



UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Secure Steganography on ML-Based Channels

Sichere Steganographie auf ML-basierten Kanälen

Bachelorarbeit

verfasst am

Institut für Theoretische Informatik

im Rahmen des Studiengangs

IT-Sicherheit

der Universität zu Lübeck

vorgelegt von

Jeremy Boy

ausgegeben und betreut von

Prof. Dr. Maciej Liśkiewicz

Lübeck, den 5. Oktober 2022

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Jeremy Boy

Zusammenfassung

Steganographie ermöglicht vertrauliche Kommunikation auch dann, wenn die Nutzung von Kryptographie verhindert oder erschwert wird. Das Meteor-Stegosystem erweitert klassische steganographische Primitive für die Verwendung auf ML-basierten Kanälen wie etwa dem generativen Modell GPT-2. Im Meteor-Stegosystem wird der Geheimtext in den verwendeten Zufall im Sampling des generativen Modells eingebettet.

In dieser Arbeit werden Zuverlässigkeit sowie Sicherheit gegen Chosen-Hiddentext-Angriffe des Meteor-Stegosystems untersucht. Wir werden feststellen, dass die Zuverlässigkeit insbesondere bei längeren Geheimtexten verletzt ist. Dies geschieht aufgrund uneindeutiger Tokenisierung von Text, welche bei Subword-Tokenisierung, einem Feature moderner generativer Modelle, auftreten kann. Zudem ist das Meteor-Stegosystem deterministisch und dadurch verwundbar gegen Chosen-Hiddentext-Angriffe mit Querykomplexität größer als eins. Wir präsentieren Modifikationen, die die Zuverlässigkeit und Sicherheit gegen Chosen-Hiddentext-Angriffe des Meteor-Stegosystems verbessern. Außerdem wird eine Modifikation diskutiert, mit der das Meteor-Stegosystem zur überzeugenderen Verwendung im Kontext von Instant Messaging verwendet werden kann.

Abstract

Steganography enables confidential communication even if the use of cryptography is prevented or impeded. The Meteor stegosystem extends classical steganographic primitives for use on ML-based channels such as the generative model GPT-2. In the Meteor stegosystem, the hiddentext is embedded in the randomness used in sampling from the generative model.

In this thesis, we analyze reliability and security against chosen hiddentext attackers of the Meteor stegosystem. We will find that Meteor is not reliable, especially for longer hiddentexts. This is due to ambiguous tokenization of text, which can occur with subword tokenization, a feature of modern generative models. Additionally, the Meteor stegosystem is deterministic and therefore vulnerable to chosen hiddentext attackers with query complexity greater than one. We present modifications that improve the reliability and security against chosen hiddentext attackers of the Meteor stegosystem. We also discuss a modification that enables us to use the Meteor stegosystem for more compelling use in the context of instant messaging.

Acknowledgements

I would like to express my sincere thanks to Prof. Dr. Maciej Liśkiewicz for sparking my interest in the topic at hand and enabling me to write this thesis. Our meetings and conversations have helped me tremendously to approach the highly interesting topic of steganography and to formalize my sometimes vague thoughts.

I am also grateful for my fellow students, for their editing help and late-night feedback sessions.

Lastly, I would like to thank my parents Viola and Torsten for their continued love and support. Without you, my studies would not have been possible.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Structure of this Thesis	1
2	Preliminaries	3
3	Backgrounds of the Meteor Stegosystem	4
3.1	Provably Secure Steganography	4
3.2	Generative Neural Networks	8
3.3	The Meteor Stegosystem	13
4	Improving the Reliability of the Meteor Stegosystem	17
4.1	Ambiguous Tokenization	17
4.2	Reliability of the Meteor Stegosystem	18
4.3	Algorithmic Reconstruction of Tokenization Candidates	21
5	Adapting the Meteor Stegosystem for Two-Way Communication	26
5.1	Protocol for Steganographic Two-Way Communication	26
5.2	Implementing Steganographic Two-Way Communication	28
6	Improving the Security of the Meteor Stegosystem	30
6.1	Security Against Chosen Hiddentext Attackers	30
6.2	Improving Meteor's Security Against Chosen Hiddentext Attackers	32
7	Conclusion	34
	Bibliography	36

1

Introduction

1.1 Motivation

Steganography extends the concepts of cryptography by not only hiding the contents of a message, but also hiding the mere existence of a hidden message. If we can establish provably secure steganographic protocols – or stegosystems – people can use them to circumvent censorship and avoid persecution in hostile environments. Steganographic techniques such as protocol obfuscation are already in use today, for example in the Tor anonymity service to circumvent the so-called “Great Firewall” of China (Winter and Lindskog, 2012). Those techniques are not necessarily cryptographically secure but might rely on the adversary to have a default open policy, i.e., allow communication if the protocol in use cannot be identified.

This thesis discusses Meteor, an innovative cryptographically secure steganographic protocol that exploits the randomness used in sampling from distributions of machine learning based text generation models, such as GPT-2, to embed a hidden message into the generated text. We will show that the Meteor stegosystem is not reliable concerning the formal definition of steganographic reliability. Afterwards, we present a modification to the Meteor stegosystem to improve its use in instant messaging applications by replacing the generative model with one that is trained to generate chat-like texts. Subsequently, Meteor’s security against chosen hiddentext attackers is discussed. We will show that Meteor is secure against chosen hiddentext attackers with a query complexity of one. Afterwards, we will argue that an attacker can distinguish Meteor’s output from a random oracle using two queries. For both reliability and security, we propose algorithms that improve the Meteor stegosystem.

1.2 Structure of this Thesis

Chapter 2 introduces notation that will be used throughout this thesis.

Chapter 3 discusses works we will need to understand the architecture of the Meteor stegosystem. First, we will discuss provably secure steganography. Afterwards, we will examine how generative neural networks (GNNs) such as GPT-2 iteratively produce

high-quality text. Finally, we present the Meteor stegosystem, a provably secure steganographic protocol on ML-based channels.

Chapter 4 analyzes the reliability of the Meteor stegosystem. This chapter relies on the material presented in Chapter 3, especially the definitions of steganographic (un-)reliability established in Definition 3.5 and Definition 3.6 as well as the discussion of GNNs in Section 3.2. We will see that the Meteor stegosystem is not reliable because of subword tokenization used in most modern GNNs. Afterwards, algorithms are presented that improve the reliability of Meteor on the receiver side. Unfortunately, these algorithms – in the worst case and for some models – introduce computational overhead exponential in the length of the generated stegotext.

In Chapter 5 we present a small modification to the Meteor stegosystem that allows two-way communication. This chapter relies on the material presented in Section 3.2 and Section 3.3. To generate dialog-like messages, we will embed a fixed number of bits in messages generated by the DialoGPT model. This modification enables users of the Meteor stegosystem to communicate more naturally in the context of instant messaging. It also makes sure that the algorithms from Chapter 4 can be applied more conveniently with smaller overhead.

In Chapter 6, we will perform a security analysis of the Meteor stegosystem using Hopper’s definition of a secure stegosystem. This chapter relies heavily on the material presented in Chapter 3, especially the definition of steganographic security against chosen hiddentext attackers in Definition 3.8. We will argue that the Meteor stegosystem is insecure against SS-CHA adversaries with query complexity greater than one. Afterwards, a modification to the Meteor stegosystem is presented that increases its security against chosen hiddentext attackers.

Finally, in Chapter 7, we will discuss the results of this thesis and propose topics for further research in a conclusion.

2

Preliminaries

In this chapter, we will introduce the notation that will be used in this thesis.

We denote by $x \leftarrow \$ S$ a value x sampled uniformly at random from the set S . We denote the power set of S by 2^S .

When working with a probability distribution D , $x \leftarrow \$ D$ denotes the random sampling of a value according to D . We also denote the probability of some value $x \in D$ as $Pr_D[x]$ or $Pr[x]$ if the distribution is clear from context. We define U_n as the uniform distribution over bit strings of length n .

In code samples, we denote $x \leftarrow a$ as assignment of value a to variable x . When working with sorted sequences, such as arrays or strings, we use pythonic slicing notation. Given a sequence $x = (x_0, x_1, \dots, x_{n-1})$ of length n , we use the following slicing operations:

- $x[i] = x_i$,
- $x[i : j] = (x_i, x_{i+1}, \dots, x_{j-1})$,
- $x[i :] = x[i : n] = (x_i, x_{i+1}, \dots, x_{n-1})$.

We use $x \leftarrow \epsilon$ as the empty value assignment. We also use the infix operator $\oplus : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$, $a \oplus b$ for the exclusive-or of values a and b . The infix concatenation operator $\| : X^n \times X^m \rightarrow X^{n+m}$, $a \| b$ appends the value or sequence b to a .

The variable λ denotes the security parameter used in cryptographic primitives or steganographic protocols. We denote by $\mu(\lambda)$ a function negligible in λ , i.e., for every integer c there exists an integer c_0 such that for all $\lambda > c_0$,

$$|\mu(\lambda)| < \frac{1}{\lambda^c}.$$

In general, the hiddentext in a steganographic protocol is a bit string $m \in \{0, 1\}^*$. In this thesis, however, we use sequences of Unicode characters as hiddentexts. We denote the set of Unicode characters as \mathcal{U} and hiddentext messages $m \in \mathcal{U}^*$ as strings of Unicode characters.

3

Backgrounds of the Meteor Stegosystem

3.1 Provably Secure Steganography

Steganography extends the concepts of cryptography to hide the mere existence of a message. This makes it robust against censorship. While cryptography can be used to hide the contents of a confidential message, its use is easily discoverable by adversaries.

Steganography is usually modeled using the Prisoners' Problem (Simmons, 1984). The two parties Alice and Bob are imprisoned. Although they can exchange messages, the messages' contents are surveilled by a warden we will call Eve. Alice and Bob now want to craft and exchange escape plans while Eve observes the messages exchanged between Alice and Bob. Since Eve does not want Alice and Bob to leave the prison prematurely, she tries to distinguish the exchange of the escape plan from innocuous messages. If Eve succeeds, she suppresses the message. For an illustration of the Prisoners' Problem, see Figure 3.1.

We suppose that Alice and Bob communicate using a sequence of *documents*, called the *stegotext*. In general, the documents Δ are defined as a set of efficiently recognizable prefix-free bit strings of finite length (Hopper, Ahn, and Langford, 2008). However, in this thesis and the Meteor paper, the set of documents is denoted by a set \mathcal{T} (for *tokens*), which are sequences of Unicode characters, i.e., $\Delta = \mathcal{T} \subseteq \mathcal{U}^*$. For example, given

$$\mathcal{T} = \{\text{"hello"}, \text{"_world"}, \text{"_stega"}, \text{"nography"}\},$$

the documents (or tokens) are “hello”, “_world”, “_stega” and “nography”. The stegotext “hello steganography” consists of three tokens. In Chapter 4, we will see that tokens in generative models used with Meteor are not prefix-free, i.e., there exist tokens $t \in \mathcal{T}$ such that there exists a prefix of t which is a token as well. This causes reliability issues due to *ambiguous tokenization*, which we will introduce in Section 4.1.

