position t may not be the same as the token we generated in the first step. After repeating this process multiple times, we might have different tokens generated at the position t. Then, we find the rank of these tokens from the sorted tokens based on their probabilities provided by the model. The maximum rank among these tokens is our estimation of the beam size. To make our estimation more reliable, we can repeat this process for multiple arbitrary sequences and pick the maximum value as our final estimation. In Appendix A, we provide more detailed examples to further illustrate this method.

Detecting combined decoding strategies: In the sampling-based decoding algorithms, note that in some cases, the decoding algorithm can be a combination of more than one decoding algorithm. In particular, the following cases are common:

1) only temperature; 2) only top-k sampling; 3) only Nucleus Sampling; 4) only random sampling; 5) both temperature and top-k Sampling; 6) both temperature and Nucleus Sampling; 7) both top-k and Nucleus Sampling; 8) all temperature, top-k, and Nucleus Sampling.

Table 2: This table presents some text examples generated by the API using the greedy or beam search decoding approach for consecutive time steps. The examples are generated using the initial prompt "Students opened their". It showcases the difference in the generated text by these two decoding approach.

Time Step	Generated Text	Approach
t	Students opened their doors	Greedy Search
t+1	Students opened their doors to	Greedy Search
t+2	Students opened their doors to the	Greedy Search
t+3	Students opened their doors to the public	Greedy Search
t+4	Students opened their doors to the public on	Greedy Search
t+5	Students opened their doors to the public on Friday	Greedy Search
t	Students opened their doors	Beam Search
t+1	Students opened their doors to	Beam Search
t+2	Students opened their doors to the	Beam Search
t+3	Students opened their doors to the public	Beam Search
t+4	Students opened their doors for the first time	Beam Search
t+5	Students opened their doors at 6 p.m	Beam Search

In other words, we want to determine which combination of decoding algorithms the API uses, and then determine the corresponding hyperparameter(s). In all cases, we send an arbitrary sequence to the API multiple times. Then, we sort all tokens generated at time step t in descending order based on the number of times they are generated. We use this approach to approximate the final probability distribution of the API. Our results in Section 5 show how many queries might be enough for each stage to have a good approximation of the final probability distribution.

Stage 3: Does the API use temperature to decode? As the first step towards detecting sampling-based decoding methods, we want to see if the API uses a temperature $\tau \neq 1$ to decode or not. In the next theorem, we attempt to find a formula for τ.

Theorem 1. Assume that the victim API uses random sampling with temperature as its decoding algorithm. Suppose that $p_1, p_2, ..., p_{|V|}$ are descending sorted inner probabilities of token among all vocabularies provided by the API's model. Also, assume that $p'_1, p'_2, ..., p'_{|V|}$ are sorted approximated final probabilities generated by the API. Then we have $\tau = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})}$.

Proof. Suppose that $l_1, l_2, ..., l_{|V|}$ are sorted logits generated by the API's model. So, we have $p_i = \frac{e^{l_i}}{\sum_{j=1}^{|V|} e^{l_j}}$ and then $p'_i = \frac{e^{\frac{l_i}{\tau}}}{\sum_{j=1}^{|V|} e^{\frac{l_j}{\tau}}}$. Set $S = \sum_{j=1}^{|V|} e^{l_j}$. So, we have $p_i \times S = e^{\frac{l_i}{\tau}}$ and $p_j \times S = e^{l_j}$. If we divide both side of the recent two equations, we have

$$\frac{p_i}{p_j} = \frac{e^{l_i}}{e^{l_j}} = e^{l_i - l_j} \Rightarrow l_i - l_j = \ln(\frac{p_i}{p_j}) \tag{5}$$

Similarly, we have

$$\frac{p'_i}{p'_j} = \frac{e^{\frac{l_i}{\tau}}}{e^{\frac{l_j}{\tau}}} = e^{\frac{l_i - l_j}{\tau}} \Rightarrow \frac{l_i - l_j}{\tau} = \ln(\frac{p'_i}{p'_j}) \tag{6}$$

If we substitute the equation (5), we have:

$$\tau = \frac{l_i - l_j}{\frac{p'_i}{p'_j}} = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})} \tag{7}$$

□

In the next theorem, we will demonstrate that even if the API employs temperature and Nucleus Sampling simultaneously, the formula $\tau = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})}$ remains valid. Intuitively, this is because the normalization applied after implementing Nucleus Sampling is canceled by the division of probabilities.

Theorem 2. Assume that the victim API uses Nucleus Sampling with the hyperparameter p and temperature τ as its decoding algorithm. Suppose that $p_1, p_2, ..., p_{|V|}$ are descending sorted inner probabilities of token among all vocabularies provided by the API's model. Also, assume that $p'_1, p'_2, ..., p'_{|P|}$ are sorted approximated final probabilities generated by the API. Then we have $\tau = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})}$.

Proof. Suppose that $l_1, l_2, ..., l_{|V|}$ are sorted logits generated by the API's model. In this scenario, in the decoding step, the API first apply the temperature τ, and then pick the minimum number of tokens whose probabilities sum exceeds p. Then it samples from this set of tokens. Now, assume that $p''_1, p''_2, ..., p''_{|V|}$ are sorted probabilities after applying the temperature and before applying Nucleus Sampling. In other words, we have $p_i = \frac{e^{l_i}}{\sum_{j=1}^{|V|} e^{l_j}}$, $p''_i = \frac{e^{\frac{l_i}{\tau}}}{\sum_{j=1}^{|V|} e^{\frac{l_j}{\tau}}}$ and $p'_i = \frac{p''_i}{\sum_{j=1}^{|P|} p''_j}$ where $|P|$ is the number of tokens selected by Nucleus Sampling. Note that after applying Nucleus Sampling, the selected probabilities will get scaled to sum up to 1. So we have:

$$\frac{p'_i}{p'_j} = \frac{p''_i}{p''_j} = \frac{e^{\frac{l_i}{\tau}}}{e^{\frac{l_j}{\tau}}} = e^{\frac{l_i - l_j}{\tau}} \Rightarrow \tau = \frac{l_i - l_j}{\ln(\frac{p'_i}{p'_j})} = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})} \tag{8}$$

The last equation is achieved from the equation (5) in the last theorem. □

Similarly, we can demonstrate that if the API utilizes a combination of temperature and top-k sampling, the parameter τ can still be obtained using the same formula.

Theorem 3. Assume that the victim API uses top-k sampling with the hyperparameter k and temperature τ as its decoding algorithm. Suppose that $p_1, p_2, ..., p_{|V|}$ are descending sorted inner probabilities of token among all vocabularies provided by the API's model. Also, assume that $p'_1, p'_2, ..., p'_k$ are sorted approximated final probabilities generated by the API. Then we have $\tau = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})}$.

