Bad Characters: Imperceptible NLP Attacks

Bad Characters: Imperceptible NLP Attacks

Nicholas Boucher
University of Cambridge
Computer Science & Technology
nicholas.boucher@cl.cam.ac.uk

Ilia Shumailov
University of Cambridge
and Vector Institute
ilia.shumailov@cl.cam.ac.uk

Ross Anderson
University of Cambridge
and University of Edinburgh
ross.anderson@cl.cam.ac.uk

Nicolas Papernot
University of Toronto
and Vector Institute
nicolas.papernot@utoronto.ca

***Abstract*—Several years of research have shown that machine-learning systems are vulnerable to adversarial examples, both in theory and in practice. Until now, such attacks have primarily targeted visual models, exploiting the gap between human and machine perception. Although text-based models have also been attacked with adversarial examples, such attacks struggled to preserve semantic meaning and indistinguishability. In this paper, we explore a large class of adversarial examples that can be used to attack text-based models in a black-box setting without making any human-perceptible visual modification to inputs. We use encoding-specific perturbations that are imperceptible to the human eye to manipulate the outputs of a wide range of Natural Language Processing (NLP) systems from neural machine-translation pipelines to web search engines. We find that with a single imperceptible encoding injection – representing one invisible character, homoglyph, reordering, or deletion – an attacker can significantly reduce the performance of vulnerable models, and with three injections most models can be functionally broken. Our attacks work against currently-deployed commercial systems, including those produced by Microsoft and Google, in addition to open source models published by Facebook, IBM, and HuggingFace. This novel series of attacks presents a significant threat to many language processing systems: an attacker can affect systems in a targeted manner without any assumptions about the underlying model. We conclude that text-based NLP systems require careful input sanitization, just like conventional applications, and that given such systems are now being deployed rapidly at scale, the urgent attention of architects and operators is required.**

***Index Terms*—adversarial machine learning, NLP, text-based models, text encodings, search engines**

I. Introduction

Do x and x look the same to you? They may look identical to humans, but not to most natural-language processing systems. How many characters are in the string "123"? If you guessed 100, you're correct. The first example contains the Latin character x and the Cyrillic character h, which are typically rendered the same way. The second example contains 97 zero-width non-joiners[1] following the visible characters. Indeed, the title of this paper contains 1000 invisible characters imperceptible to human users.

Several years of research have demonstrated that machine-learning systems are vulnerable to adversarial examples, both theoretically and in practice [**?**]. Such attacks initially targeted visual models used in image classification [**?**], though there has been recent interest in natural language processing and other applications. We present a broad class of powerful adversarial-example attacks on text-based models. These attacks apply input perturbations using invisible characters, control characters, and homoglyphs – distinct characters with similar glyphs. These perturbations are imperceptible to human users, but the bytes used to encode them can change the output drastically.

We have found that machine-learning models that process user-supplied text, such as neural machine-translation systems, are particularly vulnerable to this style of attack. Consider, for example, the market-leading service Google Translate [**?**]. At the time of writing, entering the string "paypal" in the English to Russian model correctly outputs "PayPal", but replacing the Latin character a in the input with the Cyrillic character а incorrectly outputs "папа" ("father" in English). Model pipelines are agnostic of characters outside of their dictionary and replace them with `<unk>` tokens; the software that calls them may however propagate unknown words from input to output. While that may help with general understanding of text, it opens a surprisingly large attack surface.

Simple text-encoding attacks have been used occasionally in the past to get messages through spam filters. For example, there was a brief discussion in the SpamAssassin project in 2018 about how to deal with zero-width characters, which had been found in some sextortion scams [**?**]. Although such tricks were known to engineers designing spam filters, they were not a primary concern. However, the rapid deployment of NLP systems in a large range of applications, from machine translation [**?**] through copyright enforcement [**?**] to hate speech filtering [**?**], is suddenly creating a host of high-value targets that have capable motivated opponents.

[1]Unicode character U+200C

TABLE I
IMPERCEPTIBLE PERTURBATIONS IN VARIOUS NLP TASKS

Input Rendering	Input Encoding	Task	Output
Send money to account 1234	Send money to account U+202E4321	Translation (EN→FR)	Envoyer de l'argent au compte 4321 *(Send money to account 4321)*
You are a coward and a fool.	You akU+8re aqU+8 AU+8coward and a fovU+8JU+8ol.	Toxic Content Detection	8.2% toxic *(96.8% toxic unperturbed)*
Oh, what a fool I feel! / I am beyond proud.	Oh, what a U+200BfoU+200Bol IU+200B U+200BU+200Bfeel! / I am beyond proud.	Natural Language Inference	0.3% contradiction *(99.8% contradiction unperturbed)*

The main contribution of this work is to explore and develop a class of imperceptible encoding-based attacks and to study their effect on the NLP systems that are now being deployed everywhere at scale. Our experiments show that many developers of such systems have been heedless of the risks; this is surprising given the long history of attacks on many varieties of systems that have exploited unsanitized inputs. We provide a set of examples of imperceptible attacks across various NLP tasks in Table I. As we will later describe, these attacks take the form of invisible characters, homoglyphs, reorderings, and deletions injected via a genetic algorithm that maximizes a loss function defined for each NLP task.