While the Prisoners' Problem has been established in the 1980s, the cryptographic community lacked a strict definition of steganographic protocols and security for the next two decades. In 2002, a cryptographic point of view on steganographic protocols as well as a notion of steganographic security against chosen hiddentext attackers alongside a provably secure steganographic protocol were introduced (Hopper, Langford, and Ahn, 2002). The definitions and theorems established by Hopper, Langford and von Ahn have

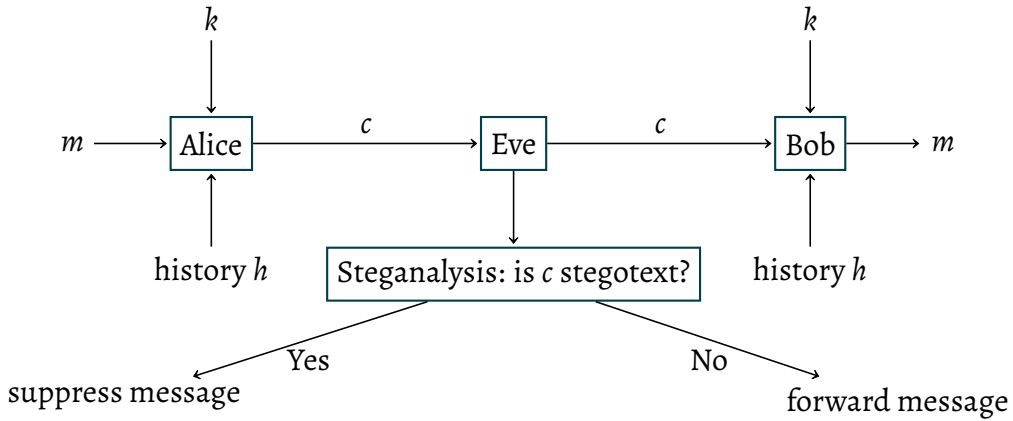


Figure 3.1: An illustration of the Prisoners' Problem (Simmons, 1984). Alice and Bob are imprisoned in separated cells. They want to escape and try to coordinate their escape plans. They are surveilled by a warden Eve who monitors their communication. If Eve detects a hidden message by successfully distinguishing it from an innocuous message, she suppresses the message. Otherwise, the message is forwarded. It is expected that Alice and Bob have been able to exchange a secret key k beforehand. They also have access to the same initial *history* h , which can be public.

successfully been advanced in Hopper's dissertation (Hopper, 2004) and further works. We will now introduce some formal definitions for steganographic protocols:

Definition 3.1 (Channel). A *channel* C is a function that takes a history $h \in \Delta^*$ as input and produces a probability distribution $\mathcal{D}_{C,h}$ on Δ . We denote the set of histories for C as \mathcal{H} .

A sequence $c = c_1 \| c_2 \| \dots \| c_\ell \in \Delta^\ell$ of documents can be generated by generating the distributions $\mathcal{D}_{C,h \| c_1 \| c_2 \| \dots \| c_i}$ by querying C with some initial history h and appending the sampled value c_i to h for the next iteration. We will now formally introduce the notion of *marginal channel distribution*, which we will later need for the definition of steganographic security.

Definition 3.2 (Marginal Channel Distribution, (Hopper, 2004)). Let C be a channel over Δ and $h \in \mathcal{H}$ a history. We let $C_h = \mathcal{D}_{C,h}$ denote the *marginal channel distribution on a single document from Δ* conditioned on the history h of already drawn documents. We let C_h^ℓ denote the *marginal channel distribution on a sequence of ℓ documents from Δ* conditioned on the history h of already drawn documents.

We will also need the notion of the *support of a distribution*. The support of a distribution consists of all elements with non-zero probability.

Definition 3.3 (Support). Let D be a distribution. We denote the *support of D* as

$$\text{supp}(D) = \{x \in D \mid \Pr_D[x] \neq 0\}.$$

Now that we have established the notion of channels, we can commence with the definition of a stegosystem.

Definition 3.4 (Stegosystem, (Hopper, 2004)). A *steganographic protocol* S , or *stegosystem*, with security parameter λ is a pair of probabilistic algorithms:

- $S.KeyGen$ takes as input a security parameter λ and returns a key $k \in \{0, 1\}^\lambda$.
- $S.Encode$ (abbreviated SE) takes as input a key $k \in \{0, 1\}^\lambda$, a string $m \in \mathcal{U}^*$ (the hiddentext), and a message history $h \in \mathcal{H}$. $SE(k, m, h)$ returns a sequence of documents $c_1 || c_2 || \dots || c_\ell$ (the stegotext) from the support of C_h^ℓ .
- $S.Decode$ (abbreviated SD) takes as input a key $k \in \{0, 1\}^\lambda$, a sequence of documents $c = c_1 || c_2 || \dots || c_\ell$, and a message history $h \in \mathcal{H}$. $SD(k, c, h)$ returns a hiddentext $m \in \mathcal{U}^*$.

This definition resembles that of a symmetric encryption scheme, except that both SE and SD take a message history h as an additional argument.

Now that we have defined what a steganographic protocol is, we will introduce some properties we would like a stegosystem to satisfy. First of all, we want a stegosystem to be reliable. The decoding of a given stegotext c , given the correct key k and history h , should yield the original hiddentext m . To measure this, we define the unreliability of a stegosystem S .

Definition 3.5 (Unreliability, (Berndt, 2017)). The *unreliability* $UnRel_S(\lambda)$ is the maximum probability that the decoder SD fails, i.e.,

$$UnRel_S(\lambda) = \max_{\substack{k \in \{0,1\}^\lambda \\ m \in \mathcal{U}^* \\ h \in \mathcal{H}}} \{Pr[SD(k, SE(k, m, h), h) \neq m]\}.$$

In some works, the term *correctness* of a stegosystem is used (Hopper, 2004; Kaptchuk et al., 2021b). In this thesis, however, we will call a stegosystem *reliable* instead.

Definition 3.6 (Reliability). A stegosystem S is *reliable* if its unreliability is negligible in λ .

The introduction of *negligible unreliability* allows the stegosystem to sometimes yield wrong results, but only with probability negligible in the security parameter λ . This proves helpful for reductions on established cryptographic primitives.

Since we naturally want our stegosystem to be prone to analysis by Eve, we also need a definition of security. Intuitively, we want Eve to be unable to efficiently detect the use of a stegosystem S without knowledge of the secret key k . We can model this as a distinguishing game between a challenger and Eve. First, the challenger samples a key $k \leftarrow U_\lambda$. Next, Eve sends a tuple (m, h) to the challenger. The challenger now secretly flips a coin. If the coin shows heads, the challenger uses an oracle ST (for *StegoText*) to generate a stegotext $c = SE(k, m, h)$. Otherwise, the challenger uses an oracle CT (for *CoverText*) to sample a coverttext c of length $\ell = |SE(k, m, h)|$ from C_h^ℓ . Either way, the challenger sends c back to Eve. If Eve can guess the result of the coin flip using a distinguisher W , she must have been able to distinguish a stegotext from a coverttext. Since Eve chooses the queries (m, h) to send to the challenger, she is called a *chosen hiddentext attacker*.

If Eve can distinguish a stegotext from a coverttext, we say that she has an *advantage* against the stegosystem in the distinguishing game. The naïve approach to win this game would be to simply guess the coin flip with a success rate of $\frac{1}{2}$ if they play this game just

once. Therefore, the challenger and Eve repeat this game $q(\lambda)$ times with up to $l(\lambda)$ bits of hiddentext chosen by Eve. Additionally, Eve is computationally limited by time $t(\lambda)$. The insecurity of a stegosystem is the maximum possible advantage any attacker can have. More formally, we define a chosen hiddentext attack as follows:

Definition 3.7 (Chosen Hiddentext Attack, (Hopper, 2004)). In a chosen hiddentext attack, Eve is given access to a “mystery oracle” M which is chosen from one of the following distributions:

1. ST : The oracle ST has a uniformly chosen key $k \leftarrow U_\lambda$ and responds to queries (m, h) with a stegotext drawn from $SE(k, m, h)$.
2. CT : The oracle CT has a uniformly chosen k as well, and responds to queries (m, h) with a coverttext of length $\ell = |SE(k, m, h)|$ drawn from C_h^ℓ .

Eve now has to determine which oracle M she has been given access to. $W^M(1^\lambda)$ outputs a bit which represents her guess about the type of M . We define Eve’s *advantage against a stegosystem S for channel C* by

$$Adv_{S,C,W}^{ss}(\lambda) = |Pr[W^{ST}(1^\lambda) = 1] - Pr[W^{CT}(1^\lambda) = 1]|,$$

where the probability is taken over the randomness of ST , CT , and W . Define the *insecurity of S with respect to channel C* by

$$InSec_{S,C}^{ss}(t, q, l, \lambda) = \max_{W \in \mathcal{W}(t, q, l)} \{Adv_{S,C,W}^{ss}(\lambda)\},$$

where $\mathcal{W}(t, q, l)$ denotes the set of all adversaries who make at most $q(\lambda)$ queries totaling at most $l(\lambda)$ bits (of hiddentext) and running in time at most $t(\lambda)$.

Definition 3.8 (Steganographic Security, (Hopper, 2004)). A stegosystem S with security parameter λ is called (t, q, l, ϵ) *steganographically secret against chosen hiddentext attack for the channel C* ((t, q, l, ϵ) -SS-CHA- C) if $InSec_{S,C}^{ss}(t, q, l, \lambda) \leq \epsilon$.

In the past two decades, many researchers built upon those cryptographic definitions of steganographic protocols. In (Dedić et al., 2008), complexity limitations of black-box steganography were investigated. In black-box steganography, the sending party Alice knows nothing about the underlying distribution except a bound to its min-entropy, which is the negative logarithm of the distribution’s element with the highest probability, and the ability to sample from it but does not have to know the generated distribution. While a black-box construction provides a definition with the least required knowledge of the channel, exponential sampling complexity is required for the encoding party with respect to the number of message bits.

In contrast, white-box steganography describes the other extreme, where Alice either knows the channel’s distributions beforehand or can efficiently compute them. Both black-box and white-box steganography are unrealistic in practice because Alice is possibly able to gain some knowledge or approximation about the channel but cannot perfectly determine it. Therefore, grey-box steganography has been introduced (Liśkiewicz, Reischuk, and Wölfel, 2013). Grey-box steganography tries to resolve these deficiencies by expecting that Alice has some initial knowledge of the coverttext channel. Alice then

uses machine learning techniques to learn about the coverttext distribution and generates suitable stegotexts. This allows the use of steganography on efficiently learnable coverttext channels and lowers the required complexity to sample stegotext documents.

3.2 Generative Neural Networks

A Generative Neural Network (GNN) establishes a model that approximates realistic distributions such as natural language. In the last years, the artificial intelligence community achieved tremendous progress in building powerful models. Most of them are based on a machine learning technique called *deep learning* (LeCun, Bengio, and Hinton, 2015).

One of the milestones in deep learning based generative models is the development and release of the Generative Pretrained Transformer, or GPT (Radford et al., 2018). Its successors GPT-2 (Radford et al., 2019) and GPT-3 (Brown et al., 2020) quickly became two of the industry-standard models in the natural language processing (NLP) community. While the GPT-n family is mostly known for its capabilities in text generation, it is not specifically designed to generate blog posts or newspaper articles. It can also be used for a variety of other NLP tasks, such as question answering, sentiment detection or automatic text summarization. While GPT-3 can only be used via a fee-based online API, GPT-2 has been open-sourced in 2019 (Solaiman et al., 2019). This makes GPT-2 ideal for use in research projects.

All members of the GPT-n family are attention-based transformer models, which were introduced in (Vaswani et al., 2017). Before transformer models, state of the art machine learning models were often built using recurrent neural networks (RNNs) such as the long-short-term memory model, or LSTM (Hochreiter and Schmidhuber, 1997).

In comparison with RNNs, transformer models achieve significantly higher training and prediction performance. While both RNNs and transformer models parse their inputs as a vector of so-called tokens, they differ in the way they process them to generate an output. In an RNN, the input tokens are parsed one word at a time, while transformer models are able to parse all input tokens at the same time. This is due to the architecture of RNNs. RNNs consist of a loop structure that parses one input token after the other together with a hidden state vector and outputs a hidden state vector for the next iteration.

A transformer model, on the other hand, uses attention and refrains from using a hidden state vector. Instead, for each output token, the transformer model weighs its attentions on the input. This technique mimics the way human attention works. For example, when translating a sentence from one language to another, the translator puts their attention to different parts of the input to determine the correct output word at a given position.

Example 3.9 (Example of LSTM and attention in translation). When translating the English sentence “The cat likes to eat catfood very much.” to German “Die Katze isst sehr gerne Katzenfutter.”, the structure of the output differs significantly from that of the input. In an RNN, the model has to remember the token “eat” until the token “isst” is generated. This information has to be encoded in the hidden state vector while parsing the input and be carried throughout the output generation. Since each output in the RNN alters the hidden state, the translation has to be performed in a serial manner.

In contrast, a transformer model uses attention at each step of output generation. When generating the token “isst”, the model’s attention will predominantly be weighed on the input token “eat”, but some attention will also be placed on “likes” and “the cat” to produce a grammatically correct sentence. Meanwhile, other tokens such as “very” will have low attention since that word is not relevant to generate “isst”.