Proof. In this scenario, in the decoding step, the API first applies the temperature τ, then selects the top k tokens with the highest probabilities from the resulting distribution. It then rescales the probabilities of these k selected tokens so that they sum to 1 and samples from this set of tokens. To clarify, suppose that

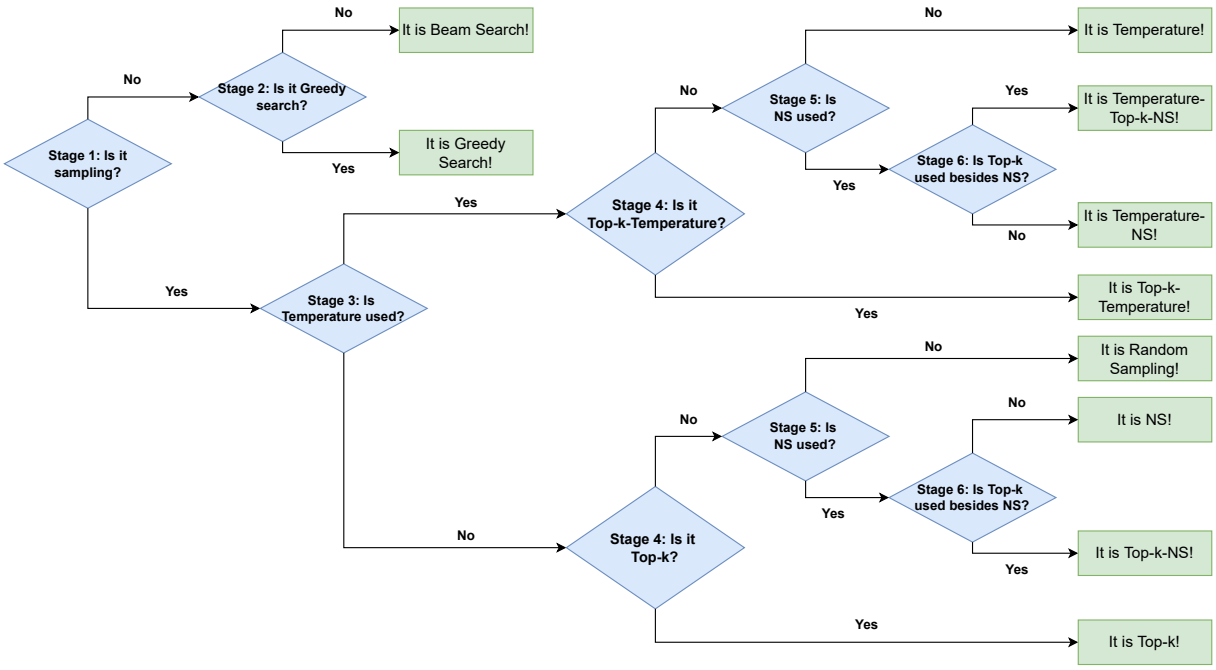

Figure 2: This flowchart presents an overview of all stages of our attack algorithm.

$l_1, l_2, ..., l_{|V|}$ are sorted logits generated by the API's model, and $p''1, p''2, ..., p''|V|$ are the probabilities resulting from applying the temperature to the logits. That is, we have $p_i = \frac{e^{l_i}}{\sum j=1^{|V|} e^{l_j}}$, $p''i = \frac{e^{\frac{l_i}{\tau}}}{\sum j=1^{|V|} e^{\frac{l_j}{\tau}}}$, and p'_i being the probabilities after applying top-k sampling, which is obtained by $\frac{p''_i}{\sum j=1^{k} p''_j}$ where k is the number of tokens selected by top-k sampling. So we have:

$$\frac{p'_i}{p'_j} = \frac{p''_i}{p''_j} = \frac{e^{\frac{l_i}{\tau}}}{e^{\frac{l_j}{\tau}}} = e^{\frac{l_i - l_j}{\tau}} \Rightarrow \tau = \frac{l_i - l_j}{\ln(\frac{p'_i}{p'_j})} = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})} \tag{9}$$

□

Eventually, in the most complicated scenario, when the API uses all three approaches, including temperature, top-k sampling and Nucleus Sampling, at the same time, we can use the same formula to estimate the temperature. In the following theorem, we will show why this is correct.

Theorem 4 Assume that the API uses all three sampling methods, including temperature, top-k and Nucleus Sampling, together as its decoding algorithm. Suppose that $p_1, p_2, ..., p_{|V|}$ are descending sorted inner probabilities of token among all vocabularies provided by the API's model. Also, assume that $p'_1, p'_2, ..., p'_k$ are sorted approximated probabilities generated by the API. Then we have $\tau = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})}$.

Proof. In this scenario, the API employs a decoding algorithm that involves multiple steps. First, it applies a temperature τ to the logits generated by its model. Next, it selects the first k tokens with the highest probabilities, based on the probabilities obtained after applying temperature. Then, it applies Nucleus Sampling, and picks the minimum number of tokens whose probabilities sum exceeds a threshold p. Finally, it samples from this set of tokens. Now, suppose that $l_1, l_2, ..., l_{|V|}$ are sorted logits generated by the API's model, $p''_1, p''_2, ..., p''_k$ are sorted probabilities after applying the temperature and before applying top-k sampling, and $p'''_1, p'''_2, ..., p'''_P$ are sorted probabilities after applying the top-k sampling and before applying Nucleus Sampling. More precisely, we have $p_i = \frac{e^{l_i}}{\sum_{j=1}^{|V|} e^{l_j}}$, $p''_i = \frac{e^{\frac{l_i}{\tau}}}{\sum_{j=1}^{|V|} e^{\frac{l_j}{\tau}}}$, $p'''_i = \frac{p''_i}{\sum_{j=1}^{k} p''_j}$, and $p'_i = \frac{p'''_i}{\sum_{j=1}^{P} p'''_j}$ where k is the hyperparameter of top-k sampling and P is the number of tokens whose probability sum exceeds p. Note that after applying top-k and Nucleus Sampling, the selected probabilities are scaled to sum up to 1. If we set $S' = \sum_{j=1}^{|V|} e^{\frac{l_j}{\tau}}$, $S'' = \sum_{j=1}^{k} p''_j$ and $S''' = \sum_{j=1}^{k} p'''_j$, we have:

$$\frac{p'_i}{p'_j} = \frac{\frac{p'''_i}{S'''}}{\frac{p'''_j}{S'''}} = \frac{p'''_i}{p'''_j} = \frac{\frac{p''_i}{S''}}{\frac{p''_j}{S''}} = \frac{p''_i}{p''_j} = \frac{\frac{e^{\frac{l_i}{\tau}}}{S'}}{\frac{e^{\frac{l_j}{\tau}}}{S'}} = \frac{e^{\frac{l_i}{\tau}}}{e^{\frac{l_j}{\tau}}} = e^{\frac{l_i - l_j}{\tau}} \tag{10}$$

Thus, we have:

$$\tau = \frac{l_i - l_j}{\ln(\frac{p'_i}{p'_j})} = \frac{\ln(\frac{p_i}{p_j})}{\ln(\frac{p'_i}{p'_j})} \tag{11}$$

□

As we showed in theorems, in all cases, we can use the same formula to estimate temperature. So, in the third step, we can see if

the API uses temperature or not, and if so, we can find the hyperparameter too (if $\tau \approx 1$ then we can conclude the temperature is not used in decoding algorrithm).

Stage 4: Is the decoding algorithm one of top-k or temperature and top-k combined? In this step, we propose a simple approach to determine whether the API uses top-k as the last component of its decoding algorithm. This approach does not determine whether top-k is used before Nucleus Sampling, if both are employed by the API. Our approach leverages the property of top-k sampling that the API always selects the first k tokens with the highest probability. To clarify, suppose we regenerate the next token for an arbitrary sequence multiple times. If we repeat this process from scratch, we consistently obtain the same set of unique tokens. The number of unique generated tokens is the hyperparameter k for this decoding approach. Consequently, by utilizing the results obtained from previous and current stages, we can determine whether the decoding algorithm is top-k or temperature and top-k.