Our findings present an attack vector that must be considered when designing any system processing natural language that may ingest text-based inputs with modern encodings, whether directly from an API or via document parsing. We then explore a series of defenses that can give some protection against this powerful set of attacks, such as discarding certain characters prior to tokenization, applying character mappings, and leveraging rendering and OCR for pre-processing. Defense is not entirely straightforward, though, as application requirements and resource constraints may prevent the use of specific defenses in certain circumstances.

This paper makes the following contributions:

- We present a novel class of imperceptible perturbations for NLP models;
- We present four black-box variants of imperceptible attacks against both the integrity and availability of NLP models;
- We show that our imperceptible attacks degrade performance against task-appropriate benchmarks for eight models implementing machine translation, toxic content detection, textual entailment classification, named entity recognition, and sentiment analysis to near zero in untargeted attacks, succeed in most targeted attacks, and slow inference down by at least a factor of two in sponge example attacks;
- We evaluate our attacks extensively against both open source models and Machine Learning as a Service (MLaaS) offerings provided by Facebook, IBM, Microsoft, Google, and HuggingFace, finding that *all* tested systems were vulnerable to three attack variants, and most were vulnerable to four;
- We present defenses against these attacks, and discuss why defense can be complex.

II. MOTIVATION

Researchers have already experimented with adversarial attacks on NLP models [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. However, up until now, such attacks were noticeable to human inspection and could be identified with relative ease. If the attacker inserts single-character spelling mistakes [**?**], [**?**], [**?**], [**?**], they look out of place, while paraphrasing [**?**] often changes the meaning of a text enough to be noticeable. The attacks we discuss in this paper are the first class of attacks against modern NLP models that are imperceptible and do not distort semantic meaning.

Our attacks can cause significant harm in practice. Consider two examples. First, consider a nation-state whose primary language is not spoken by the staff at a large social media company performing content moderation – already a well-documented challenge [**?**]. If the government of this state wanted to make it difficult for moderators to block a campaign to incite violence against minorities, it could use imperceptible perturbations to stifle the efficacy of both machine-translation and toxic-content detection of inflammatory sentences.

Second, the ability to hide text in plain sight, by making it easy for humans to read but hard for machines to process, could be used by many bad actors to evade platform content filtering mechanisms and even impede law-enforcement and intelligence agencies. The same perturbations even prevent proper search-engine indexing, making malicious content hard to locate in the first place. We found that production search engines do not parse invisible characters and can be maliciously targeted with well-crafted queries. At the time of initial writing, Googling "The meaning of life" returned approximately 990 million results. Prior to responsible disclosure, searching for the visually identical string containing 250 invisible "zero width joiner" characters[2] returned exactly none.

III. RELATED WORK

A. Adversarial Examples

Machine-learning techniques are vulnerable to many large classes of attack [**?**], with one major class being adversarial examples. These are inputs to models which, during inference, cause the model to output an incorrect result [**?**]. In a white-box environment – where the adversary knows the model – such examples can be found using a number of gradient-based methods which typically aim to maximize the loss

[2]Unicode character U+200D

TABLE II
TAXONOMY OF ADVERSARIAL NLP ATTACKS IN ACADEMIC LITERATURE.

Attack	Features			Integrity		Availability
	Imperceptible	Semantic Similarity	Blackbox	Classification	Translation	DoS
RNN Adversarial Sequences [?]				✓		
Synthetic and Natural Noise [?]			✓		✓	
DeepWordBug [?]			✓	✓		
HotFlip [?]				✓		
Syntactically Controlled Paraphrase [?]		✓	✓	✓		
Natural Adversarial Examples [?]			✓	✓	✓	
Natural Language Adversarial Examples [?], [?]		✓	✓	✓		
TextBugger [?]			✓	✓		
seq2seq Adversarial Perturbations [?]		✓			✓	
Probability Weighted Word Saliency [?]		✓		✓		
Sponge Examples [?]			✓			✓
Reinforced Generation [?]		✓	✓		✓	
Imperceptible Perturbations	✓	✓	✓	✓	✓	✓

function under a series of constraints [?], [?], [?]. In the black-box setting, where the model is unknown, the adversary can transfer adversarial examples from another model [?], or approximate gradients by observing output labels and, in some settings, confidence [?].

Training data can also be poisoned to manipulate the accuracy of the model for specific inputs [?], [?]. Bitwise errors can be introduced during inference to reduce the model's performance [?]. Inputs can also be chosen to maximize the time or energy a model takes during inference [?], or to expose confidential training data via inference techniques [?]. In other words, adversarial algorithms can affect the *integrity*, *availability* and *confidentiality* of machine-learning systems [?], [?], [?].

B. NLP Models

Natural language processing (NLP) systems are designed to process human language. Machine translation was proposed as early as 1949 [?] and has become a key sub-field of NLP. Early approaches to machine translation tended to be rule-based, using expert knowledge from human linguists, but statistical methods became more prominent as the field matured [?], eventually yielding to neural networks [?], then recurrent neural networks (RNNs) because of their ability to reference past context [?]. The current state of the art is the Transformer model, which provides the benefits of RNNs and CNNs in a traditional network via the use of an attention mechanism [?].