When we use GPT-2 in the Meteor stegosystem, we can abstract its prediction functionality as a function $GPT: \mathcal{T}^\zeta \rightarrow \mathbb{Z}^{|\mathcal{T}|}$. It takes an input vector $v \in \mathcal{T}^\zeta$ of tokens (with ζ being the history length supported by the model) and transforms it into a *logits vector* of unweighted model outputs (the next token prediction) of dimension $|\mathcal{T}|$.

The input vector $v = T(h)$ is a representation of a history h generated using a tokenizer T . Since we work with text, we need to find a way for T to transform our history h into the realm of numbers, i.e., *tokenize* the history. To do that, we use a set of tokens $\mathcal{T} = \{t_1, t_2, \dots, t_\ell\}$ specific to the model. In the simplest case, the tokens can be Unicode characters, i.e., $\mathcal{T} = \mathcal{U}$. We can then encode a string h with $|h| \leq \zeta$ as a vector $v = T(h) \in \mathcal{T}^{|h|}$, fill the remaining $\zeta - |h|$ entries in the vector with a special padding token ϵ and pass it to GPT .

While the character-based approach could work, it is hard for the model to learn the relationship between words solely on the sequence of characters, which worsens the quality of outputs. Are “do” and “don’t” more closely related than “do” and “to”? Probably yes. But on the character level, “do” and “to” are very close to each other as they only differ in one character. Meanwhile, “do” and “don’t” share a common prefix but differ in three characters (if we pad “do” with padding token ϵ). As we can see, the disadvantages of character-based tokenization are long token sequences and meaninglessness of each token for itself.

We could also use words of the English language as tokens, where each word is represented by a unique token. But this results in large token sets. It is also unlikely that any token set of full words can be exhaustive for all kinds of texts, especially if we take typographical errors or neologisms into account. Also, word-based tokenization makes it hard for a machine learning algorithm to learn relationships between words. Similar words – such as the examples “do” and “to” from above – can have vastly different meanings, while the mistyped word “bananna” probably means the same as “banana”.

To tackle the challenges of natural languages, most modern text generation models use subword tokenization. Subword tokenization uses encoding techniques to find common sequences of characters and establishes these as tokens. One of the commonly used encoding techniques in NLP (and coincidentally the one used in the GPT-n family) is byte-pair encoding, or BPE, which was originally introduced as a data compression algorithm (Gage, 1994). In 2016, it was proposed as an encoding algorithm in neural machine translation (Sennrich, Haddow, and Birch, 2016). With subword tokenization, a token may not only be a single character or a full word of a given language but also a substring of a word. The use of subword tokens allows a model to better predict relationships between words. For example, the words “do” and “don’t” are probably closely related. With subword tokens, a tokenizer can parse “don’t” as “do||n’t”. An NLP model might then learn that the token “n’t” can be used to negate a verb. While this approach makes it easier for the model to learn relationships between different words, subwords tokens are not neces-

sarily prefix-free and therefore there might exist multiple tokenizations for a given word, a pitfall that will be important when we discuss reliability in Chapter 4.

After the tokenizer has encoded the history as a vector of tokens it is passed as input to the generative model. The model's output is a *logits vector* $z \in \mathbb{Z}^{|\mathcal{T}|}$ with components z_i for each token t_i in the model's token set. A logits vector is a vector of unweighted activations of the model's underlying layers, where a higher value in a component represents a stronger activation and therefore a higher likelihood of a token to appear as the next token in the sequence. Since the logits vector is unweighted, it might contain positive and negative values. To get a probability distribution from a logits vector we can use the softmax function σ , which transforms a logits vector of some dimension ℓ into another vector of the same dimension representing the probability mass function for the components of the logits vector:

$$\sigma: \mathbb{Z}^\ell \rightarrow [0, 1]^\ell, \sigma(z) = \left(\frac{e^{z_i}}{\sum_{j=1}^{\ell} e^{z_j}} \right)_{1 \leq i \leq \ell}.$$

When we combine all the previous steps, we get a black box for the ML model we can use as a channel in the Meteor stegosystem, illustrated in Figure 3.2.

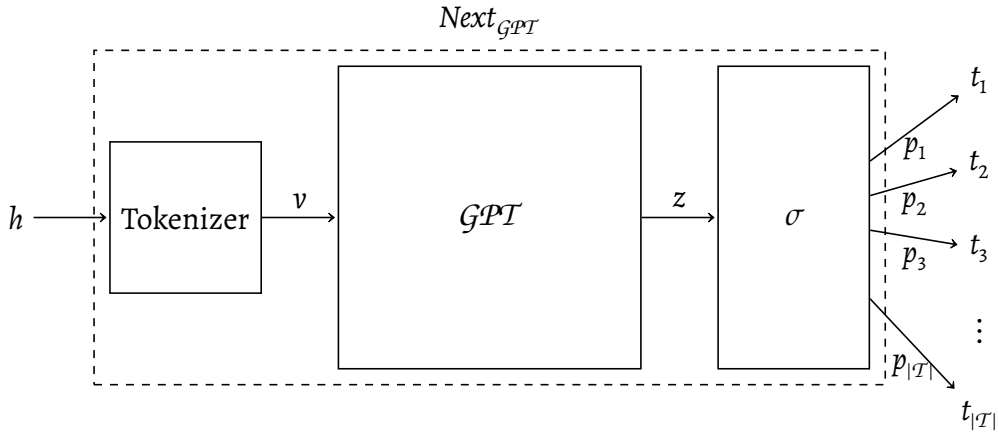


Figure 3.2: Generative ML Model as a channel $C = \text{Next}_{GPT}$ used in the Meteor stegosystem. Next_{GPT} takes a history $h \in \mathcal{H}$ as input and outputs a tuple $(\mathcal{T}, \mathcal{P})$, where $\mathcal{P} = \{p_1, p_2, \dots, p_{|\mathcal{T}|}\}$ is a probability distribution over tokens $\mathcal{T} = \{t_1, t_2, \dots, t_{|\mathcal{T}|}\}$. The history h is a string of text which is transformed to a vector $v \in \mathcal{T}^\zeta$ (with model-specific ζ) of token indices using a tokenizer. The prediction model GPT takes v as input and returns a logits vector $z \in \mathbb{Z}^{|\mathcal{T}|}$ of unscaled outputs. This logits vector is then transformed to a probability distribution $p \in [0, 1]^{|\mathcal{T}|}$ using the softmax function. The component p_i represents the probability of token $t_i \in \mathcal{T}$ to be the next token in the sequence s given the history v of length ζ : $p_i = \Pr[s_{\zeta+1} = t_i | s = v]$.

To generate the next token in the sequence, we can sample according to the generated distribution. After sampling from the distribution, we append the sampled token to the history and repeat the process until sufficiently many tokens have been generated. We

then convert the token outputs to a string using the tokenizer to get the generated text. For an example of GNN model outputs using GPT-2, see Example 3.10.

Example 3.10 (Example of GPT-2 model output for initial history “What is steganography in computer science?”). It is a form of digital communication that uses data to hide or obscure messages from a third party. It has become so common that today it is one of the most popular ways of communicating online. Steganography can be used to send photos or videos to another user or to hide messages from someone who will not discover the sender is actually the sender. As a result, it has become a real phenomenon known as “steganography”.

To use GPT-2 with the Python programming language, we can use the Hugging Face Transformers library together with PyTorch. PyTorch is a machine learning framework that implements high-performance operations needed for machine learning with support for hardware acceleration (The Linux Foundation, 2022). The Hugging Face Transformers library is built upon PyTorch and provides an API for easy access to several pre-trained machine learning models (Hugging Face Inc., 2022). In its simplest form, the pipeline API generates texts using not only GPT-2 but many other generative models. A Python program that generates the text in Example 3.10 can be found in Listing 3.3.

While the use of the pipeline API is very convenient for developing simple text generation or question-answering programs, it abstracts some steps that we will need to intercept for use in a stegosystem. A more sophisticated way to generate text using GPT-2 can be found in Listing 3.4. Its output resembles that shown in Example 3.10. This code example shows how we can:

- use a tokenizer to convert strings to a vector of tokens,
- use the model to get a logits vector,
- transform a logits vector into a probability mass function using the softmax function,
- sample the next token from the distribution and
- convert a vector of tokens back to a string using the tokenizer.

As we can see in Listing 3.4, a pseudorandom value *sample* is generated to select the next token *selection* from the distribution. This is what the Meteor stegosystem will abuse. There, the sending party will replace the randomness with a crafted value to embed a hidden message. While that value seems random to an observer, a recipient who possesses the secret key can recover the hidden message.

Listing 3.3: Python text generator using the Hugging Face Transformers Pipeline API

```
generator = pipeline("text-generation", model='gpt2-large')

print(generator("What is steganography in computer science?",
    max_length=128, do_sample=True)[0]['generated_text'])
```

Listing 3.4: Python text generator using the Hugging Face Transformers library

```
# initialize tokenizer and model for GPT-2
tokenizer = GPT2Tokenizer.from_pretrained('gpt2-large')
model = GPT2LMHeadModel.from_pretrained('gpt2-large')
model.eval()

# tokenize history string using tokenizer, convert to context tensor
# (vector), set up some parameters
context_tokens = tokenizer.encode("What is steganography in computer
    science?")
context = torch.tensor([tokenizer.encoder['<|endoftext|>']] +
    context_tokens, dtype=torch.long)
prev = context
output = context
past = None
max_len = 128
temperature = 0.7
for _ in range(max_len):
    # get model result for next token
    result = model(prev.unsqueeze(0), past_key_values=past)
    logits = result.logits
    past = result.past_key_values
    # preprocess logits: sort, convert to double, scale by temperature
    logits, indices = logits[0, -1, :].sort(descending=True)
    logits = logits.double()
    logits_temp = logits / temperature
    # apply the softmax function to convert logits to probabilities
    probs_temp = F.softmax(logits_temp, dim=0)
    # accumulate probabilities
    cum_probs = probs_temp.cumsum(0)
    # sample from probabilities
    sample = np.random.random_sample()
    selection = (cum_probs > sample).nonzero()[0].item()

    # append selected token to output
    prev = indices[selection].view(1)
    output = torch.cat((output, prev))
    # stop if endoftext was sampled
    if prev == tokenizer.encoder['<|endoftext|>']:
        break

print(tokenizer.decode(output, skip_special_tokens=True))
```

3.3 The Meteor Stegosystem

In their paper (Kaptchuk et al., 2021b), the authors present the Meteor stegosystem, a provably secure steganographic system for realistic distributions approximated by ML models. The central idea behind Meteor is that an encoding party Alice manipulates the randomness used in a sampling scheme to embed a hiddentext in a token sequence. The decoding party Bob receives the generated stegotext from Alice and recovers a prefix of the random value used for sampling. This steganographic protocol is secure if the value replacing the randomness used for sampling is indistinguishable from a random value.

While it is easily possible for the encoding party to manipulate the sampling because they control the randomness, it is intuitively unclear whether the decoding party can extract a prefix of the randomness from the stegotext to recover parts of the hiddentext message. Luckily, with many generative neural networks, such as GPT-2, the decoding party can do that. The Meteor authors introduce the concept of a *Ranged Randomness Recoverable Sampling Scheme*, or RRRSS. This definition formally models the desired behavior described above.

Definition 3.11 (Ranged Randomness Recoverable Sampling Scheme, (Kaptchuk et al., 2021b)). We call $(\text{Sample}_C^\beta, \text{Recover}_C^\beta, C, \beta, \mathcal{H})$ with ppt. algorithms $\text{Sample}_C^\beta: \mathcal{H} \times \{0, 1\}^\beta \rightarrow \Delta$ and $\text{Recover}_C^\beta: \mathcal{H} \times \Delta \rightarrow 2^{\{0, 1\}^\beta}$ a *Ranged Randomness Recoverable Sampling Scheme* (RRRSS) over channel C with precision $\beta \in \mathbb{N}$ and histories \mathcal{H} , if:

1. $\text{Sample}_C^\beta(h, r)$, on history $h \in \mathcal{H}$ and randomness $r \in \{0, 1\}^\beta$, samples an output $s \in \Delta$ from the marginal channel distribution C_h .
2. $\text{Recover}_C^\beta(h, s)$, on history $h \in \mathcal{H}$ and sample $s \in \Delta$ returned from Sample_C^β , outputs the set $\mathcal{R} = \{r \in \{0, 1\}^\beta \mid \text{Sample}_C^\beta(\mathcal{H}, r) = s\}$ of possible values for r .