To be more specific, we begin with an arbitrary prompt and regenerate the following token N times. If we repeat this process with other arbitrary prompts and obtain the same number of unique tokens, it suggests with a high probability that the API employs top-k sampling as the last step of its decoding algorithm, either top-k sampling or temperature and top-k sampling. The probability that the API does not use top-k sampling and we obtain the same number of unique tokens for some arbitrary sequences is very low. We do a theoretical evaluation in Section 5 to show how probable this happens.

Stage 5: Does the API use Nucleus Sampling as part of its decoding algorithm? In this section, we will show if the API uses Nucleus Sampling as part of its decoding algorithm or not and what its hyperparameter is. In the following theorems, we attempt to find a formula to estimate the Nucleus Sampling hyperparameter p.

Theorem 5. Assuming that the victim API employs Nucleus Sampling with the hyperparameter p as its decoding algorithm, and let $p_1, p_2, ..., p_{|V|}$ be the descending sorted inner probabilities of tokens among all vocabularies provided by the API's model. Also, assume that $p'1, p'2, ..., p'_{|P|}$ are the sorted approximated probabilities generated by the API. Then, the ratio $\frac{p_i}{p'_i}$ serves as an estimation for p.

Proof. Suppose that $p_1, p_2, ..., p_{|V|}$ are sorted inner probabilities generated by the API's model at a specific time step. Also, assume that $p'_1, p'_2, ..., p'_{|P|}$ are sorted final probabilities generated by the API after applying Nucleus Sampling as decoding algorithm. Hence, we have:

$$p'_i = \frac{p_i}{\sum_{j=1}^{|P|} p_j} \Rightarrow \sum_{j=1}^{|P|} p_j = \frac{p_i}{p'_i} \tag{12}$$

We propose that $\sum_{j=1}^{|P|} p_j$ serves as an acceptable approximation for p. As per the definition of Nucleus Sampling, we select the minimum number of tokens such that the sum of their probabilities exceeds p. Therefore, $\sum_{j=1}^{|P|} p_j$ represents the sum of the probabilities of these tokens. While it may be slightly different from the actual value of p for some cases, it is still an acceptable estimation. Even if we utilize Nucleus Sampling with our estimated value, the final probability distribution will remain unchanged. This claim is further confirmed by our results in Section 5.

□

The above formula provides an estimation of p when the probabilities before and after applying Nucleus Sampling are known. Additionally, we can use the ratio $\frac{p_i}{p'_i}$ at stage 5 to determine whether Nucleus Sampling is employed or not. To clarify, if $\frac{p_i}{p'_i} \neq 1$, it indicates that neither top-k nor Nucleus Sampling is utilized in the decoding algorithm. Once the type of sampling is determined, this formula can be used to estimate p.

For instance, after determining that the API applies temperature and Nucleus Sampling in combination, we first use equation (8) to estimate the temperature, then apply this temperature and provide a probability distribution of tokens $p_1, p_2, ..., p_{|V|}$. After that, we can use equation (12) to estimate the hyperparameter p.

Second approach for estimating p in Nucleus Sampling. In this section, we propose an alternative approach for estimating the value of p in Nucleus Sampling that is more straightforward. While the first approach yields accurate estimates, it is highly dependent on the probabilities of the victim model, and small changes in these probabilities can lead to inaccurate estimates. To reduce the reliance on these probabilities, we propose the following approach.

Suppose that $p_1, p_2, ..., p_{|V|}$ are the inner probabilities generated by the API's model, and $p'_1, p'_2, ..., p'_{|P|}$ are the final probabilities generated by the API. In this case, we can estimate p as $\sum_{i=1}^{|P|} p_i$. While this approach may be less accurate than the first method, it may be useful in situations where the exact inner probabilities are not available.

Stage 6: Is top-k used before Nucleus Sampling? In the final cases, we aim to investigate the potential solutions when the API employs top-k sampling before Nucleus Sampling. In the previous stage, we determined whether Nucleus Sampling is employed or not. Suppose that top-k is utilized before Nucleus Sampling in the decoding algorithm. In this scenario, if we apply formula (12) for consecutive time steps, the value obtained from the formula will vary. In other words, top-k truncates the probability distribution prior to applying Nucleus Sampling. As different probability distributions exist across different time steps, the resulting value from the formula will vary. If temperature is also utilized (as determined in stage 3), we can apply it first and then proceed to determine if top-k is employed before Nucleus Sampling. It is important to note that this case emerges as the most complicated one in our analysis and, arguably, not very practical. A summary of our algorithm is presented in Figure 2.

Estimating k and p when they are used together: In stage 6 of our investigation, we determine whether the top-k sampling method is applied prior to Nucleus Sampling. If this is indeed the case, we propose a systematic approach for estimating the values of p and k when top-k sampling and Nucleus Sampling are employed concurrently as components of a decoding algorithm. Suppose that $p_1, p_2, ..., p_{|V|}$ and $q_1, q_2, ..., q_{|V|}$ are descending sorted inner probabilities of a token among all vocabularies provided by the API's model at time steps t_1 and t_2 respectively. Also, assume that

$p'_1, p'_2, ..., p'_{|P|}$ and $q'_1, q'_2, ..., q'_{|P|}$ are sorted approximated probabilities generated by the API at these time steps. Set $S_k^{(1)} = \sum_{j=1}^{k} p_j$, $S_k^{(2)} = \sum_{j=1}^{k} q_i$, $S_p^{(1)} = \sum_{j=1}^{|P|} p''_j$, and $S_p^{(2)} = \sum_{j=1}^{|P|} q''_j$. where $p''_1, p''_2, ..., p''_k$ and $q''_1, q''_2, ..., q''_k$ are sorted probabilities after applying the top-k and before applying Nucleus Sampling. It is not so hard to show that $p'_i = \frac{p_i}{S_k^{(1)} S_p^{(1)}}$ and $q'_i = \frac{q_i}{S_k^{(2)} S_p^{(2)}}$.

Since p is fixed, we can assume that S_p is also fixed. Thus, it can be canceled from the equations. Hence, we have $p'_i = \frac{p_i}{S_k^{(1)}}$ and $q'_i = \frac{q_i}{S_k^{(2)}}$. By dividing the both sides of the recent equations, we have:

$$\frac{S_k^{(1)}}{S_k^{(2)}} = \frac{p'_i}{q'_i} \times \frac{q_i}{p_i} \tag{13}$$

Since there are two unknown variables here ($S_k^{(1)}$ and $S_k^{(2)}$), we set the value for one of them to estimate the other one. To accomplish this, we begin by considering different values of k, and then calculate $S_k^{(1)}$ at time step t_1. Using the formula (13), we can compute $S_k^{(2)}$ for time step t_2. Then, we can examine the corresponding value of k, which leads to the sum $S_k^{(2)}$ in the probability distribution of time step t_2. We repeat this procedure until we find a k such that the corresponding hyperparameter k' resulting from the second time step equals k. Next, we use the estimated k to provide the new probability distribution resulting from top-k, and then apply formula (12) to estimate p. Our experiments will demonstrate the accuracy of the estimation resulting from this approach.