Transformers are a form of encoder-decoder model [?], [?] that map sequences to sequences. Each source language has an encoder that converts the input into a learned interlingua, an intermediate representation which is then decoded into the target language using a model associated with that language.

Regardless of the details of the model used for translation, natural language must be encoded in a manner that can be used as its input. The simplest encoding is a dictionary that maps words to numerical representations, but this fails to encode previously unseen words and thus suffers from limited vocabulary. N-gram encodings can increase performance, but increase the dictionary size exponentially while failing to solve the unseen-word problem. A common strategy is to decompose words into sub-word segments prior to encoding, as this enables the encoding and translation of previously unseen words in many circumstances [?].

C. Adversarial NLP

Early adversarial ML research focused on image classification [?], [?], and the search for adversarial examples in NLP systems began later, targeting sequence models [?]. Adversarial examples are inherently harder to craft due to the discrete nature of natural language. Unlike images in which pixel values can be adjusted in a near-continuous and virtually imperceptible fashion to maximize loss functions, perturbations to natural language are more visible and involve the manipulation of more discrete tokens.

More generally, source language perturbations that will provide effective adversarial samples against human users need to account for semantic similarity [?]. Researchers have proposed using word-based input swaps with synonyms [?] or character-based swaps with semantic constraints [?]. These methods aim to constrain the perturbations to a set of transformations that a human is less likely to notice. Both neural machine-translation [?] and text classification [?], [?] models generally perform poorly on noisy inputs such as misspellings, but such perturbations create clear visual artifacts that are easier for humans to notice.

Using different paraphrases of the same meaning, rather than one-to-one synonyms, may give more leeway. Paraphrase sets can be generated by comparing machine back-translations of large corpora of text [?], and used to systematically generate adversarial examples for machine-translation systems [?]. One can also search for neighbors of the input sentence in an embedded space [?]; these examples often result in low-performance translations, making them candidates for adversarial examples. BLEU score is commonly used for assessing the quality of machine translations [?], and therefore also for assessing related attacks. Although paraphrasing can indeed help preserve semantics, humans often notice that the results look odd. Our attacks on the other hand do not introduce any visible perturbations, use fewer substitutions, and preserve semantic meaning perfectly.

Genetic algorithms have been used to find adversarial perturbations against inputs to sentiment analysis systems, presenting an attack viable in the black-box setting without access to gradients [**?**]. Reinforcement learning can be used to efficiently generate adversarial examples for translation models [**?**]. There have even been efforts to combine academic NLP adversarial techniques into easily consumable toolkits available online [**?**], making these attacks relatively easy to use. Unlike the techniques described in this paper, though, all existing NLP adversarial example techniques result in human-perceptible visual artifacts within inputs.

Michel et al. also propose that unknown tokens `<unk>`, which are used to encode text sequences not recognized by the natural language encoder in NLP settings, can be leveraged to make compelling source language perturbations due to the flexibility of the characters which encode to `<unk>` [**?**]. However, all methods proposed so far for generating `<unk>` use visible characters.

We present a taxonomy of adversarial NLP attacks in Table II.

D. Unicode

Unicode is a character set designed to standardize the electronic representation of text [**?**]. As of the time of writing, it can represent 143,859 characters across many different languages and symbol groups. Characters as diverse as Latin letters, traditional Chinese characters, mathematical notation, and emojis can all be represented in Unicode. It maps each character to a code point, or numerical representation.

These numerical code points, often denoted with the prefix `U+`, can be encoded in a variety of ways, although UTF-8 is the most common. This is a variable-length encoding scheme that represents code points as 1-4 bytes.

A font is a collection of glyphs that describe how code points should be rendered. Most computers support many different fonts. It is not required that fonts have a glyph for every code point, and code points without corresponding glyphs are typically rendered as an 'unknown' placeholder character.

E. Unicode Security

As it has to support a globally broad set of languages, the Unicode specification is quite complex. This complexity can lead to security issues, as detailed in the Unicode Consortium's technical report on Unicode security considerations [**?**].

One primary security consideration in the Unicode specification is the multitude of ways to encode homoglyphs, which are unique characters that share the same or nearly the same glyph. This problem is not unique to Unicode; for example, in the ASCII range, the rendering of the lowercase Latin 'l'[3] is often nearly identical to the uppercase Latin 'I'[4]. In some fonts, character sequences can act as pseudo-homoglyphs, such as the sequences 'rn' and 'm' in most sans serif fonts.

Such visual tricks provide a tool in the arsenal of cyber scammers [**?**]. The earliest example we found is that of *paypaI.com* (notice the last domain name character is an uppercase 'I'), which was used in July 2000 to trick users into disclosing passwords for *paypal.com* [**?**]. Indeed, significant attention has since been given to homoglyphs in URLs [**?**], [**?**], [**?**], [**?**]. Some browsers attempt to remedy this ambiguity by rendering all URL characters in their lowercase form upon navigation, and the IETF set a standard to resolve ambiguities between non-ASCII characters that are homoglyphs with ASCII characters. This standard, called Punycode, resolves non-ASCII URLs to an encoding restricted to the ASCII range. For example, most browsers will automatically re-render the URL *pаypаl.com* (which uses the Cyrillic а[5]) to its Punycode equivalent *xn–pypl-53dc.com* to highlight a potentially dangerous ambiguity. However, Punycode can introduce new opportunities for deception. For example, the URL *xn–google.com* decodes to four semantically meaningless traditional Chinese characters. Furthermore, Punycode does not solve cross-script homoglyph encoding vulnerabilities outside of URLs. For example, homoglyphs have in the past caused security vulnerabilities in various non-URL areas such as certificate common names.