With the definition of an RRRSS, the authors define sampling and recovery algorithms for the GPT-2 model. While they define the *Sample* and *Recover* algorithms for GPT-2 only, they argue that many generative models satisfy the RRRSS property.

Listing 3.5: RRRSS *Sample* algorithm for GPT-2 (Kaptchuk et al., 2021b). *Sample* produces, given a history h and a value r , the next token sampled according to r from the distribution generated by $C_h = \text{Next}_{\text{GPT}}(h)$.

algorithm $\text{Sample}_{\text{GPT}}^\beta(h, r)$
 Output: Token t
 $\mathcal{T}, \mathcal{P} \leftarrow \text{Next}_{\text{GPT}}(h)$
 $\text{cuml} \leftarrow 0$
for $i \in \{0, 1, \dots, |\mathcal{T}| - 1\}$ **do**
 $\text{cuml} \leftarrow \text{cuml} + \mathcal{P}[i]$
 if $\text{cuml} > r$ **then**
 Output $t \leftarrow \mathcal{T}[i]$
 Output $t \leftarrow \mathcal{T}[|\mathcal{T}| - 1]$

Listing 3.6: RRRSS *Recover* algorithm for GPT-2 (Kaptchuk et al., 2021b). Given a history h and a sample s , *Recover* returns a set of possible random values used to generate s with history h .

```

algorithm  $\text{Recover}_{\text{GPT}}^\beta(h, s)$ 
  Output: Randomness set  $\mathcal{R}$ 
   $\mathcal{T}, \mathcal{P} \leftarrow \text{Next}_{\text{GPT}}(h)$ 
   $\text{cuml} \leftarrow 0$ 
  for  $i \in \{0, 1, \dots, |\mathcal{T}| - 1\}$  do
    if  $\mathcal{T}[i] = s$  then
      Output  $\mathcal{R} \leftarrow \{r \in \{0, 1\}^\beta \mid \text{cuml} \leq r < \text{cuml} + \mathcal{P}[i]\}$ 
       $\text{cuml} \leftarrow \text{cuml} + \mathcal{P}[i]$ 
  Output  $\mathcal{R} \leftarrow \emptyset$ 

```

In the *Sample* algorithm, Next_{GPT} is the channel \mathcal{C} that generates the probability distribution for the next token given a history h as shown in Figure 3.2. The tokens are denoted by \mathcal{T} and probabilities are denoted by \mathcal{P} . The $\text{Sample}_{\text{GPT}}^\beta$ algorithm cumulates the token probabilities p_i into *cuml* until the threshold r has been reached. Once reached, the current token t_i is the token to be sampled from the distribution. We can see that the order of tokens is relevant for the recovery to succeed, so we expect Next_{GPT} to return probabilities in a somehow ordered fashion.

Now that an RRRSS scheme for GPT-2 is established, we can commence with the definition of the stegosystem $\mathcal{S} = (\text{KeyGen}_C^\beta, \text{Encode}_C^\beta, \text{Decode}_C^\beta)$. The authors described these algorithms as follows while introducing helper functions LenPrefix^β and Prefix^β .

Listing 3.7: *KeyGen* algorithm (Kaptchuk et al., 2021b)

```

algorithm  $\text{KeyGen}_C^\beta(1^\lambda)$ 
  Output: Key  $k_{\text{prg}}$ 
  Output  $k_{\text{prg}} \leftarrow \text{PRG.Setup}(1^\lambda)$ 

```

Listing 3.8: *Encode* algorithm (Kaptchuk et al., 2021b)

```

algorithm  $\text{Encode}_C^\beta(k_{\text{prg}}, m, h)$ 
  Output: Stegotext message  $c$ 
   $c \leftarrow \epsilon, n \leftarrow 0, i \leftarrow 0$ 
  while  $n < |m|$  do
     $\text{mask} \leftarrow \text{PRG.Next}(k_{\text{prg}})$ 
     $r \leftarrow m[n : n + \beta] \oplus \text{mask}$ 
     $c_i \leftarrow \text{Sample}_C^\beta(h, r)$ 
     $\mathcal{R} = \text{Recover}_C^\beta(h, c_i)$ 
     $n_i \leftarrow \text{LenPrefix}^\beta(\mathcal{R})$ 
     $c \leftarrow c || c_i, n \leftarrow n + n_i, h \leftarrow h || c_i, i \leftarrow i + 1$ 
  Output  $c$ 

```

Listing 3.9: Decode algorithm (Kaptchuk et al., 2021b)

```

algorithm  $\text{Decode}_C^\beta(k_{\text{prg}}, c, h)$ 
  Output: Plaintext message  $m$ 
   $m \leftarrow \epsilon$ 
  Parse  $c$  as  $c_0 || c_1 || \dots || c_\tau$ 
  for  $i \in \{0, 1, \dots, \tau\}$  do
     $\mathcal{R} = \text{Recover}_C^\beta(h, c_i)$ 
     $m_i \leftarrow \text{Prefix}^\beta(\mathcal{R})$ 
     $\text{mask} \leftarrow \text{PRG.Next}(k_{\text{prg}})$ 
     $m \leftarrow m || (m_i \oplus \text{mask}[0 : |m_i|])$ 
     $h \leftarrow h || c_i$ 
  Output  $m$ 

```

Listing 3.10: LenPrefix algorithm (Kaptchuk et al., 2021b)

```

algorithm  $\text{LenPrefix}^\beta(\mathcal{R} = \{r_1, \dots, r_n\})$ 
  Output: Length  $l$ 
   $l \leftarrow 1$ 
  while  $l < \beta$  do
    if  $\exists i, j \in \{1, \dots, n\}$  such that  $r_i[0 : l] \neq r_j[0 : l]$  then
      Output  $l - 1$ 
     $l \leftarrow l + 1$ 
  Output  $l$ 

```

Listing 3.11: Prefix algorithm (Kaptchuk et al., 2021b)

```

algorithm  $\text{Prefix}^\beta(\mathcal{R} = \{r_1, \dots, r_n\})$ 
  Output: Bit String  $p$ 
  Output  $p \leftarrow r_1[0 : \text{LenPrefix}^\beta(\mathcal{R})]$ 

```

Encode uses the PRG to sample a pseudorandom mask and applies that mask to the next β bits of m into r . Since mask is pseudorandom, r is pseudorandom as well. We then use r to sample from the RRRSS to get the next token c_i . Afterwards, Recover_C^β and LenPrefix^β are queried to get the number of bits n_i recoverable from r . This procedure is repeated until the entire hiddentext has been encoded. Consider that the output of *Encode* is deterministic, i.e., for a given k_{prg} , m , and h , the same stegotext c will be generated, which will be important when we discuss security in Chapter 6.

Decode takes a key k_{prg} , a stegotext c , and history h and decodes c to the hiddentext message. It repeatedly uses Recover_C^β and Prefix^β to recover bits from the pseudorandom value used to sample from the distributions generated by C . It then generates a mask from the PRG (which will be the same mask generated in the encoding step) to decrypt the original message.

For reliability, the authors introduce a definition of *steganographic correctness* that is roughly equivalent to our definition of reliability in Definition 3.6 without giving a formal proof for the Meteor stegosystem. In Chapter 4, we will argue that the Meteor stegosystem is not reliable using GPT-2 as a channel and propose a modification to Meteor that improves reliability while – in the worst case – introducing computational overhead exponential in stegotext length.

In their paper, the Meteor authors provide performance statistics for different models and hardware setups. About three bits of hiddentext can be encoded in a stegotext token when using the GPT-2 model. For other generative models, such as a model that generates HTTP headers or a model trained on Wikipedia articles, the rate is significantly lower at less than one bit of hiddentext per token. The *GPU* setup is a Linux workstation with an Intel i7-6700 CPU, NVIDIA TITAN X GPU, and 8 GiB of RAM. The *CPU* setup is a laptop running Linux with an Intel Core i7-4700MQ CPU, no discrete GPU, and 8 GiB of RAM. The presented results for GPT-2 running on CPU are consistent with our experiments, which were conducted on a Macbook Air M1 (2020) with 16 GiB of RAM.

Model	GPU (sec)	CPU (sec)	Stegotext Length (bytes)	Rate (bits/token)
GPT-2	13.089	82.214	1976	3.09
Wikipedia	19.791	46.583	2002	0.64
HTTP Headers	49.380	103.280	6144	0.21

Table 3.12: Meteor encoding statistics for a 160-byte hiddentext (Kaptchuk et al., 2021b).

Meteor’s security proof is also outlined. In the paper, it is argued that a steganographic protocol is secure if no ppt. adversary can distinguish the output of the stegosystem from the output of a random oracle that samples from the distribution. In Chapter 6, we will show Meteor’s security against chosen hiddentext attackers with a query complexity of one. We will also show that Meteor is insecure against chosen hiddentext attackers with two or more queries.

4

Improving the Reliability of the Meteor Stegosystem

The Meteor authors have released an online demo of Meteor (Kaptchuk et al., 2021a). On the website, they state that “due to issues with the GPT-2 algorithm interface, you sometimes may see extra output from a decoded stegotext. This does not impact the underlying security of the scheme”. After experimentation, we conclude that the cause of this is not due to issues with the algorithm interface but to the way GNNs with subword tokenization such as GPT-2 tokenize text.

In this chapter, we will, after describing the issue at hand, show that the Meteor Stegosystem is indeed unreliable by providing a counterexample. Afterwards, we will discuss an experiment that approximates the probability of incorrect decodings. The experiment shows that for sufficiently long hiddentexts the probability of incorrect decodings increases to approximately 87 %. To improve reliability, we present a change to Meteor’s *Decode* algorithm to increase the probability of successfully recovering the hidden-text message using a DFS-based graph search while – in the worst case – introducing computational overhead exponential in stegotext length.

4.1 Ambiguous Tokenization

In Section 3.2, we discussed that GNNs use a set of tokens to represent text. To improve the quality of the generated text, models (not only) in the GPT-n family use subword tokenization. Subword tokenization is opposed to word tokenization, where text is split by spaces, and character-based tokenization, where text is tokenized as a sequence of characters. Subword tokenization allows for more effective machine learning, as the model can easier connect related words. This technique proved to be very effective in enhancing GNN performance in the last few years.

On the other hand, this allows for one word to have multiple tokenizations because subword tokens are usually not prefix-free. For example, the word “doesn’t” might be tokenized as “do||es||n’t”, but it could also be tokenized as “does||n’t” or even “d||o||e||s||n||’||t”. This situation is what we call *ambiguous tokenization*. If Alice generates a stegotext that has an ambiguous tokenization and Bob choses a different tokenization than Alice, the

decoding will fail because a wrong prefix of the sample used to generate a token will be recovered from the stegotext.