5 EVALUATIONS AND EXPERIMENTS

In this study, we conduct separate experiments on GPT-2, GPT-3, and GPT-Neo models due to the extensive utilization of these models across numerous APIs in various applications. For the experiments on GPT-2 and GPT-Neo, we use the Huggingface[4] library, which allows us to implement various combinations of decoding algorithms. On the other hand, for the experiments on GPT-3, we use the OpenAI API, which only provides access to temperature and Nucleus Sampling. While our proposed attacks are agnostic to the size of the underlying LM, due to the large number of experiments and computational constraints we primarily conduct our experiments on the smaller versions of both models. Nevertheless, in Appendix B, we conduct some representative experiments on larger GPT-2 (medium, large) and GPT-3 models (babbage and curie) to show generalization of our proposed attacks across model sizes. Furthermore, we perform some experiments utilizing the GPT-Neo model. Also, it should be noted that our experiments involve utilizing a custom decoding algorithm in conjunction with GPT-2/3/Neo, rather than directly attacking an actual deployment of these language models.

5.1 Evaluation Metrics

To determine the similarity in functionality between two LMs or text generation-based APIs, we must compare the probability distributions generated by them at each time step. This is because similar decoding algorithms and corresponding hyperparameters will result in similar truncated probability distributions. To evaluate our estimation, we compare the probability distributions of the victim API and the API that uses our estimation as the type and corresponding hyperparameters. In this paper, we use two metrics to achieve this goal: the Kolmogorov-Smirnov test [29] and the Kullback-Leibler divergence (KL divergence) [25].

Kolmogorov-Smirnov Test: One such metric is the Kolmogorov-Smirnov test (K-S test). The K-S test is a non-parametric statistical test that is used to compare two discrete probability distributions. The test evaluates the similarity between the two distributions by comparing the cumulative distribution functions (CDFs) of the observed and hypothesized distributions. The K-S test calculates the maximum distance (referred to as the K-S statistic) between the two CDFs and compares it to a critical value determined by the sample size. If the calculated K-S statistic is larger than the critical value, the null hypothesis that the two distributions are the same is rejected. Results of the K-S test are typically reported as a p-value, which represents the probability that the observed differences between the two distributions are due to chance. A small p-value (typically less than 0.05) indicates that the observed differences are statistically significant and that the two distributions are likely different from each other.

KL Divergence: Another metric for comparing probability distributions is the KL divergence. KL divergence is a measure of the difference between two probability distributions. It is defined as the sum of the product of the probability of each event in one distribution with the natural logarithm of the ratio of the probability of that event in the other distribution. KL divergence is a non-negative value, and it is zero only if the two distributions are identical.

Given that the KL-divergence metric yields a single value, it is necessary to determine what is a good KL-divergence score in our specific context. To address this concern, we compare numerous pairs of probability distributions generated by using the same decoding type and hyperparameters. The average score for these pairs is 0.0017 ± 0.0007. This value can serve as a benchmark for evaluation. However, it should be noted that KL-divergence scores typically exceed 0.1 when different hyperparameters are used.

It is worth noting that MAUVE [36] is a recently proposed automatic metric for evaluating language generators. In other words, it measures the similarity of token distribution between two generated texts. However, like KL-divergence, it also produces a single scalar value. Additionally, evaluating each pair of decoding algorithms using MAUVE requires generating a large number of lengthy sequences using each decoding algorithm, which is not feasible for the number of experiments conducted in this study.

5.2 Evaluation of Each Stage on GPT-2

Since each stage of our attack uses a different algorithm, here we evaluate the performance of each stage of our attack, described in Section 4, one by one.

Stage 1: As theoretical evaluation for this stage, assume the API uses a sampling-based method. The API rarely generates the same sequence if we query the API many times. More precisely, suppose we use the API to generate a sequence of length 50 each time. Also, assume that $p_1, p_2, ..., p_{50}$ are the probabilities that each tokens

[4] https://huggingface.co

Table 3: Results of temperature estimation with different decoding combinations. The accuracy of the estimation is measured using the K-S test p-value and KL divergence score, which indicate the similarity between the provided probability distributions using the estimated hyperparameters and the actual hyperparameters.

Decoding Strategy	Real Temperature	Estimated Temperature	p-value	KL Divergence
Top-k ($k = 30$) & Temperature	0.85	0.8568 ± 0.016	1.0	0.002 ± 0.016
Top-k ($k = 40$) & Temperature	0.75	0.7569 ± 0.016	1.0	0.007 ± 0.011
Top-k ($k = 60$) & Temperature	0.65	0.6586 ± 0.014	1.0	0.002 ± 0.009
NS ($p = 0.9$) & Temperature	0.85	0.8575 ± 0.018	1.0	0.004 ± 0.01
NS ($p = 0.8$) & Temperature	0.8	0.8080 ± 0.014	1.0	0.008 ± 0.01
NS ($p = 0.85$) & Temperature	0.75	0.7579 ± 0.017	1.0	0.007 ± 0.016
Top-k ($k = 40$) & NS ($p = 0.8$) & Temperature	0.9	0.9096 ± 0.017	0.996	0.001 ± 0.0163
Top-k ($k = 50$) & NS ($p = 0.8$) & Temperature	0.85	0.8603 ± 0.017	1.0	0.003 ± 0.012
Top-k ($k = 60$) & NS ($p = 0.8$) & Temperature	0.80	0.8022 ± 0.014	1.0	0.002 ± 0.01

can be generated. Thus, if we send the same prompt as query N times, the probability that API generates the same sequence is $p_1^N \times p_1^N \times ... \times p_1^N$. Even if we set $p_i = 0.99$ and $N = 20$, this probability will be around $4.31e-5$ which is very low. So, it always predicts correctly.

Stage 2: In the second stage of our evaluation, we consider 1000 different APIs. Each API may randomly use either greedy search or beam search as its decoding algorithm. Additionally, if the API uses beam search, the beam size is chosen randomly from a range of common values, specifically between 2 and 10. Our results show that we can detect the type of decoding method used with 100% accuracy. Furthermore, our experiments demonstrate that we can detect the beam size with a high degree of accuracy, using only 40 arbitrary prompts. While it is not straightforward to provide a theoretical justification for the number of queries required to guarantee the correct detection of the beam size, we provide examples in the Appendix A that support our results.

Stage 3: In the third stage, we estimate the temperature for scenarios where it is used as part of the decoding algorithm. We use the inner and final probabilities generated by the API and apply the formula (7) to estimate the temperature. To approximate the final probabilities generated by the API, we send an arbitrary prompt to the API and generate the next token N times. By increasing N, we can achieve a more accurate estimation. In our experiment, we use the arbitrary sequence "Students opened their" as a prompt. Figure 3 illustrates the number of queries required to achieve an accurate estimation of temperature. Additionally, we consider different cases where temperature is used as part of the decoding method and show that the same formula can be used to estimate the temperature in all cases. To demonstrate this, we use various arbitrary values of hyperparameters k and p for top-k and Nucleus Sampling respectively (Table 3).

As shown in Figure 3, with only 1000 queries, we cannot get an acceptable estimation. However, increasing the number of queries makes the estimation more accurate. Even with 10000 queries, we can estimate the temperature.