Homoglyphs have also been proposed for information hiding, such as encoding information via sequences of different whitespace characters [**?**].

Unicode attacks can also exploit character ordering. Some character sets (such as Hebrew and Arabic) naturally display in right-to-left order. The possibility of intermixing left-to-right and right-to-left text, as when an English phrase is quoted in an Arabic newspaper, necessitates a system for managing character order with mixed character sets. For Unicode, this is the Bidirectional (Bidi) Algorithm [**?**]. Unicode specifies a variety of control characters that allow a document creator to fine-tune character ordering, including Bidi override characters that allow complete control over display order. The net effect is that an adversary can force characters to render in a different order than they are encoded, thus permitting the same visual rendering to be represented by a variety of different encoded sequences. Historically, Bidi overrides have been used by scammers to change the appearance of file extensions, thus enabling stealthy dissemination of malware [**?**].

Lastly, an entire class of vulnerabilities stems from bugs in Unicode implementations. These have historically been used to generate a range of interesting exploits about which it is difficult to generalize. While the Unicode Consortium does publish a set of software components for Unicode support [**?**], many operating systems, platforms, and other software ecosystems have different implementations. For example, GNOME produces Pango [**?**], Apple produces Core Text [**?**], while Microsoft produces a Unicode implementation for Windows [**?**].

In what follows, we will mostly disregard bugs and focus on attacks that exploit correct implementations of the Unicode

[3]ASCII value `0x6C`

[4]ASCII value `0x49`

[5]Unicode character U+0430

standard. We instead exploit the gap between visualization and NLP pipelines.

IV. Background

A. Attack Taxonomy

In this paper, we explore the class of imperceptible attacks based on Unicode and other encoding conventions which are generally applicable to text-based NLP models. We see each attack as a form of adversarial example whereby imperceptible perturbations are applied to fixed inputs into existing text-based NLP models.

We define these *imperceptible perturbations* as modifications to the encoding of a string of text which result in either:

- No visual modification to the string's rendering by a standards-compliant rendering engine compared to the unperturbed input, or
- Visual modifications sufficiently subtle to go unnoticed by the average human reader using common fonts.

For the latter case, it is alternatively possible to replace human imperceptibility as indistinguishability by a computer vision model between images of the renderings of two strings, or a maximum pixel-wise difference between such rendering.

We consider four different classes of imperceptible attack against NLP models:

1) **Invisible Characters**: Valid characters which by design do not render to a visible glyph are used to perturb the input to a model.
2) **Homoglyphs**: Unique characters which render to the same or visually similar glyphs are used to perturb the input to a model.
3) **Reorderings**: Directionality control characters are used to override the default rendering order of glyphs, allowing reordering of the encoded bytes used as input to a model.
4) **Deletions**: Deletion control characters, such as the backspace, are injected into a string to remove injected characters from its visual rendering to perturb the input to a model.

These imperceptible text-based attacks on NLP models represent an abstract class of attacks independent of different text-encoding standards and implementations. For the purpose of concrete examples and experimental results, we will assume the near-ubiquitous Unicode encoding standard, but we believe our results to be generalizable to any encoding standard with a sufficiently large character and control-sequence set.

Further classes of text-based attacks exist, as detailed in Table I, but all other attack classes produce visual artifacts.

The imperceptible text-based attacks described in this paper can be used against a broad range of NLP models. As we explain later, imperceptible perturbations can manipulate machine translation, break toxic content classifiers, degrade search engine querying and indexing, and generate sponge examples [**?**] for denial-of-service (DoS) attacks, among other possibilities.

B. NLP Pipeline

Modern NLP pipelines have evolved through decades of research to include a large number of performance optimizations. Text-based inputs undergo a number of pre-processing steps before model inference. Typically a *tokenizer* is first applied to separate words and punctuation in a task-meaningful way, an example being the Moses tokenizer [**?**] used in the Fairseq models evaluated later in this paper. Tokenized words are then encoded. Early models used dictionaries to map tokens to encoded embeddings, and tokens not seen during training were replaced with a special `<unk>` embedding. Many modern models now apply Byte Pair Encoding (BPE) or the WordPiece algorithm [**?**] before dictionary lookups. BPE, a common data compression technique, and WordPiece both identify common subwords in tokens. This often results in increased performance, as it allows the model to capture additional knowledge about language semantics from morphemes [**?**]. Both of these pre-processing methodologies are commonly used in deployed NLP models, including all five open source models published by Facebook, IBM, and HuggingFace evaluated in this paper.

Modern NLP pipelines process text in a very different manner than text-rendering systems, even when dealing with the same input. While the NLP system is dealing with the semantics of human language, the rendering engine is dealing with a large, rich set of different control characters. This structural difference between what models see and what humans see is what we exploit in our attacks.