For party $P \in \{A, B\}$ we denote by T_P the tokenizer used to generate (encoding party A) or parse (decoding party B) a stegotext c as a sequence of tokens

$$T_P(c) = t_1 || t_2 || \dots || t_{|t|} \in \mathcal{T}^*.$$

To measure the difference in tokenizations of a given stegotext between Alice and Bob, we define a distance function D that measures the number of mismatching subsequences of two tokenizations $t = T_A(c)$, $t' = T_B(c)$ as follows:

Definition 4.1 (Tokenization Distance). Let $t = t_1 || t_2 || \dots || t_{|t|} \in \mathcal{T}^*$, $t' = t'_1 || t'_2 || \dots || t'_{|t'|} \in \mathcal{T}^*$ be sequences of tokens where $|t| \geq |t'|$ (otherwise, swap t and t'). If $t_1 \neq t'_1$ or $t_{|t|} \neq t'_{|t'|}$ prepend or append, respectively, the *empty token* ϵ to both t and t' . We define the *tokenization distance of t and t'* as a function $D: \mathcal{T}^* \times \mathcal{T}^* \rightarrow \mathbb{N}$ with

$$D(t, t') = |\{i \in [1, |t|-1] \mid \exists i' > 0 \exists l > 0 \exists l' \geq l : t_i = t'_{i'} \wedge \left(\bigwedge_{j=1}^l t_{i+j} \neq t'_{i'+j} \right) \wedge t_{i+l+1} = t'_{i'+l'+1}\}|.$$

Example 4.2 (Tokenization Distance). Let $c = \text{"hello"}$ be a text generated by Alice with tokenization $T_A(c) = \text{he} || \text{l} || \text{lo}$. Let $T_B(c) = \text{he} || \text{llo}$ be the tokenization of c generated by Bob. The distance between $T_A(c)$ and $T_B(c)$ is

$$D(T_A(c), T_B(c)) = D(\text{he} || \text{l} || \text{lo}, \text{he} || \text{llo}) = D(\text{he} || \text{l} || \text{lo} || \epsilon, \text{he} || \text{llo} || \epsilon) = |\{1\}| = 1.$$

4.2 Reliability of the Meteor Stegosystem

In Listing 3.9, we see that during decoding, the stegotext c generated by Listing 3.8 should be parsed as $c = c_0 || c_1 || \dots || c_\tau$. This task is performed by a tokenizer T . Meteor's *Decode* algorithm expects that T can recover the tokens c_i originally generated during encoding. This unfortunately is, at least for models with subword tokenizations such as GPT-2, not the case for some combinations of hiddentext, key and history due to ambiguous tokenization described above. In Section 4.3, we will show how to fix these reliability issues.

Theorem 4.3. *The Meteor stegosystem is not reliable.*

Proof. For Meteor to be reliable, its unreliability must be negligible in λ , i.e.

$$\text{UnRel}_S(\lambda) = \max_{\substack{k \in \{0,1\}^\lambda \\ m \in \mathcal{U}^* \\ h \in \mathcal{H}}} \{\Pr[SD(k, SE(k, m, h), h) \neq m]\} < \mu(\lambda).$$

We show unreliability by finding a tuple (k', m', h') where decoding always fails, i.e.,

$$\Pr[SD(k', SE(k', m', h'), h') \neq m'] = 1.$$

The following counterexample has been generated by repeatedly encoding a fixed message m' and history h' with random keys $k' \leftarrow \$ U_{512}$. The decoding failure is independent of the choice of λ . We will later show that the decoding fails for many or most keys k , especially if the encoded messages m are long ($|m| \geq 1024$ bytes). Let

$k' = \text{0xb95e03a1d01b304f11dcf2bc844e5fd3cbcd41253b0506876004207b2c2a10e2d89c1a40e93530bfcfaaee54e66ae048d2d2a536615b0a81afe792883877d5b6}$
 $m' = \text{"Hello world"}$
 $h' = \text{"Despite a long history of research and wide-spread applications to censorship resistant systems, practical steganographic systems capable of embedding messages into realistic communication distributions, like text, do not exist.\n\n"}$

be the inputs to the encoder. The stegotext when using the Meteor demo code at (Kaptchuk et al., 2021a) is

$c = \text{'\nZeus communication system, controlled by anÆ2 desktop mic with'}$

When passed to the standard GPT-2 tokenizer, the substring “Zeus” of c is parsed as “Z||eus”, while the encoding party has generated “Zeus” with token sequence “Ze||us”. Therefore, the stegotext cannot be successfully decoded to the original hiddentext “Hello world”, which violates reliability. \square

After we have seen that this problem appears at least once by finding a counterexample, another question arises: How often does this happen? If this happens only for a negligible number of combinations of message, key, and history (or maybe even only for this exact combination), this might be a non-issue.

To approach this question, we first define a random variable X as the distance between tokenizations $T_A(c)$ and $T_B(c)$ generated by Alice and Bob, respectively, for a stegotext $c \in \Delta^*$, i.e., for randomly chosen c we define $X = D(T_A(c), T_B(c))$.

As we have argued in Section 4.1, the decoding can only succeed if no tokenization mismatches happen, i.e., the tokenizations generated by Alice and Bob, respectively, are the same. Therefore, for a successful decoding of a stegotext, the tokenization distance must be zero. We can establish the tokenization distance as an upper bound on the probability of a successful decoding:

$$\Pr[\text{Decode}_C^\beta(k, \text{Encode}_C^\beta(k, m, h), h) = m] \leq \Pr[X = 0].$$

Unfortunately, we cannot determine $\Pr[X = 0]$ exactly because there are infinitely many stegotexts $c \in \Delta^*$. Instead, we will estimate $\Pr[X = 0]$ with an experimentally determined approximation $\hat{Pr}_n[X = 0]$ using a finite number of samples $(k, m, h) \in S_n$ with keys $k \in \{0, 1\}^\lambda$ for a fixed security parameter λ , messages m from some set of text samples $M_n \subset \mathcal{U}^n$ for message lengths n and histories $h \in \mathcal{H}$. For each sample in S_n , we generate stegotexts $c = \text{Encode}_C^\beta(k, m, h)$ and calculate the tokenization distances $D(T_A(c), T_B(c))$. Afterwards, we can establish the approximation

$$\hat{Pr}_n[X = 0] = \frac{|\{(k, m, h) \in S_n \mid D(T_A(\text{Encode}_C^\beta(k, m, h)), T_B(\text{Encode}_C^\beta(k, m, h))) = 0\}|}{|S_n|}.$$

For this experiment, we use William Shakespeare’s drama Hamlet as the source for messages and histories. For the message set M_n , we use blocks of length $n = 128$ and 1024 bytes from the script of Hamlet, respectively. The entire script of Hamlet is 191726

bytes long. That gives us $|M_{128}| = \lceil \frac{191726}{128} \rceil = 1498$ and $|M_{1024}| = \lceil \frac{191726}{1024} \rceil = 188$ samples for m . Even though the sample size for blocks of 1024 bytes is rather small, the average stegotext length grows proportionally to the number of bits encoded, so we can expect about the same total length of token sequences for both experiments. For each hiddentext message, we generate a random key $k \leftarrow U_{512}$ as well as an initial history h of length 128 bytes randomly chosen from the script of Hamlet.

After defining the sample data, we encode and subsequently decode each sample using a modified version of the Meteor demo code from (Kaptchuk et al., 2021a). After decoding, we compute the tokenization distances between Alice and Bob for each sample.

The experiment shows that most stegotexts have at least one mismatch if they are of significant length (Figure 4.1). When encoding 128 bytes of hiddentext, the success probability is $\hat{Pr}_{128}[X = 0] \approx 0.57$. With hiddentext lengths of 1024 bytes, it decreases even further to $\hat{Pr}_{1024}[X = 0] \approx 0.13$. It is reasonable to expect that the probability of a successful decoding converges to zero for even longer hiddentexts.

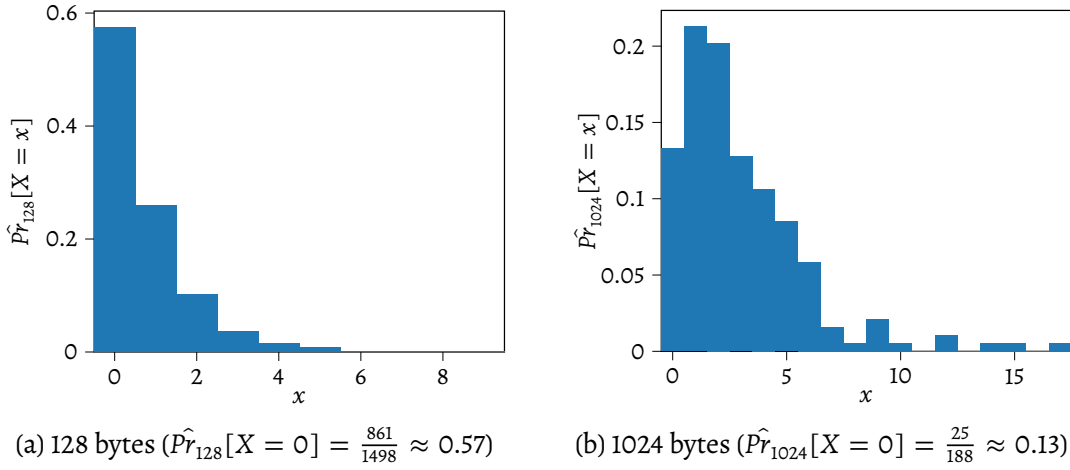


Figure 4.1: Tokenization mismatch count probability when using Meteor to encode Shakespeare’s Hamlet block-wise for different block sizes. We denote the random variable X as the number of encoding mismatches in a stegotext. The x-axis shows the number of tokenization mismatches between the encoding and the decoding party for hiddentext blocks of size...

Therefore, we have to find a way to deal with these failed decodings, especially if we plan to encode hiddentext of significant length such as HTTP messages. In Section 4.3, we will present an approach to recover from decoding errors while introducing stegotext overhead linear in hiddentext length and – in the worst case – computational overhead exponential in stegotext length. In Chapter 5, we will show how we can split the hiddentext message and use DialoGPT – a derivation of GPT-2 – to generate chat-like stegotext. Splitting the hiddentext results in shorter stegotexts and thus decreases the probability of tokenization mismatches.

4.3 Algorithmic Reconstruction of Tokenization Candidates

Unfortunately, with subword tokenization, the decoding party cannot decide which tokens have been used to generate a given stegotext. To allow successful decoding of ambiguously tokenized stegotexts, we will introduce modifications to the *Encode* and *Decode* algorithms to detect and fix wrong tokenizations.

Before encoding, split the message m into blocks m_i of length γ . After each block m_i , add a checksum block $q_i = q(m_i)$ of length δ into the hiddentext, i.e., $q: \{0, 1\}^\gamma \rightarrow \{0, 1\}^\delta$. This value helps the decoder to decide if the decoding is still correct up to block i . By adding checksum blocks, we introduce $\delta \cdot \frac{|m|}{\gamma}$ bits of overhead to the hiddentext. For the modified *MarkedEncode* algorithm, see Listing 4.2.

Now, the decoding party Bob has to verify that after each block m_i of γ bits, the checksum $q(m_i)$ of length δ appears. If not, a tokenization mismatch occurred. To recover from this, Bob has to roll back his decoding and generate all possible tokenizations of a substring of c starting with the position in c after his last successful checksum check. The BPE tokenizer used in GPT-2 does not contain tokens that span word boundaries, i.e., no token contains inner spaces. Therefore, we can assume that a tokenization mismatch ends when the current word does. When a tokenization with correct checksums was found, the decoder expects that the correct tokenization has been found and proceeds. Note that this approach will not work with models that use language independent tokenizers such as SentencePiece (Kudo and Richardson, 2018), which tokenizes strings as raw sentences and treats spaces as regular characters. With language independent tokenizers, the entire suffix of c starting with the block failing the checksum has to be considered, potentially introducing overhead exponential in stegotext length.

It is still possible that a wrong tokenization randomly yields a correct checksum. In an actual implementation, one might want to check multiple blocks in advance before expecting that the correct tokenization has been found and choose γ and δ depending on the security parameter λ such that the probability of a false-positive checksum is negligible. Determining specific values for γ and δ is a trade-off between stegotext length and computational overhead in the case of a decoding error. While longer checksums δ potentially decrease the risk of false-positive checksum checks, longer checksums cause longer stegotexts. If we chose $\gamma = |m|$, only one checksum is generated over the entire message. If an error is detected after decoding the entire hiddentext, every possible tokenization for the stegotext c has to be generated of which there are exponentially many in stegotext length. If we choose a small γ , many checksums are inserted, drastically increasing stegotext length. A modified *Decode* algorithm can be found in Listing 4.3.