Stage 4: In the fourth stage, we aim to determine whether the decoding algorithm used by the API employs top-k sampling as its last component, and if so, estimate the value of the hyperparameter k. To do this, we select four arbitrary prompts and repeatedly generate

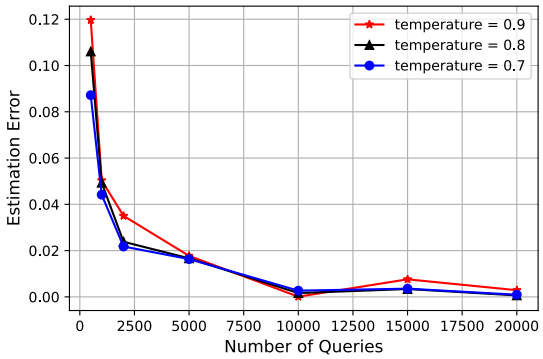

Figure 3: Estimation error of the temperature τ for different numbers of queries. The graph illustrates that using 10000 queries may be sufficient to achieve an accurate estimation.

the next token N times, where N is chosen based on the expected value of k. If the number of unique tokens generated across all sequences is the same, it is likely that the API's decoding algorithm uses top-k sampling, and the number of unique tokens will serve as our estimate for the value of k. We apply this algorithm to eight different decoding strategies and for a range of values for k from 10 to 100. The results in Table 4 indicate the number of queries used in our experiments to estimate k. It should be noted that these values do not necessarily represent the minimum number of queries required to obtain an accurate estimate of k.

Additionally, it is important to consider the number of queries needed to ensure that all top-k tokens have been generated at least once, including the least probable token among them. This question can be viewed as a variant of the well-known problem of determining the expected number of trials until success, where success is defined as the generation of the least probable top-k token. If the probability of success is p, the expected number of trials until success is $\frac{1}{p}$. By using the inner probability distribution provided by the API's model, we can gain insight into the lower bound for the number of queries needed. In practice, the attacker may choose to send more queries than this lower bound to ensure an accurate estimation.

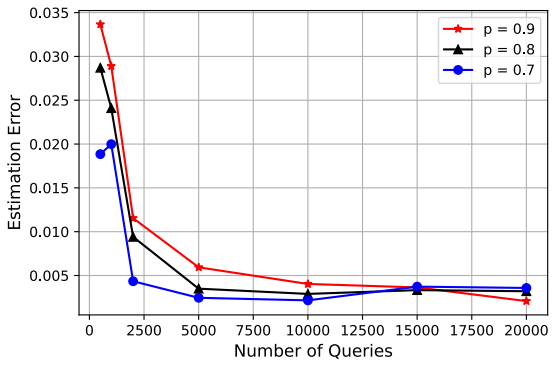

Figure 4: Estimation error of the hyperparameter p for different numbers of queries. The graph shows that even with as few as 5000 queries, an accurate estimation of the hyperparameter p can be achieved.

As previously discussed in Section 3, this approach is ineffective if there is at least one token among the top-k tokens that the API will not generate. The probability of this occurrence is $(1-p)^N$, where p is the probability of that token being generated by the API's model. As an example, if $p = 0.0001$ for a token, and we send queries $N = 50000$ times, we will have $(1-p)^N = 6.7 \times 10^{-3}$. To decrease the likelihood of this approach failing, we must either increase N or p. Increasing N may be cost-prohibitive, so to increase p, we can select prompts that result in a more flattened probability distribution over the next token. We use Kurtosis as a metric to evaluate the level of flattening in the probability distribution.

It is also important to note that when using temperature in addition to top-k sampling, the probability of less likely tokens being generated decreases. This means that more queries are needed to ensure they will be generated in order to make an accurate prediction for the hyperparameter k.

Stage 5: In order to evaluate this stage, we will only consider scenarios that involve Nucleus Sampling and Nucleus Sampling with temperature. As previously discussed in Section 3, it is not possible to use Equation (12) to estimate the hyperparameter p when the API also utilizes top-k sampling. This will be addressed in the next stage. However, we can use Equation (12) to confirm that Nucleus Sampling is being used (if p computed by Equation (12) does not equal 1). We calculate Equation (12) for multiple tokens to obtain a more reliable estimation, and then take the average. Figure 4 illustrates the estimation and the required number of queries, and Table 5 presents the results for various decoding strategies.

Stage 6: In the last stage, we aim to figure out if top-k sampling is used before Nucleus Sampling and then estimate p and k (if top-k is used). To do so, as described in Section 3, we need the final probability distributions in two different time steps. Hence, we use two different prompts: "My school is close to the" and "Students opened their". Then we apply the formula (13) to estimate k. After finding k, then we can apply the formula (12) to estimate p. Results shown in Table 6.

End-to-end Analysis of the Attack: We consider all these steps as a single framework in the second part of our experiments. In other words, we consider 100 different APIs randomly picks one of discussed decoding strategies scenario with random values as their hyperparameters. Then, we apply our algorithm and see which combination of decoding algorithms the API uses. The whole API's decoding algorithms have been predicted correctly by our attack.

Table 4: This table displays the query count required for a precise estimation of the hyperparameter k, which may be slightly higher than the minimal necessary number.

Decoding Strategy	Real k	Number of Queries
Top-k	30	700
Top-k	40	1,000
Top-k	60	1,500
Top-k	80	4,000
Top-k	100	8,000
Top-k & Temperature ($\tau = 0.8$)	30	1,500
Top-k & Temperature ($\tau = 0.8$)	40	2,000
Top-k & Temperature ($\tau = 0.8$)	60	5,000
Top-k & Temperature ($\tau = 0.8$)	80	15,000
Top-k & Temperature ($\tau = 0.8$)	100	20,000

Table 5: Results of hyperparameter p estimation employing strategies of Nucleus Sampling (NS) alone or combined with temperature.

Decoding Strategy	Real p	Estimated p	p-value
NS	0.75	0.75 ± 0.005	1.0
NS	0.90	0.899 ± 0.005	1.0
NS	0.85	0.847 ± 0.005	0.99 ± 0.011
Temperature ($\tau = 0.85$) & NS	0.80	0.801 ± 0.002	1.0
Temperature ($\tau = 0.75$) & NS	0.70	0.699 ± 0.003	0.99 ± 0.009
Temperature ($\tau = 0.65$) & NS	0.70	0.696 ± 0.005	1.0

5.3 Experiments on GPT-3

Unlike HuggingFace, which offers a variety of options for utilizing the GPT-2 model, OpenAI's functionality is more limited. Specifically, the text completion function provided by OpenAI includes options for controlling the diversity of the output, such as temperature and top-p for Nucleus Sampling. However, it does not offer features such as top-k or beam search. Given this constraint, we decided to conduct experiments with GPT-3 in a separate section. We also discovered that in the generation function, employing both temperature and top-p results in only the temperature parameter being applied. Therefore, it is generally recommended to set either top-p or temperature to 1 when using one of these options.

Our experiments involving GPT-3 were executed in a manner analogous to those conducted with GPT-2. OpenAI facilitates access to the GPT-3 API via Python code, employing a public key for querying purposes. This approach allows us to derive the final probability distribution through multiple API queries. The GPT-3 API supplies the inner probability, which we then utilize in conjunction with the formula proposed in our paper to estimate the hyperparameters effectively.

In order to determine which decoding algorithm has been employed by the API, we must first identify whether temperature or Nucleus Sampling is being used. To do this, we evaluate the

Table 6: Hyperparameter estimation results for a combined top-k and Nucleus Sampling (NS) decoding strategy, focusing on the hyperparameters k and p.