C. Attack Methodology

We approach the generation of adversarial samples as an optimization problem. Assume an NLP function $f(\mathbf{x}) = \mathbf{y} : X \to Y$ mapping textual input $\mathbf{x}$ to $\mathbf{y}$. Depending on the task, Y is either a sequence of characters, words, or hot-encoded categories. For example, translation tasks such as WMT assume Y to be a sequence of characters, whereas categorization tasks such as MNLI assume Y to be one of three categories. Furthermore, we assume a strong black-box threat model where adversaries have access to model output but cannot observe the internals. This makes our attack realistic: we later show it can be mounted on existing commercial ML services. In this threat model, an adversary's goal is to imperceptibly manipulate f using a perturbation function p.

These manipulations fall into two categories:

- **Integrity Attack**: The adversary aims to find p such that $f(p(\mathbf{x})) \neq f(\mathbf{x})$. For a targeted attack, the adversary further constrains p such that the perturbed output matches a fixed target $\mathbf{t}$: $f(p(\mathbf{x})) = \mathbf{t}$.
- **Availability Attack**: The adversary aims to find p such that $time(f(p(\mathbf{x}))) > time(f(\mathbf{x}))$, where *time* measures the inference runtime of f.

We also define a constraint on the perturbation function p:

- **Budget**: A budget b such that $dist(\mathbf{x}, p(\mathbf{x})) \leq b$. The function *dist* may refer to any distance metric.

We define the attack as optimizing a set of operations over the input text, where each operation corresponds to

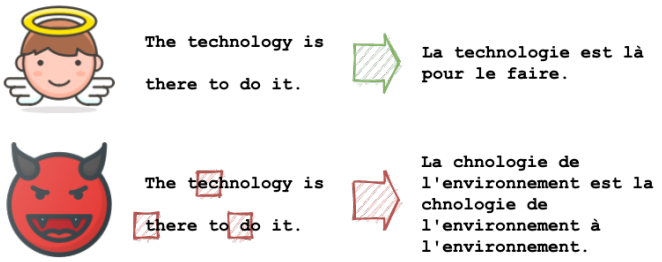

Fig. 1. Attack using invisible characters. Example machine translation input is on the left with model output on the right. Invisible characters are denoted by red boxes, such as between the 'e' and 'c'.

the injection of one short sequence of Unicode characters to perform a single imperceptible perturbation of the chosen class. The length of the injected sequence is dependent upon the class chosen and attack implementation; in our evaluation we use one character injections for invisible characters and homoglyphs, two characters for deletions, and ten characters for reorderings, as later described. We select a gradient-free optimization method – differential evolution [**?**] – to enable this attack to work in the black-box setting without having to recover approximated gradients. This approach randomly initializes a set of candidates and evolves them over many iterations, ultimately selecting the best-performing traits.

The attack algorithm is shown in Algorithm 1. It takes as parameters input text x and attack $\mathcal{A}$, representing either an invisible character, homoglyph, reordering, or deletion attack. $\mathcal{A}$ is a function which applies its attack according to the parameters passed to it encoding the location and degree of the perturbation, bounded by $\mathcal{B}_\mathcal{A}$ according to budget β. It also takes a model $\mathcal{T}$ implementing an NLP task, and optionally a target output y if performing a targeted attack. Finally, it expects parameters representing a population size s, number of evolutions m, differential weight F, and crossover probability CR, which are all standard parameters of differential evolution optimization [**?**]. In summary, the attack algorithm defines an objective function $\mathcal{F}(\cdot)$, which seeks to either maximize perturbed model output Levenshtein distance from its unperturbed output, minimize model output Levenshtein distance to a target value, or maximize model inference time. This objective function is then optimized using differential evolution, a common gradient-free genetic optimization. Finally, the perturbed text optimizing the objective function $\mathcal{F}(\cdot)$ is returned.

D. Invisible Characters

Invisible characters are encoded characters that render to the absence of a glyph and take up no space in the resulting rendering. Invisible characters are typically not font-specific, but follow from the specification of an encoding format. An example in Unicode is the zero-width space character[6] (ZWSP). An example of an attack using invisible characters is shown in Figure 1.

[6]Unicode character U+200B

Algorithm 1: Imperceptible perturbations adversarial example via differential evolution.

Input: text $\mathbf{x}$, attack $\mathcal{A}$ with input bounds distribution $\mathcal{B}_\mathcal{A}$, NLP task $\mathcal{T}$, target $\mathbf{y}$, perturbation budget β, population size s, evolution iterations m, differential weight $F \in [0, 2]$, crossover probability $CR \in [0, 1]$
Result: Adversarial example visually identical to $\mathbf{x}$ against task $\mathcal{T}$ using attack $\mathcal{A}$