Listing 4.2: *MarkedEncode* algorithm. The modification this algorithm introduces is that after every block m_i of γ bits a checksum $q(m_i)$ is inserted into the hiddentext. This allows the recipient to check for decoding errors after decryption. q is a function $q: \{0, 1\}^\gamma \rightarrow \{0, 1\}^\delta$ that generates a checksum of length δ for a message block of length γ .

algorithm $\text{MarkedEncode}_C^{\beta, \gamma, \delta}(k_{\text{prg}}, m, h, q)$

Output: Stegotext message c

$c \leftarrow \epsilon, n \leftarrow 0, i \leftarrow 0$

```

Parse  $m$  as blocks  $m_1 || m_2 || \dots || m_{\xi}$  of size  $\gamma$ 
 $m^* \leftarrow m_1 || q(m_1) || m_2 || q(m_2) || \dots || m_{\xi} || q(m_{\xi})$ 
while  $n < |m^*|$  do
     $mask \leftarrow PRG.Next(k_{prg})$ 
     $r \leftarrow m^*[n : n + \beta] \oplus mask$ 
     $c_i \leftarrow Sample_C^{\beta}(h, r)$ 
     $\mathcal{R} = Recover_C^{\beta}(h, c_i)$ 
     $n_i \leftarrow LenPrefix^{\beta}(\mathcal{R})$ 
     $c \leftarrow c || c_i, n \leftarrow n + n_i, h \leftarrow h || c_i, i \leftarrow i + 1$ 
Output  $c$ 
    
```

Listing 4.3: *MarkedDecode* algorithm. In comparison to Meteor’s *Decode* algorithm, *MarkedDecode* verifies the checksums $q(m_i)$ of blocks m_i . If the checksum does not match, a decoding error occurred. It then performs a lookbehind on the stegotext and generates all possible tokenizations *paths* for a substring of c . Afterwards, it rewinds the internal state and retry decoding with a path p selected from *paths*.

```

algorithm MarkedDecode $_C^{\beta, \gamma, \delta}(k_{prg}, c, h, q)$ 
    Output: Plaintext message  $m$ 
     $m^* \leftarrow \epsilon, n \leftarrow 0, j \leftarrow 0, \alpha^* \leftarrow 0$ 
     $paths \leftarrow \emptyset$ 
    Parse  $c$  as  $c_0 || c_1 || \dots || c_{\tau}$ 
    for  $i \in \{0, 1, \dots, \tau\}$  do
         $\mathcal{R} = Recover_C^{\beta}(h, c_i)$ 
         $n_i \leftarrow LenPrefix^{\beta}(\mathcal{R})$ 
         $n \leftarrow n + n_i$ 
         $j \leftarrow j + n_i$ 
         $m_i \leftarrow Prefix^{\beta}(\mathcal{R})$ 
         $mask \leftarrow PRG.Next(k_{prg})$ 
         $m^* \leftarrow m^* || (m_i \oplus mask[0 : |m_i|])$ 
         $h \leftarrow h || c_i$ 
        if  $j \geq \gamma + \delta$ 
            # calculate checksum
            Parse  $m^*$  as  $m_1^* || q_1^* || m_2^* || q_2^* || \dots || m_{\alpha^*}^* || q_{\alpha^*}^* || \dots || m_{\alpha}^* || q_{\alpha}^*$  ignoring suffix of length  $< \gamma + \delta$ 
            if  $q_{\alpha^*+1}^* \neq q(m_{\alpha^*+1}^*) \vee q_{\alpha^*+2}^* \neq q(m_{\alpha^*+2}^*) \vee \dots \vee q_{\alpha}^* \neq q(m_{\alpha}^*)$ 
                Find first index  $j^* > i^*$  such that  $c_{j^*}$  starts with space
                 $c^* \leftarrow c_{i^*} || c_{i^*+1} || \dots || c_{j^*-1}$ 
                if  $paths = \emptyset$ 
                     $paths \leftarrow AllPaths(TokenizeCandidates_T(c^*), c^*)$ 
                 $p \leftarrow SelectPath(paths)$ 
                 $paths \leftarrow paths \setminus p$ 
                replace  $c_{i^*} || c_{i^*+1} || \dots || c_{j^*-1}$  in  $c$  with  $p$ 
                rewind PRG state and variables to state at  $i^*$ 
                retry decoding starting at  $m_{\alpha^*+1}$  with  $p$ 
    
```

```

else # checksum verified, expect first  $\alpha$  blocks and  $i$  tokens to be correctly decoded
   $j \leftarrow j \bmod (\gamma + \delta)$ 
   $paths \leftarrow \emptyset$ 
   $\alpha^* \leftarrow \alpha$ 
   $i^* \leftarrow i$ 
Parse  $m^*$  as  $m_1^* || q(m_1^*) || m_2^* || q(m_2^*) || \dots || m_\xi^* || q(m_\xi^*)$ 
Output  $m \leftarrow m_1^* || m_2^* || \dots || m_\xi^*$ 

```

For the modifications in *MarkedDecode*, we need helper algorithms *TokenizeCandidates_T*, *AllPaths* and *SelectPath*.

When passed a string c , *TokenizeCandidates_T* generates a directed acyclic graph, or DAG, $G = (V, E)$, representing the possible tokenizations of c with tokens \mathcal{T} . The nodes V represent all reachable suffix strings of c (including the empty string ϵ) using tokens from \mathcal{T} with $|V| \leq |c| + 1$. The edges E represent tokens used to transition between suffixes of c . For an example graph for input “hello”, see Figure 4.5. For example, “hello” can be transformed to “lo” with token “hel”.

AllPaths takes a graph $G = (V, E)$ generated by *TokenizeCandidates_T*(c) for a stego-text c and a start node $v_i \in V$ as inputs and returns a set of all possible paths between v_i and a fixed sink $v_j = \epsilon$ (the empty suffix). Each path in that list represents a possible tokenization of c . But how many paths exist between v_i and v_j ? A path between two vertices is a set $V' \subseteq V$ that contains both v_i and v_j and where there exist edges in E from topologically greater vertices to topologically smaller vertices in V' . In a complete DAG, there are up to $2^{|V|-2} \leq 2^{|c|-1}$ subsets of V which contain both v_i and v_j . Therefore, the output size of *AllPaths* is, in the worst case, exponential in input length. For a DFS-based implementation of *AllPaths* see Listing 4.6.

Lastly, *SelectPath* selects a path from a list of *paths* generated by *AllPaths*. A simple strategy is to choose one element at random from *paths*. With this approach, we will find the correct tokenization on average after $\frac{|paths|}{2} \leq \frac{2^{|c|-1}}{2} = 2^{|c|-2}$ attempts. While there are more advanced strategies to select a tokenization, e.g. by selecting a tokenization according to the probability distribution generated by the ML model, this simple approach is still viable when using the English language since we only generate paths for single words that are relatively short. As has been shown by analyzing books from different epochs, the average word length in the English language is about five characters (Bochkarev, Shevlyakova, and Solovyev, 2012). Therefore, an average English word can have up to $2^4 = 16$ possible tokenizations. There still are commonly used (and therefore generated) words with a length of up to 13 characters such as “international” or “circumstances”, which could have up to $2^{12} = 4096$ possible tokenizations. Our experiments have shown that when using the GPT-2 tokenizer, the graphs for common longer words in the English language are still relatively small with less than 2^{11} possible paths. For an implementation of *SelectPath* that chooses a path at random, see Listing 4.7.

Listing 4.4: *TokenizeCandidates* algorithm. This generates a graph $G = (V, E)$ from a string c . Vertices are substrings of c , each edge represents a token used to transform between substrings. This algorithm can be sped up by using dynamic programming to cache results of invocations of *TokenizeCandidates*.

algorithm $\text{TokenizeCandidates}_{\mathcal{T}}(c)$

Output: Graph $G = (V, E)$

if $c = \epsilon$

return (\emptyset, \emptyset)

for $t \in \{t' \in \mathcal{T} \mid t' \text{ is prefix of } c\}$ do

$V \leftarrow V \cup \{c[t:] \}$

$E \leftarrow E \cup \{(c, c[t:])\}$

$G \leftarrow G \cup \text{TokenizeCandidates}_{\mathcal{T}}(c[t:])$

return G

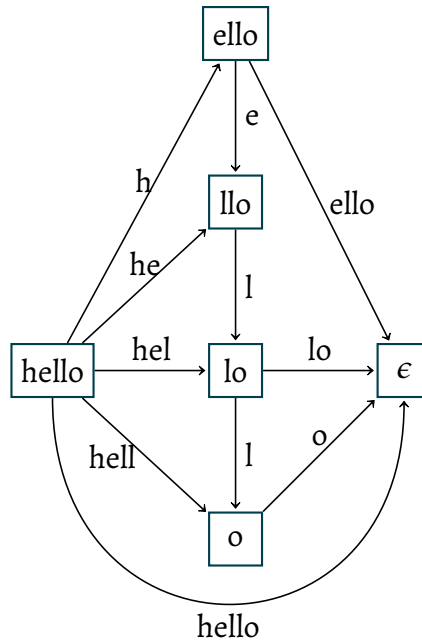


Figure 4.5: Tokenization graph generated by $\text{TokenizeCandidates}_{\mathcal{T}}(\text{"hello"})$ with tokens $\mathcal{T} = \{h, e, l, o, he, lo, hel, hell, ello, hello\}$. The vertices represent substrings of $c = \text{"hello"}$ that are reachable by removing a prefix token $t \in \mathcal{T}$ from c . The edges are labeled with the token $t \in \mathcal{T}$ used to transform the left-hand side vertex v_i to the right-hand side vertex v_j , i.e., $v_i = t||v_j$. The special node ϵ represents the empty string and is a sink in the tokenization graph. The list of all possible paths between c and ϵ are the possible tokenizations of c using tokens \mathcal{T} .

Listing 4.6: DFS-based *AllPaths* algorithm that generates a list of all possible paths between a root and a sink ϵ in a DAG. This algorithm's performance can be sped up by using dynamic programming to cache results of invocations of *AllPaths*.

```

algorithm AllPaths( $G, \text{root}$ )
  Output: List of paths between root and sink  $\epsilon$ 
  if  $\text{root} = \epsilon$  do
    return  $\emptyset$ 
  paths  $\leftarrow \emptyset$ 
  hops  $\leftarrow \text{OutEdges}(G, \text{root})$ 
  for hop  $\in$  hops do
    subpaths  $\leftarrow \text{AllPaths}(G, \text{hop})$ 
    for subpath  $\in$  subpaths do
      paths  $\leftarrow \text{paths} \cup \{\text{root}||\text{subpath}\}$ 
  return paths

```

Listing 4.7: *SelectPath* algorithm with random path selection strategy.

```

algorithm SelectPath( $\text{paths} = \{p_1, p_2, \dots, p_{|\text{paths}|}\}$ )
  Output: path  $p \in \text{paths}$ 
   $i \leftarrow \$\{1, 2, \dots, |\text{paths}|\}$ 
  paths  $\leftarrow \text{paths} \setminus \{p_i\}$ 
  return  $p_i$ 

```

5

Adapting the Meteor Stegosystem for Two-Way Communication

In the Prisoners' Problem (Simmons, 1984), a unidirectional communication is described. The sending party Alice embeds a hidden message in a covert channel, the receiving party Bob decodes the received message. The Meteor stegosystem adopts this approach of sending and receiving party. But in most practical applications, a bidirectional communication channel between participants is needed. For example, in client-server architectures such as HTTP or key exchange protocols such as Diffie-Hellman, the participants exchange messages bidirectionally. Also, one of the most commonly used mediums for text messages is instant messaging, which is characterized by a continuous back-and-forth of messages between participants.

Since the Meteor stegosystem is constructed to be very adaptable to different underlying distributions (as long as they are RRRSS), we can modify Meteor to generate stegotext messages for use in the context of instant messaging. In this chapter, we will introduce a simple protocol for two-way communication using Meteor and adapt Meteor to use DialoGPT (Zhang et al., 2020), a GPT-based generative model that is trained to generate text messages as they could appear in a chat.

5.1 Protocol for Steganographic Two-Way Communication

In Figure 5.1, we introduce a simple chat protocol between two participants Alice and Bob. Here, Alice and Bob send messages back and forth. Alice sends stegotext blocks, while Bob generates a sequence of responses, which can be stegotexts or handwritten messages. After each message from Alice, Bob uses the Meteor stegosystem to decode s_i to a message block m_i . After ℓ iterations, the entire stegotext has been sent. Now, Bob can recover the hiddentext m by concatenating the decoded blocks m_i .