Decoding Strategy	Estimated k	Estimated p	p-value
NS ($p = 0.80$) & Top-k ($k = 30$)	28	0.824	0.96 ± 0.043
NS ($p = 0.90$) & Top-k ($k = 30$)	29	0.916	0.99 ± 0.005
NS ($p = 0.80$) & Top-k ($k = 40$)	35	0.833	0.97 ± 0.023
NS ($p = 0.90$) & Top-k ($k = 40$)	38	0.909	0.99 ± 0.004
NS ($p = 0.80$) & Top-k ($k = 50$)	49	0.806	0.99 ± 0.004
NS ($p = 0.90$) & Top-k ($k = 50$)	48	0.909	0.99 ± 0.01

temperature formula and check if it is equal to 1, which indicates that temperature is not being utilized. In this case, either top-p is being used or neither temperature nor top-p are being utilized. To determine whether top-p is being used, we apply the formula for top-p and check if Nucleus Sampling is being applied. Through this process, we are able to detect the type of decoding algorithm as well as its corresponding value.

In these experiments, we set the initial prompt as "My school is close to" and generated the next two tokens 10000 times. We then applied the corresponding formulas to both the original probability distribution provided by the OpenAI API and the resulting distribution obtained by querying the victim API. In order to increase the reliability of our results, we repeated these experiments 8 times and present the results in the Table 7.

5.4 Experiments on GPT-Neo

As previously mentioned, GPT-Neo is a language model similar to GPT-3 that is pre-trained on the Pile dataset. Developed by EleutherAI, the Pile is a large-scale, diverse, and high-quality dataset designed for training language models. In this section, we conduct several experiments using GPT-Neo 1.3B, which has 1.3 billion parameters. The experimental settings remain the same, with 10,000 queries sent each time, and the process is repeated four times to obtain more consistent estimations. The results are presented in Table 8 and 9.

These results effectively illustrate the close alignment of hyperparameter estimations across various scenarios for the GPT-Neo model when employed by the API. By analyzing these results, it becomes apparent that our estimation methods for GPT-Neo yield results that are on par with those of GPT-2 and GPT-3, thus confirming the validity and robustness of our approach in accurately estimating the hyperparameters of the decoding algorithm for GPT-Neo.

5.5 Attack Efficacy in an API with Prompt Engineering

As previously discussed, the only knowledge the attacker requires is the probabilities of the top two tokens generated by the API's LM. There are multiple ways in which an attacker can acquire this information, which are described in detail in Section 3.2. One option is to use a model stealing approach to obtain the internal probability distribution. However, since there have been no successful model-stealing attacks proposed for text generation tasks, the attacker may be motivated to employ our attack without direct access to the probabilities. In this section, we will demonstrate how the attacker can do this when the API relies on prompt engineering rather than fine-tuning.

Table 7: Hyperparameter estimation in a GPT-3-based API utilizing either Nucleus Sampling (NS) or temperature decoding.

Decoding Strategy	Real Value	Estimated Value	p-value
Temperature	0.6	0.596 ± 0.005	1.0
Temperature	0.7	0.701 ± 0.004	1.0
Temperature	0.8	0.795 ± 0.006	1.0
NS	0.6	0.601 ± 0.001	1.0
NS	0.8	0.801 ± 0.007	1.0
NS	0.9	0.903 ± 0.006	1.0

GPT-2/3/Neo models can be applied in various applications through direct usage, prompt engineering, or fine-tuning. Fine-tuning these models for downstream tasks requires computational resources and a private dataset. These limitations have motivated the use of prompt engineering to generate desired text in many text-based tools and applications. There are various platforms that allow users to generate high-quality written content such as blog posts, product descriptions, and other personalized content, such as OpenAI playground, ChatGPT[5], Articoolo[6], and Perplexity AI[7].

In these cases, the attacker can use long prompts to generate probability distributions that are similar to those of the victim model. Specifically, by querying the API with a long prompt, the attacker can use the same prompt and the base model as a reference to approximate the internal probability distribution. Our experiments have shown that adding a prompt to the beginning of a long text does not significantly change the probability distribution over the next token. It is important to note that the long text must be in the same context. For example, if the attacker is attempting to attack a story generation API, using a long drama story and asking the API to complete it as a drama story will result in a probability distribution that is not vastly different. More detailed results are provided in the Appendix C.

6 ANALYSIS OF THE COST OF THE ATTACK

In this section, we aim to provide an estimation of the costs associated with querying the API in order to execute our hyperparameter stealing algorithm. Our primary motivation for this attack is that hiring individuals for human evaluation to determine the best decoding algorithm and corresponding hyperparameters can be costly. Therefore, stealing this information at a lower cost is desirable.

To analyze the cost of our algorithm, we consider the worst-case scenario. In the worst case, if the API utilizes both top-k and Nucleus Sampling as its decoding algorithm, as depicted in Figure 2, all stages of our algorithm must be executed. To do so, we need to send approximately 400, 000 queries to the API. Each query consists of 5 tokens. As a result, we will have 2, 000, 000 tokens in total. Therefore, in the worst-case scenario, the cost of querying the API

[5]https://chat.openai.com
[6]http://articoolo.com
[7]https://www.perplexity.ai

Table 8: Results of hyperparameter estimation for GPT-Neo. This table presents the results of our estimation for the hyperparameters τ and p when the API uses GPT-Neo 1.3B models.

Decoding Strategy	Real Value	Estimated Value	p-value	KL Divergence
Temperature	0.7	0.7132 ± 0.021	1.0	0.003 ± 0.002
	0.8	0.8195 ± 0.035	0.99 ± 0.008	0.002 ± 0.011
	0.9	0.9154 ± 0.066	1.0	0.006 ± 0.003
Nucleus Sampling	0.7	0.706 ± 0.011	1.0	0.006 ± 0.01
	0.8	0.8056 ± 0.016	0.97 ± 0.006	0.006 ± 0.012
	0.9	0.9104 ± 0.017	1.0	0.009 ± 0.002

Table 9: Temperature estimation for the GPT-Neo 1.3B model with various decoding combinations. The K-S test p-value and KL divergence score assess the estimation accuracy by comparing the derived and actual hyperparameter distributions.

Decoding Strategy	Real Temperature	Estimated Temperature	p-value	KL Divergence
Top-k ($k = 30$) & Temperature	0.85	0.8659 ± 0.033	1.0	0.001 ± 0.001
Top-k ($k = 40$) & Temperature	0.75	0.7563 ± 0.021	1.0	0.003 ± 0.001
Top-k ($k = 60$) & Temperature	0.65	0.6561 ± 0.015	1.0	0.001 ± 0.001
NS ($p = 0.9$) & Temperature	0.85	0.8665 ± 0.037	1.0	0.004 ± 0.01
NS ($p = 0.8$) & Temperature	0.8	0.8097 ± 0.025	0.992	0.008 ± 0.013
NS ($p = 0.85$) & Temperature	0.75	0.7567 ± 0.019	1.0	0.007 ± 0.016

with $2,000,000$ tokens would be $(2,000,000/1000 \times x)$, where x is the cost of querying the API with 1000 tokens.

As an example, OpenAI provides pricing information for different versions of the GPT-3 API. GPT-3 has four versions, named Ada, Babbage, Curie, and Davinci. The cost for these models are \$0.0004, \$0.0005, \$0.002, and \$0.02 per 1000 tokens, respectively. Ada is the fastest and cheapest version, while Davinci is the most powerful and expensive one. Therefore, the cost for this API would be \$0.8, \$1, \$4, and \$40 for the four versions respectively.