Randomly initialize population $\mathbf{P} := \{\mathbf{p_0}, \ldots, \mathbf{p_s}\}$, where $\mathbf{p_n} \sim \mathcal{B}_\mathcal{A}(\mathbf{x})$
if availability attack **then**
  $\mathcal{F}(\cdot) = \text{execution\_time}(\mathcal{T}(\mathcal{A}(\mathbf{x}, \cdot)))$
else if integrity attack **then**
  if targeted attack **then**
    $\mathcal{F}(\cdot) = \text{levenshtein\_distance}(\mathbf{y}, \mathcal{T}(\mathcal{A}(\mathbf{x}, \cdot)))$
  else
    $\mathcal{F}(\cdot) = \text{levenshtein\_distance}(\mathcal{T}(\mathbf{x}), \mathcal{T}(\mathcal{A}(\mathbf{x}, \cdot)))$
  end if
end if
for $i := 0$ **to** m **do** ▷ $\mathcal{U}$ is uniform dist.
  for $j := 0$ **to** s **do**
    $\mathbf{p_a}, \mathbf{p_b}, \mathbf{p_c} \xleftarrow{\text{rand}} \mathbf{P}$ s.t. $j \neq a \neq b \neq c$
    $R \sim \mathcal{U}(0, |\mathbf{p_j}|)$
    $\hat{\mathbf{p}}_\mathbf{j} := \mathbf{p_j}$
    for $k := 0$ **to** $|\mathbf{p_j}|$ **do**
      $r_j \sim \mathcal{U}(0, 1)$
      if $r_j < CR$ **or** $R = k$ **then**
        $\hat{\mathbf{p}}_{\mathbf{j_k}} = \mathbf{p_{a_k}} + F \times (\mathbf{p_{b_k}} - \mathbf{p_{b_k}})$
      end if
    end for
    if $\mathcal{F}(\hat{\mathbf{p}}_\mathbf{j}) \geq \mathcal{F}(\mathbf{p_j})$ **then**
      $\mathbf{p_j} = \hat{\mathbf{p}}_\mathbf{j}$
    end if
  end for
end for
$\bar{\mathbf{f}} := \{\mathcal{F}(\mathbf{p_0}), \ldots, \mathcal{F}(\mathbf{p_s})\}$
return $\mathcal{A}(\mathbf{x}, \mathbf{p}_{\text{argmax}(\bar{\mathbf{f}})})$

It is important to note that characters lacking a glyph definition in a specific font are not typically treated as invisible characters. Due to the number of characters in Unicode and other large specifications, fonts will often omit glyph definitions for rare characters. For example, Unicode supports characters from the ancient Mycenaean script Linear B, but these glyph definitions are unlikely to appear in fonts targeting modern languages such as English. However, most text-rendering systems reserve a special character, often □ or 🞏, for valid Unicode encodings with no corresponding glyph. These characters are therefore visible in rendered text.

In practice, though, invisible characters are font-specific. Even though some characters are designed to have a non-glyph rendering, the details are up to the font designer. They might, for example, render all traditionally invisible characters

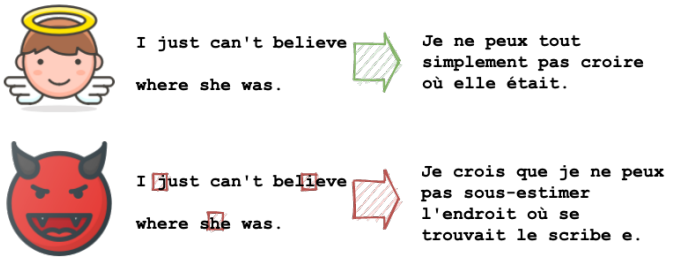

Fig. 2. Attack using homoglyphs. Example machine translation input is on the left with model output on the right. Homoglyphs are highlighted with red boxes, where *j* is replaced with U+3F3, *i* with U+456 and *h* with U+4BB.

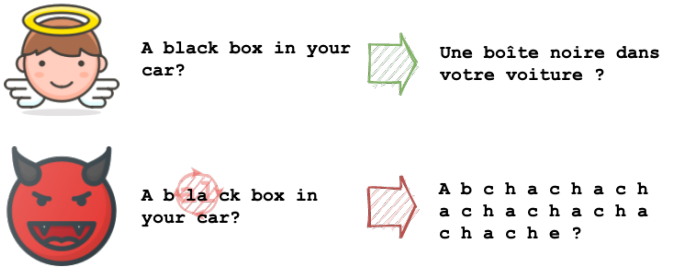

Fig. 3. Attack using reorderings. Example machine translation input is on the left with model output on the right. The red circle denotes the string is encoded in reverse order surrounded by Bidi override characters.

by printing the corresponding Unicode code point as a base 10 numeral. Yet a small number of fonts dominate the modern world of computing, and fonts in common use are likely to respect the spirit of the Unicode specification. For the purposes of this paper, we will determine character visibility using GNU's Unifont [**?**] glyphs. Unifont was chosen because of its relatively robust coverage of the current Unicode standard, its distribution with common operating systems, and its visual similarity to other common fonts.

Although invisible characters do not produce a rendered glyph, they nevertheless represent a valid encoded character. Text-based NLP models operate over encoded bytes as inputs, so these characters will be "seen" by a text-based model even if they are not rendered to anything perceptible by a human user. We found that these bytes alter model output. When injected arbitrarily into a model's input, they typically degrade the performance both in terms of accuracy and runtime. When injected in a targeted fashion, they can be used to modify the output in a desired way, and may coherently change the meaning of the output across many NLP tasks.