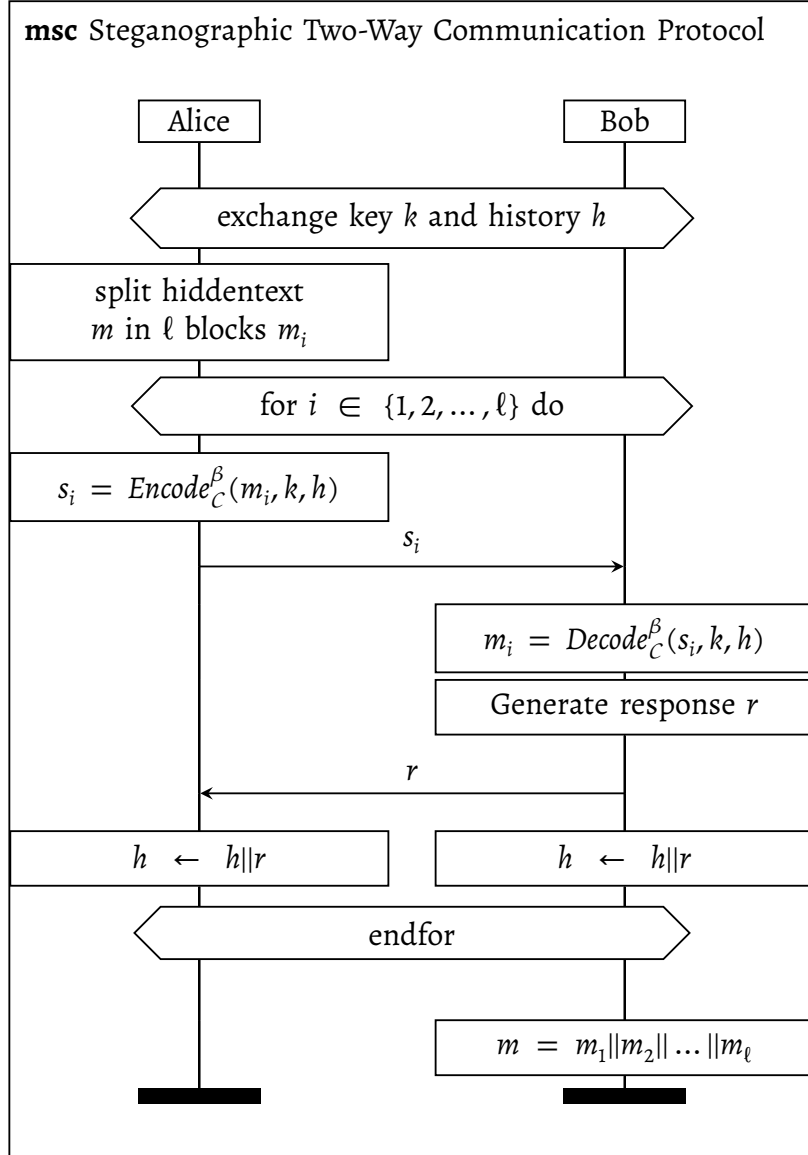


Figure 5.1: Steganographic two-way communication protocol for participants Alice and Bob. In this scheme, Alice encodes blocks of her hiddentext message m into stegotexts s_i and sends them to Bob, who decodes the blocks using the Meteor stegosystem. Bob then generates his response r and sends it back to Alice. The response can be generated using a GNN, be (part of) a stegotext, or handwritten chat messages. After ℓ iterations, Bob can reconstruct the hiddentext m by concatenating the m_i .

5.2 Implementing Steganographic Two-Way Communication

While we can generate stegotexts for two-way communication using the GPT-2 model, as the original Meteor stegosystem does, the texts generated with GPT-2 are not convincing as chat messages. In the following example, we see a conversation between Alice and Bob using the protocol described in Figure 5.1. The initial history consists of a short sequence of messages where both participants greet and ask each other how their respective day was. Alice's messages encode the hiddentext "Hi there!" in blocks of 32 bits, while Bob's responses are generated manually.

Example 5.1 (Meteor Conversation for hiddentext "Hi there!" using GPT-2 with 32 bit block-size).

- Alice: I've been trying to get my hands on a couple of these since they were first released last year, and it's been a fun time. It's been a long time since I've had the chance to play through
- Bob: What games do you like to play?
- Alice: "It's a long haul, but it's a long haul," said Steve Young, the former director of the FBI's counter
- Bob: Are you affiliated with the FBI?
- Alice: How To Use

As we can see, the transcript does not read as if it was an actual conversation between friends. Also, the stegotexts appear to lack context. Alice's texts don't seem to take Bob's responses into account and cover quite different topics in each iteration. This effect is due to the nature of GPT's training dataset called WebText, which is a large undisclosed collection of online newspaper articles, books, blog posts, and the like. Therefore, it is very likely that GPT generates probability distributions for article-like texts. When sampling from these distributions, it is likely to generate text that reads like a newspaper article as well.

Fortunately, the NLP community is very active in building different models for many use cases. In 2020, DialoGPT, a pre-trained transformer trained on public multi-turn dialogues from Reddit threads, has been published (Zhang et al., 2020). As the name suggests, DialoGPT is a derivation of GPT that is trained and fine-tuned for dialogue generation. When we repeat the same experiment as above, but with the DialoGPT model as a drop-in replacement for GPT-2, the results read more like what we would expect from a chat conversation between friends. Again, Alice's text messages encode the hiddentext "Hi there!" with 32 bits of hiddentext per chat message. Bob's responses are handwritten. While a little bit incoherent, the transcript is closer to an actual conversation between friends.

Example 5.2 (Meteor Conversation for hiddentext "Hi there!" using DialoGPT with 32 bit block-size).

- Alice: I suppose it is . I'm a bit jealous of the weather today , though . It's nice , but it's nice to be in a nicer
- Bob: I don't know, it's too hot for me today to be honest :D

5 Adapting the Meteor Stegosystem for Two-Way Communication

- Alice: Yeah it is . I wish it was cooler though . I love it here . I wish I had another place to live though . We're all pretty cool here . Lol
- Bob: haha you're cool for sure
- Alice: I love it here as
- Bob: Where are you currently?

6

Improving the Security of the Meteor Stegosystem

When we communicate using a steganographic protocol, security is of utmost importance. We probably communicate in an environment that is very hostile against its parties. Therefore, we should satisfy the most rigorous security requirements we can.

In Definition 3.8, we established the notion of steganographic security against chosen hiddentext attacks (SS-CHA). In this chapter, we will show that the Meteor stegosystem is secure against SS-CHA adversaries with a query complexity of one, i.e., adversaries that are allowed to send up to one query to the encoding oracle. Afterwards, we will argue that Meteor is insecure against adversaries with query complexity greater than one. In conclusion, we will modify the Meteor stegosystem to improve its security against SS-CHA adversaries by replacing the one-time pad used in Meteor with a symmetric encryption scheme in counter mode (SES-CTR).

6.1 Security Against Chosen Hiddentext Attackers

In Definition 3.8, we introduced Hopper’s notion of steganographic security against chosen hiddentext attackers (SS-CHA). In this chapter, we will show that the Meteor stegosystem is secure against ppt. SS-CHA adversaries with query complexity of one and insecure against ppt. SS-CHA adversaries with query complexity greater than one.

Theorem 6.1. *Meteor is secure against ppt. SS-CHA adversaries with query complexity of one.*

Proof. We show SS-CHA security by reduction to an attack against the underlying PRG. Let \mathcal{A} be a ppt. SS-CHA attacker against the Meteor stegosystem, i.e., \mathcal{A} can distinguish the output of ST , i.e., $Encode_C^\beta$, from that of CT , an oracle randomly sampling from the RRRSS \mathcal{C} , with one query (m, h) .

We will now show that \mathcal{A} can be used to attack the PRG. For the given query (m, h) , the attacker gets the response $c = c_0 || c_1 || \dots || c_\tau$ from ST or CT , i.e., a stegotext from $Encode$ or a random sequence of samples from the RRRSS, respectively, depending on the challenger’s random bit. The attacker can then use $Prefix$ and $LenPrefix$ (as Bob would during decoding) to recover the bit string r used to sample from the channel, since these functions don’t need the secret key k .

How is r constructed if the challenger used ST to generate c ? It is the exclusive-or of β bits of m and a pseudorandom value $mask$ from a PRG. Since $mask$ is pseudorandom, the exclusive-or of $mask$ and some other value is pseudorandom as well. To distinguish the output of ST from that of CT , the attacker has to distinguish r from a randomly sampled token, i.e., the attacker has to distinguish r from a truly random value r' . Therefore, \mathcal{A} can be used to attack the PRG. Since we expect the underlying PRG to be secure, \mathcal{A} cannot distinguish r from r' with non-negligible probability in polynomial time. \square

But is this sufficient to satisfy Hopper’s definition of steganographic security? Unfortunately not. In Hopper’s (t, q, l) -SS-CHA game, an attacker may perform up to $q(\lambda)$ queries. The aforementioned security proof only considers security against one query.

Theorem 6.2. *Meteor is insecure against ppt. SS-CHA adversaries with query complexity greater than one.*

Proof. Let \mathcal{A} be an SS-CHA adversary with at least two queries. Let \mathcal{A} repeat the same query (m, h) twice. While the oracle CT will – with overwhelming probability – sample an entirely different coverttext from the distribution, ST will output the same stegotext twice. After only two queries, \mathcal{A} is able to distinguish Meteor’s outputs from those of a random oracle with high certainty.

We can verify this using the Meteor online demo (Kaptchuk et al., 2021a). For example, let

```
m = "Hello world"
```

h = "Despite a long history of research and wide-spread applications to censorship resistant systems, practical steganographic systems capable of embedding messages into realistic communication distributions, like text, do not exist.\n\n"

be a query to the Meteor online demo. Besides a key, the Meteor online demo requires a nonce, which is neither part of the stegotext nor contained in the protocol, but has to be shared between participants. Therefore, we expect the nonce to be part of the key. For the sake of this example, let $k = (key, nonce)$ be given as follows:

```
key      = 0x0101010101010101010101010101010101010101010101010101010101010101
          0101010101010101010101010101010101010101010101010101010101010101
nonce    = 0x0303030303030303030303030303030303030303030303030303030303030303
          0303030303030303030303030303030303030303030303030303030303030303
```

The specific choice of k is not relevant, as the Meteor stegosystem is deterministic for any k . The stegotext generated for this combination (k, m, h) is

c = "- The great strength of steganography is the ability to send messages without any significant communication layer of encryption, Pinnew"

while random sampling from the distribution would – with overwhelming probability – yield a different covertext for each invocation of the encoding oracle. Therefore, \mathcal{A} has significant advantage against the stegosystem after two queries. \square

In the following section, we will propose a modification to Meteor that improves the security against SS-CHA adversaries by randomizing the stegotext returned by *Encode*.

6.2 Improving Meteor's Security Against Chosen Hiddentext Attackers

To improve Meteor's security, we have to randomize the output of *Encode*, i.e., on each invocation with the same inputs *Encode* should, with overwhelming probability, return a different stegotext while retaining indistinguishability from random sampling.

To achieve that we introduce a new cryptographic primitive, a symmetric encryption scheme in counter mode with a pseudorandom function F (SES-CTR or simply *SES*). We will introduce a definition of *SES* which is equivalent to that introduced in (Berndt, 2017). There, it is argued that *SES* satisfies the IND $\$$ -CPA property if F is a secure pseudorandom function.

The idea behind encryption in *SES* is simple: First, split the message m into n blocks of size $F.in(\lambda)$. Choose an IV of size $F.in(\lambda)$ at random. It is important not to reuse an IV with a given key k to retain one-time pad security. Now, for each block m_i , use $F.Eval$ to generate a pseudorandom value $r = F.Eval(k, IV + i)$. Afterwards, calculate the exclusive-or of r and m_i to get the next block c_i . The ciphertext of size $(n + 1) \cdot F.in(\lambda)$ is the IV concatenated with ciphertext blocks c_i .

Decryption is similar to encryption. First, split c into $n + 1$ blocks of size $F.in(\lambda)$. Interpret the first block of c as IV . For each block c_i , calculate $r = F.Eval(k, IV + i)$. Then, calculate the exclusive-or of r and c_i to recover the message block $m_i = c_i \oplus r$.