7 POTENTIAL COUNTERMEASURE

This paper demonstrates that LM-based APIs are vulnerable to hyperparameter stealing attacks, specifically, that the information in the decoding section of LMs in open-ended text generation tasks can be extracted. This highlights the need for APIs to develop defense mechanisms against such attacks. These attacks are dependent on the accuracy of the API's final probability distribution. As a defense, the API can introduce noise into this probability distribution by randomly replacing generated tokens at certain time steps with a probability of 0.1.

A similar approach, known as watermarking, was introduced in [44]. By introducing this noise, the final probability distribution is different from the original one, thus making it more difficult for an attacker to extract the hyperparameters using the formula proposed in the paper. This method is particularly effective for sensitive hyperparameters such as temperature, as small changes in temperature can lead to significant changes in the probability distribution. Table 10 shows that this method is able to effectively defend against the attack.

It should be noted that any defense mechanism may hurt the API's performance. To minimize this impact, the API can replace the generated token with a random one among the most probable tokens. This will disrupt the attacker's ability to extract the hyperparameters while minimizing the effect on the API's performance.

However, to demonstrate that our proposed defense does not negatively impact performance, we select 150 samples from our genre-based story generation dataset and provide 150 corresponding prompts for the text completion task. We then employ both the modified text generation system and the system without our defense to complete these prompts. We utilize perplexity as a metric to illustrate that the performance of our proposed defense is not significantly affected. Table 10 presents the perplexity of the generated text both with and without our proposed defense.

8 ETHICS DISCUSSIONS

Through the course of our experiments, we did not attempt to steal the decoding algorithms of any real-world LM systems. Instead, our experiments were all performed on our own LM systems.

Our work demonstrates the possibility of stealing IP information from real-world LM systems. Note that our demonstrated attacks do not target GPT-2/3/Neo, but instead third parties who use these LMs in building their downstream tasks. Given the abundance of such third parties, it will not be possible to contact these third parties for responsible disclosure of this vulnerability. This paper

Table 10: Results of hyperparameter estimation after applying proposed countermeasure. The table shows the p-values of the new estimations, which confirm that the ruined estimations lead to a significantly different probability distribution. Additionally, the table compares system perplexity before and after deployment, indicating a minor alteration.

Decoding Strategy	New Estimated Hyperparameter	p-value	Perplexity Without Defense	Perplexity With Defense
Temperature ($\tau = 0.9$)	1.0711	1.43e-93	36.659 ± 1.883	36.748 ± 1.827
Temperature ($\tau = 0.8$)	0.9248	1.6312e-37	25.562 ± 1.573	26.403 ± 1.079
Temperature ($\tau = 0.7$)	0.7888	7.135e-08	17.971 ± 1.213	19.664 ± 1.042
NS ($p = 0.9$)	0.9863	1.3e-15	26.418 ± 1.164	28.646 ± 1.204
NS ($p = 0.8$)	0.9040	2.9e-80	19.674 ± 0.812	21.024 ± 0.772
NS ($p = 0.7$)	0.7863	3.2e-60	16.239 ± 0.842	17.1942 ± 0.625

serves to disclose the threat of stealing decoding algorithms of LM-based systems to the whole community. Additionally, we discussed potential countermeasure for the API providers.

9 LIMITATIONS

As previously discussed, certain stages of our attack, such as detecting greedy and beam search or identifying k in top-k sampling, do not necessitate access to inner probabilities, and these stages can be applied to any type of LMs. However, some stages do require such information, resulting in a potential limitation: the reference model used to obtain inner probabilities must be of the same type as the targeted model, as different models exhibit distinct behaviors on sequences. Most text generation APIs rely on a limited set of models, such as GPT-2, GPT-3, or GPT-Neo. Despite their distinct behaviors on identical sequences, it is possible to compare the outputs with texts generated by these models. It is important to note that decoding algorithms do not alter the order of tokens based on their probabilities; they only adjust the probabilities to assign different weights to various tokens. Consequently, detecting the base model employed by the API is not an overly challenging task.

10 RELATED WORKS: ATTACKS ON NLP

Like image applications, NLP models are vulnerable to any types of attacks. Membership Inference Attacks (MIA) disclose if a datapoint was used to train the victim model [43]. Recent works have shown how powerful these attacks are on NLP classification tasks [41]. Mahloujifar et al. [28] show that word embedding is also vulnerable to MIA. Carlini et al. [6] investigate memorization in large LMs that leads to data extraction attacks. Model inversion attacks [14] reconstruct representative views of subset of examples.

The backdoor attack [18] is another emergent issue that threats language models. In backdoor attacks, the adversary injects backdoor into language models during the training. Hence, the adversary uses a trigger in a poisoned example to activate the backdoor to make the model produces the output it wants, while the model has a normal behavior for other benign examples. Recently, many works have been done to investigate backdoor attack in language models [8, 52, 26, 42]. Besides these, two more relevant attacks to our work are model stealing/imitation and hyperparameter stealing.

Model Stealing/Imitation/Extraction Attack. Model stealing attacks, also called model extraction attacks or model imitation attacks, have been widely explored in simple classification [45] computer vision tasks [35], both theoretically [30] and empirically. Model extraction attacks seek to replicate the functionality of the victim model, thereby facilitating further attacks against it. For instance, the adversary can use the extracted model to construct adversarial examples that make the model to have incorrect predictions [20].

Many works [24, 20, 27] study model extraction attacks against BERT-based APIs. Wallace et al. [48] investigate model stealing attacks on machine translation by querying them in black-box setting. Most of these works solve the problem for the tasks where the output or response is more restricted or predictable, including sentiment classification, machine translation, or extractive QA tasks. Model extraction attacks has not been explored for open-ended text generation that requires the output to be more diverse. This problem is a potential research path as future works.

Hyperparameter Stealing Attack. Wang et al. [49] demonstrate that various machine learning algorithms are vulnerable to hyperparameter stealing attacks. Oh et al. [34] propose a meta-model to infer some attributes of the Neural Network related to the architecture and training process. In LMs, hyperparameters can be classified into two groups: one comprising parameters universal to many machine learning algorithms (e.g., batch size, regularization term, k in KNN), and the other exclusive to NLP models. This paper concentrates on decoding algorithms and their corresponding hyperparameters. Stealing hyperparameters has not been studied as much as other attacks.

11 CONCLUSION

In this paper, we presented the first decoding algorithm stealing attack on LMs. The decoding algorithms used in open-ended text generation are critical, and organizations are willing to invest significant resources to find the best decoding strategies and corresponding hyperparameters. This motivates adversaries to attempt to steal this information. Our results showed that it is possible to do so at a relatively low cost. We also proposed potential defense against this mathematical attack. Additionally, we highlighted that inferring certain information is possible even without direct access to the probabilities provided by the API's LM. We hope that this work brings attention to the vulnerabilities of decoding algorithms in LM-based systems and encourages further research in this area.

ACKNOWLEDGMENTS

This work was supported by the NSF grant 2131910.

REFERENCES

[1] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. 1985. A learning algorithm for boltzmann machines. *Cognitive science*, 9, 1, 147–169.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[3] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2022. Gpt-neo: large scale autoregressive language modeling with mesh-tensorflow, 2021. *URL: https://doi. org/10.5281/zenodo*, 5297715.

[4] Tom Brown et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877–1901.

[5] Massimo Caccia, Lucas Caccia, William Fedus, Hugo Larochelle, Joelle Pineau, and Laurent Charlin. 2020. Language gans falling short. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=BJgza6VtPB.