E. Homoglyphs

Homoglyphs are characters that render to the same glyph or to a visually similar glyph. This often occurs when portions of the same written script are used across different language families. For example, consider the Latin letter 'A' used in English. The very similar character 'A' is used in the Cyrillic alphabet. Within the Unicode specification these are distinct characters, although they are typically rendered as homoglyphs.

An example of an attack using homoglyphs is shown in Figure 2. Like invisible characters, homoglyphs are font-specific. Even if the underlying linguistic system denotes two characters in the same way, fonts are not required to respect this. That said, there are well-known homoglyphs in the most common fonts used in everyday computing.

The Unicode Consortium publishes two supporting documents with the Unicode Security Mechanisms technical report [**?**] to draw attention to similarly rendered characters. The first defines a mapping of characters that are intended to be homoglyphs within the Unicode specification and should therefore map to the same glyph in font implementations [**?**]. The second document [**?**] defines a set of characters that are likely to be visually confused, even if they are not rendered with precisely the same glyph.

For the experiments in this paper, we use the Unicode technical reports to define homoglpyh mappings. We also note that homoglyphs, particularly for specific less common fonts, can be identified using an unsupervised clustering algorithm against vectors representing rendered glyphs. To illustrate this, we used a VGG16 convolution neural network [**?**] to transform all glyphs in the Unifont font into vectorized embeddings and performed various clustering operations. Figure 4 visualizes mappings provided by the Unicode technical reports as a dimensionality-reduced character cluster plot. We find that the results of well-tuned unsupervised clustering algorithms produce similar results, but have chosen to use the official Unicode mappings in this paper for reproducibility.

F. Reorderings

The Unicode specification supports characters from languages that read in both the left-to-right and right-to-left directions. This becomes nontrivial to manage when such scripts are mixed. The Unicode specification defines the Bidirectional (Bidi) Algorithm [**?**] to support standard rendering behavior for mixed-script documents. However, the specification also

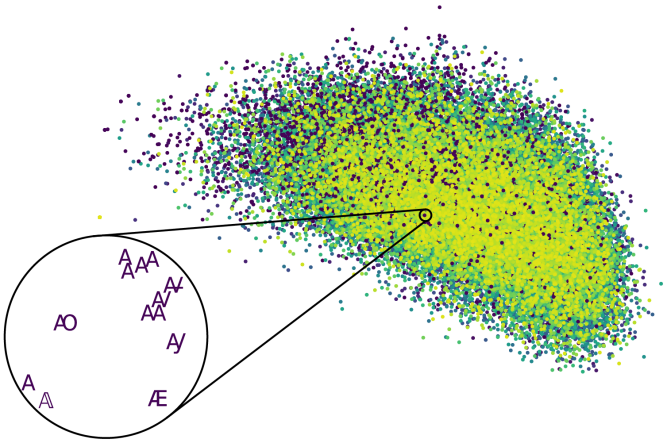

Fig. 4. Clustering of Unicode homoglyphs according to the Unicode Security Confusables document, plotted as a 2D PCA of Unifont glyph images via a VGG16 model.

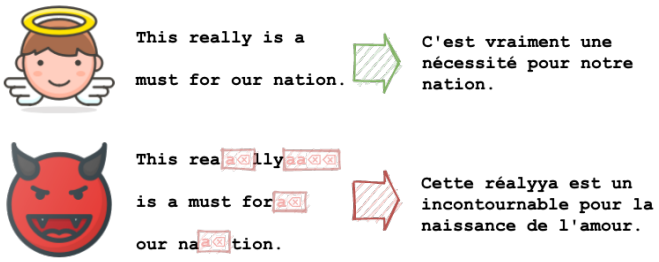

Fig. 5. Attack using deletions. Example machine translation input is on the left with model output on the right. The red boxes highlight injected characters followed by backspace characters.

allows the Bidi Algorithm to be overridden using invisible direction-override control characters, which allow near-arbitrary rendering for a fixed encoded ordering.

An example of an attack using reorderings is shown in Figure 3. In an adversarial setting, Bidi control characters allow the encoded ordering of characters to be shuffled without affecting character rendering thus making them a form of imperceptible perturbation.

Unlike invisible character and homoglyph attacks, the class of reordering attacks is font-independent and relies only on the implementation of the Unicode Bidi Algorithm. Bidi algorithm implementations sometimes differ in how they handle specific control sequences, meaning that some attacks may be platform or application specific in practice, but most mature Unicode rendering systems behave similarly. Appendix Algorithm 2 defines an algorithm for generating 2^{n-1} unique reorderings for strings of length n using nested Bidi control characters. At the time of writing, it has been tested to work against the Unicode implementation in Chromium [**?**].

G. Deletions

A small number of control characters in Unicode can cause neighbouring text to be removed. The simplest examples are the backspace (BS) and delete (DEL) characters. There is also the carriage return (CR) which causes the text-rendering algorithm to return to the beginning of the line and overwrite its contents. For example, encoded text which represents "Hello **CR**Goodbye World" will be rendered as "Goodbye World".

An example of an attack using deletions is shown in Figure 5. Deletion attacks are font-independent, as Unicode does not allow glyph specification for the basic control characters inherited from ASCII including BS, DEL, and CR. In general, deletion attacks are also platform independent as there is not significant variance in Unicode deletion implementations. However, these attacks can be harder to exploit in practice because most systems do not copy deleted text to the clipboard. As such, an attack using deletion perturbations generally requires an adversary to submit encoded Unicode bytes directly into a model, rather than relying on a victim's copy+paste functionality.