Listing 6.1: *SES.Encrypt_F* algorithm

```

algorithm SES.EncryptF( $k, m = m_1 m_2 \dots m_n \in \{0, 1\}^{n \cdot F.in(\lambda)}$ )
Output: Ciphertext  $c$ 
 $\lambda = |k|$ 
 $IV \leftarrow \$ \{0, 1\}^{F.in(\lambda)}$ 
for  $i \in \{1, 2, \dots, n\}$  do
   $c_i \leftarrow F.Eval(k, (IV + i) \bmod 2^{F.in(\lambda)}) \oplus m_i$ 
Output  $IV || c_1 || c_2 || \dots || c_n$ 

```

Listing 6.2: *SES.Decrypt_F* algorithm

```

algorithm SES.DecryptF( $k, c = c_0 c_1 \dots c_n \in \{0, 1\}^{(n+1) \cdot F.in(\lambda)}$ )
Output: cleartext  $m$ 
 $\lambda = |k|$ 
 $IV \leftarrow c_0$ 
for  $i \in \{1, 2, \dots, n\}$  do
   $m_i \leftarrow F.Eval(k, (IV + i) \bmod 2^{F.in(\lambda)}) \oplus c_i$ 
Output  $m_1 || m_2 || \dots || m_n$ 

```

In Listing 6.3, we use *SES* to encrypt the hiddentext message. Since IV as well as all blocks c_i are indistinguishable from randomness, the entire output of *SES.Encrypt_F* is in-

distinguishable from randomness. In Listing 6.4, *SES* is used to decrypt the hiddentext. Since $SES.Encrypt_F$ yields different ciphertexts on each invocation (even with the same message m and key k because IV is sampled at random on each invocation), an attacker can only distinguish $r \leftarrow SES.Encrypt_F(k, m)$ from randomness if an IV is reused for the same pair (m, k) , which is very unlikely.

These modifications improve Meteor's security against SS-CHA adversaries. Repeated invocations of *EncodeRandomized* return different stegotexts with overwhelming probability. Since *SES* is IND\$-CPA secure, we can, with the same reduction as in Theorem 6.1, show SS-CHA security against polynomial attackers as long as the same IV is not generated twice, which is very unlikely.

Listing 6.3: *EncodeRandomized* algorithm. This algorithm differs from the original *Encode* algorithm by using *SES* to encrypt the message. The ciphertext (which is indistinguishable from randomness and nondeterministic) is then used to sample from C_h .

```

algorithm EncodeRandomized $_C^\beta(k_{prg}, m, h)$ 
  Output: Stegotext message c
   $c \leftarrow \epsilon, i \leftarrow 0, n \leftarrow 0$ 
   $r \leftarrow SES.Encrypt_F(k, m)$ 
  while  $n < |r|$  do
     $c_i \leftarrow Sample_C^\beta(h, r[n : n + \beta])$ 
     $\mathcal{R} \leftarrow Recover_C^\beta(h, c_i)$ 
     $n_i \leftarrow LenPrefix^\beta(\mathcal{R})$ 
     $c \leftarrow c || c_i, n \leftarrow n + n_i, h \leftarrow h || c_i, i \leftarrow i + 1$ 
  Output c

```

Listing 6.4: *DecodeRandomized* algorithm. This algorithm differs from the original *Decode* algorithm by using *SES* to decrypt the stegotext.

```

algorithm DecodeRandomized $_C^\beta(k_{prg}, c, h)$ 
  Output: Plaintext message m
   $m \leftarrow \epsilon, n \leftarrow 0, r \leftarrow \epsilon$ 
  Parse c as c0 || c1 || ... || cτ
  for  $i \in \{0, 1, \dots, \tau\}$  do
     $\mathcal{R} \leftarrow Recover_C^\beta(h, c_i)$ 
     $r_i \leftarrow Prefix^\beta(\mathcal{R})$ 
     $r \leftarrow r || r_i$ 
     $h \leftarrow h || c_i$ 
   $m \leftarrow SES.Decrypt_F(k, r)$ 
  Output m

```

7

Conclusion

We have discussed Meteor, an innovative stegosystem that uses generative neural networks to embed a hiddentext into a covert channel. It achieves that by replacing the randomness used for sampling with an encryption of the hiddentext.

If the encrypted hiddentext is indistinguishable from randomness, this process is secure against chosen hiddentext attackers. In Chapter 6, we have seen that this is the case if an attacker is allowed to send up to one query in the SS-CHA game. Unfortunately, Meteor’s security guarantee does not hold for attackers who can send multiple queries, because Meteor’s output is deterministic. To fix this, we discussed a modification to Meteor that uses a symmetric encryption scheme (SES) in counter mode to become indeterministic while maintaining indistinguishability from randomness.

We have also modified Meteor for two-way communication for use in the context of instant messaging in Chapter 5. To use Meteor in instant messaging, the hiddentext is split into blocks to generate shorter stegotext messages. While the stegotexts generated with GPT-2 are not very convincing as chat messages, we can easily adapt Meteor to use other GNN models, as long as they satisfy the RRRSS property defined in Definition 3.11. We then adapted Meteor to use the DialoGPT model. The stegotexts generated with the DialoGPT model more closely resemble those found in instant messaging.

We also found that Meteor sometimes generates stegotexts that cannot be decoded correctly in Chapter 4. These decoding errors happen more frequently the longer the hiddentext is. This problem occurs due to subword tokenization, a feature of modern GNNs which greatly increases model performance. We have discussed a modification to the Meteor stegosystem which fixes these decoding errors while – in the worst case and for some models – introducing computational decoding overhead exponential in stegotext length. This modification first tries to detect a decoding error with a checksum of length δ for message blocks of length γ and, if a decoding error occurs, generates every possible tokenization of a part of the stegotext. Further research could compare different values for γ and δ to find a good trade-off between stegotext length and computational overhead or develop more efficient techniques to recover from decoding errors.

While the Meteor authors have discussed that the Meteor stegosystem will be adaptable to GPT-3 once it is released, OpenAI, unfortunately, did not yet decide to make their model publicly available and only allows limited access to GPT-3 using a private (and fee-based) online API. While it would technically still be possible to use this private API with

Meteor, its practical use in a stegosystem is limited. Also, the occurring costs of US\$ 1.12 to US\$ 55.92 per megabyte of hiddentext (US\$ 0.0004 to US\$ 0.02 per 1000 tokens with an average of 3 bits of hiddentext per token) are a problem in practical use (OpenAI L.L.C., 2022).

Fortunately, researchers have built and trained open-source alternatives to GPT-3. One alternative might be GPT-NeoX, a generative model with 20 billion parameters that can arguably compete with GPT-3 in prediction quality (Black et al., 2022). Also, GPT-NeoX seems to be compatible with GPT-2, since it uses the same tokenizer and token dictionary. Further research could incorporate GPT-NeoX as an alternative model for use with the Meteor stegosystem. First experiments conducted for this thesis have concluded that text generation with large models, i.e., models with parameters an order of magnitude greater than GPT-2's 345 million parameters, requires a considerable amount of computational power which state-of-the-art laptops cannot deliver without hardware acceleration.

Another promising topic for further research could be to adapt the Meteor stegosystem to use public keys instead of symmetric keys and implement steganographic group communication.

The Meteor stegosystem establishes a very promising approach for using generative neural networks in cryptographically secure steganographic systems. Since it can be expected that computational performance in machine learning will continue to improve in the coming years, steganography on ML models is likely to remain a promising area of research.

Bibliography

Berndt, S. (2017). New Results on Feasibilities And Limitations of Provable Secure Steganography. PhD thesis. Germany. URL: <https://www.zhb.uni-luebeck.de/epubs/ediss2063.pdf>.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., et al. (2022). *GPT-NeoX-20B: An Open-Source Autoregressive Language Model*. DOI: 10.48550/arXiv.2204.06745.

Bochkarev, V., Shevlyakova, A., and Solovyev, V. (Aug. 2012). Average word length dynamics as indicator of cultural changes in society. *Social Evolution and History* 14, 153–175. DOI: 10.48550/arXiv.1208.6109.

Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language Models are Few-Shot Learners. DOI: 10.48550/arXiv.2005.14165.

Dedić, N., Itkis, G., Reyzin, L., and Russell, S. (Apr. 2008). Upper and Lower Bounds on Black-Box Steganography. *Journal of Cryptology* 22, 365–394. DOI: 10.1007/s00145-008-9020-3.

Gage, P. (Feb. 1994). A New Algorithm for Data Compression. *C Users Journal* 12, 23–38. ISSN: 0898-9788. DOI: 10.5555/177910.177914.

Hochreiter, S. and Schmidhuber, J. (Nov. 1997). Long Short-Term Memory. *Neural Computation* 9, 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

Hopper, N., Ahn, L. von, and Langford, J. (2008). Provably Secure Steganography. *IEEE Transactions on Computers* 58, 662–676. DOI: 10.1109/TC.2008.199.

Hopper, N.J. (2004). Toward a Theory of Steganography. AAI3143943. PhD thesis. USA. ISBN: 0496019643. URL: <https://www.cs.cmu.edu/~hopper/thesis.pdf>.

Hopper, N.J., Langford, J., and Ahn, L. von (2002). Provably Secure Steganography. In *Advances in Cryptology — CRYPTO 2002*, (Yung, M., ed.). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 77–92. ISBN: 978-3-540-45708-4. DOI: 10.1007/3-540-45708-9_6.

Hugging Face Inc. (2022). *Hugging Face Transformers Project Website*. URL: <https://huggingface.co/docs/transformers/index> (visited on 09/22/2022).

Bibliography

Kaptchuk, G., Jois, T., Green, M., and Rubin, A. (Aug. 2021a). *Meteor Live Demo*. URL: <https://colab.research.google.com/gist/tusharjois/ec8603b711ff61e09167d8fef37c9b86> (visited on 09/17/2022).

Kaptchuk, G., Jois, T.M., Green, M., and Rubin, A.D. (2021b). Meteor: Cryptographically Secure Steganography for Realistic Distributions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21*. Virtual Event, Republic of Korea: Association for Computing Machinery, pp. 1529–1548. ISBN: 9781450384544. DOI: 10.1145/3460120.3484550.

Kudo, T. and Richardson, J. (Nov. 2018). SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Brussels, Belgium: Association for Computational Linguistics, pp. 66–71. DOI: 10.18653/v1/D18-2012.

LeCun, Y., Bengio, Y., and Hinton, G. (May 2015). Deep Learning. *Nature* 521, 436–44. DOI: 10.1038/nature14539.

Liśkiewicz, M., Reischuk, R., and Wölfel, U. (Sept. 2013). Grey-Box Steganography. *Theoretical Computer Science* 505, 27–41. DOI: 10.1016/j.tcs.2012.03.037.

OpenAI L.L.C. (2022). *OpenAI API Pricing*. URL: <https://web.archive.org/web/20220917214410/https://openai.com/api/pricing/> (visited on 09/17/2022).

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Sennrich, R., Haddow, B., and Birch, A. (Aug. 2016). Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Berlin, Germany: Association for Computational Linguistics, pp. 1715–1725. DOI: 10.18653/v1/P16-1162.

Simmons, G.J. (1984). The Prisoners' Problem and the Subliminal Channel. *Advances in Cryptology: Proceedings of Crypto 83*, (Chaum, D., ed.). Boston, MA: Springer US, pp. 51–67. ISBN: 978-1-4684-4730-9. DOI: 10.1007/978-1-4684-4730-9_5.

Solaiman, I., Brundage, M., Clark, J., Askill, A., Herbert-Voss, A., Wu, J., Radford, A., Krueger, G., Kim, J.W., Kreps, S., et al. (2019). *Release Strategies and the Social Impacts of Language Models*. DOI: 10.48550/arXiv.1908.09203.

Bibliography

The Linux Foundation (2022). *PyTorch Project Website*. URL: <https://pytorch.org/> (visited on 09/22/2022).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., and Polosukhin, I. (2017). *Attention Is All You Need*. DOI: 10.48550/arXiv.1706.03762.

Winter, P. and Lindskog, S. (2012). *How China Is Blocking Tor*. DOI: 10.48550/arXiv.1204.0447.

Zhang, Y., Sun, S., Galley, M., Chen, Y.-C., Brockett, C., Gao, X., Gao, J., Liu, J., and Dolan, B. (July 2020). DIALOGPT : Large-Scale Generative Pre-training for Conversational Response Generation. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, Online: Association for Computational Linguistics, pp. 270–278. DOI: 10.18653/v1/2020.acl-demos.30.