[6] Nicholas Carlini et al. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, 2633–2650.

[7] Asli Celikyilmaz, Elizabeth Clark, and Jianfeng Gao. 2020. Evaluation of text generation: a survey. *arXiv preprint arXiv:2006.14799*.

[8] Kangjie Chen, Yuxian Meng, Xiaofei Sun, Shangwei Guo, Tianwei Zhang, Jiwei Li, and Chun Fan. 2021. Badpre: task-agnostic backdoor attacks to pre-trained nlp foundation models. *arXiv preprint arXiv:2110.02467*.

[9] Elizabeth Clark, Yangfeng Ji, and Noah A Smith. 2018. Neural text generation in stories using entity representations as context. In *Proceedings of the 2018 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long papers)*, 2250–2260.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[11] Yao Dou, Maxwell Forbes, Rik Koncel-Kedziorski, Noah A Smith, and Yejin Choi. 2021. Scarecrow: a framework for scrutinizing machine text. *arXiv preprint arXiv:2107.01294*.

[12] Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833*.

[13] Jessica Ficler and Yoav Goldberg. 2017. Controlling linguistic style aspects in neural language generation. *arXiv preprint arXiv:1707.02633*.

[14] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 1322–1333.

[15] Leo Gao and Stella Biderman. 2020. Sid black, laurence golding, travis hoppe, charles foster, jason phang, horace he, anish thite, noa nabeshima, shawn presser, and connor leahy. 2020. the pile: an 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

[16] Sebastian Gehrmann, Elizabeth Clark, and Thibault Sellam. 2022. Repairing the cracked foundation: a survey of obstacles in evaluation practices for generated text. *arXiv preprint arXiv:2202.06935*.

[17] Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

[18] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. Badnets: identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*.

[19] Tatsunori B Hashimoto, Hugh Zhang, and Percy Liang. 2019. Unifying human and statistical evaluation for natural language generation. *arXiv preprint arXiv:1904.02792*.

[20] Xuanli He, Lingjuan Lyu, Qiongkai Xu, and Lichao Sun. 2021. Model extraction and adversarial transferability, your bert is vulnerable! *arXiv preprint arXiv:2103.10013*.

[21] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*.

[22] Ari Holtzman, Jan Buys, Maxwell Forbes, Antoine Bosselut, David Golub, and Yejin Choi. 2018. Learning to write with cooperative discriminators. *arXiv preprint arXiv:1805.06087*.

[23] Kalpesh Krishna, Yapei Chang, John Wieting, and Mohit Iyyer. 2022. Rankgen: improving text generation with large ranking models. *arXiv preprint arXiv:2205.09726*.

[24] Kalpesh Krishna, Gaurav Singh Tomar, Ankur P Parikh, Nicolas Papernot, and Mohit Iyyer. 2020. Thieves on sesame street! model extraction of bert-based apis. In *International Conference on Learning Representations*.

[25] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics*, 22, 1, 79–86.

[26] Linyang Li, Demin Song, Xiaonan Li, Jiehang Zeng, Ruotian Ma, and Xipeng Qiu. 2021. Backdoor attacks on pre-trained models by layerwise weight poisoning. *arXiv preprint arXiv:2108.13888*.

[27] Lingjuan Lyu, Xuanli He, Fangzhao Wu, and Lichao Sun. 2021. Killing two birds with one stone: stealing model and inferring attribute from bert-based apis. *arXiv preprint arXiv:2105.10909*.

[28] Saeed Mahloujifar, Huseyin A Inan, Melissa Chase, Esha Ghosh, and Marcello Hasegawa. 2021. Membership inference on word embedding and beyond. *arXiv preprint arXiv:2106.11384*.

[29] Frank J Massey Jr. 1951. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46, 253, 68–78.

[30] Smitha Milli, Ludwig Schmidt, Anca D Dragan, and Moritz Hardt. 2019. Model reconstruction from model explanations. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, 1–9.

[31] Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. 2016. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*.

[32] Ali Naseh, Kalpesh Krishna, Mohit Iyyer, and Amir Houmansadr. 2023. Stealing the decoding algorithms of language models. (2023). arXiv: 2303.04729 [cs.LG].

[33] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*.

[34] Seong Joon Oh, Max Augustin, Mario Fritz, and Bernt Schiele. 2018. Towards reverse-engineering black-box neural networks. In *International Conference on Learning Representations*.

[35] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Knockoff nets: stealing functionality of black-box models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 4954–4963.

[36] Krishna Pillutla, Swabha Swayamdipta, Rowan Zellers, John Thickstun, Sean Welleck, Yejin Choi, and Zaid Harchaoui. 2021. Mauve: measuring the gap between neural text and human text using divergence frontiers. *Advances in Neural Information Processing Systems*, 34, 4816–4828.

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1, 8, 9.

[38] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21, 140, 1–67.

[39] Abigail See, Aneesh Pappu, Rohun Saxena, Akhila Yerukola, and Christopher D. Manning. 2019. Do massively pretrained language models make better storytellers? In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*. Association for Computational Linguistics, Hong Kong, China, (Nov. 2019), 843–861. DOI: 10.18653/v1/K19-1079.

[40] Louis Shao, Stephan Gouws, Denny Britz, Anna Goldie, Brian Strope, and Ray Kurzweil. 2017. Generating high-quality and informative conversation responses with sequence-to-sequence models. In https://arxiv.org/abs/1701.03185.

[41] Virat Shejwalkar, Huseyin A Inan, Amir Houmansadr, and Robert Sim. 2021. Membership inference attacks against nlp classification models. In *NeurIPS 2021 Workshop Privacy in Machine Learning*.

[42] Lujia Shen, Shouling Ji, Xuhong Zhang, Jinfeng Li, Jing Chen, Jie Shi, Chengfang Fang, Jianwei Yin, and Ting Wang. 2021. Backdoor pre-trained models can transfer to all. *arXiv preprint arXiv:2111.00197*.

[43] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 3–18.

[44] Sebastian Szyller, Buse Gul Atli, Samuel Marchal, and N Asokan. 2021. Dawn: dynamic adversarial watermarking of neural networks. In *Proceedings of the 29th ACM International Conference on Multimedia*, 4417–4425.

[45] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction {apis}. In *25th USENIX security symposium (USENIX Security 16)*, 601–618.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In https://arxiv.org/pdf/1706.03762.pdf.

[47] Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. 2016. Diverse beam search: decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*.

[48] Eric Wallace, Mitchell Stern, and Dawn Song. 2020. Imitation attacks and defenses for black-box machine translation systems. *arXiv preprint arXiv:2004.15015*.

[49] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing hyperparameters in machine learning. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 36–52.

[50] Ziqi Yang, Ee-Chien Chang, and Zhenkai Liang. 2019. Adversarial neural network inversion via auxiliary knowledge alignment. *arXiv preprint arXiv:1902.08552*.

[51] Susan Zhang et al. 2022. Opt: open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

[52] Xinyang Zhang, Zheng Zhang, Shouling Ji, and Ting Wang. 2021. Trojaning language models for fun and profit. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 179–197.

[53] Ming Zhong, Pengfei Liu, Yiran Chen, Danqing Wang, Xipeng Qiu, and Xuanjing Huang. 2020. Extractive summarization as text matching. *arXiv preprint arXiv:2004.08795*.