V. NLP Attacks

A. Integrity Attack

Regardless of the tokenizer or dictionary used in an NLP model, systems are unlikely to handle imperceptible perturbations gracefully in the absence of specific defenses. Integrity attacks against NLP models exploit this fact to achieve degraded model performance in either a targeted or untargeted fashion.

The specific affect on input embedding transformation depends on the class of perturbation used:

- **Invisible characters** (between words): Invisible characters are transformed into `<unk>` embeddings between properly-embedded adjacent words.
- **Invisible characters** (within words): In addition to being transformed into `<unk>` embeddings, the invisible characters may cause the word in which it is contained to be embedded as multiple shorter words, interfering with the standard processing.
- **Homoglyphs**: If the token containing the homoglyph is present in the model's dictionary, a word that contains it will be embedded with the less-common, and likely lower-performing, vector created from such data. If the homoglyph is not known, the token will be embedded as `<unk>`.
- **Reorderings**: In addition to the Bidi control characters each being treated as invisible characters, the other characters input into the model will be in the underlying encoded order rather than the rendered order.
- **Deletions**: In addition to deletion-control characters each being treated as an invisible character, the deleted characters encoded into the input are still validly processed by the model.

Each of these modifications to embedded inputs degrades a model's performance. The cause is model-specific, but for attention-based models we expect that tokens in a context of `<unk>` tokens are treated differently.

B. Availability Attack

Machine-learning systems can be attacked by workloads that are unusually slow. The inputs generating such computations are known as sponge examples [**?**].

In this paper we show that sponge examples can be constructed in a targeted way, both with fixed and increased input size. For a fixed-size sponge example, an attacker can replace individual characters with homoglyphs that take longer to process. If an increase in input size is tolerable, the attacker can also inject invisible characters, forcing the model to take additional time to process these additional steps in its input sequence.

Such attacks may be carried out more covertly if the visual appearance of the input does not arouse users' suspicions. If launched in parallel at scale, the availability of hosted NLP models may be degraded, suggesting that a distributed denial-of-service attack may be feasible on text-processing services.

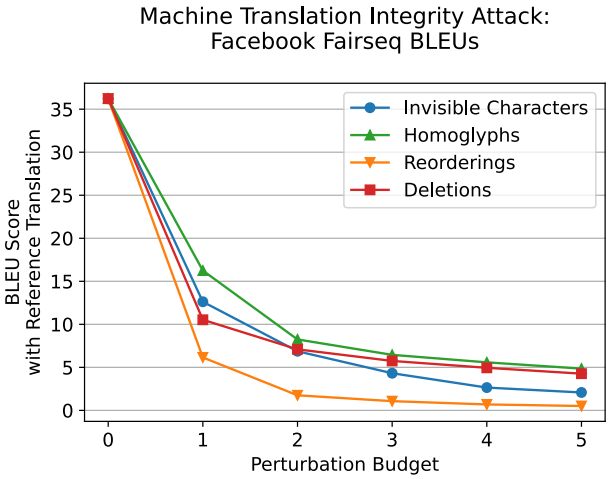

Fig. 6. BLEU scores of imperceptible perturbations vs. unperturbed WMT data on Fairseq EN-FR model

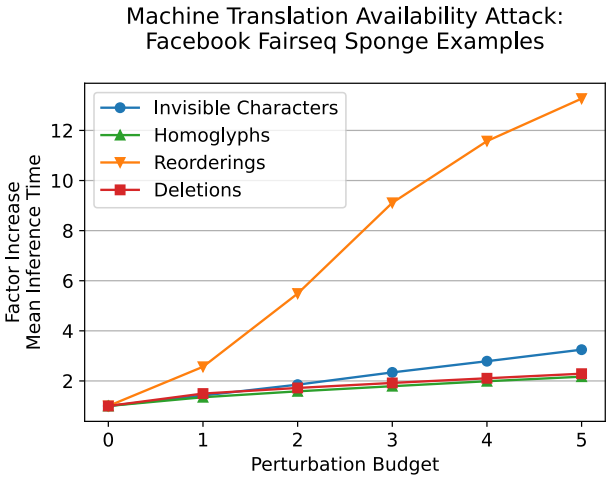

Fig. 7. Fairseq sponge example average inference time

VI. Evaluation

A. Experiment Setup

We evaluate the performance of each class of imperceptible perturbation attack – invisible characters, homoglyphs, reorderings, and deletions – against five NLP tasks: machine translation, toxic content detection, textual entailment classification, named entity recognition, and sentiment analysis. We perform these evaluations against a collection of five open-source models and three closed-source, commercial models published by Google, Facebook, Microsoft, IBM, and HuggingFace. We repeat each experiment with perturbation budget values varying from zero to five.

All experiments were performed in a black-box setting in which unlimited model evaluations are permitted, but accessing the assessed model's weights or state is not permitted. This represents one of the strongest threat models for which attacks are possible in nearly all settings, including against commercial Machine-Learning-as-a-Service (MLaaS) offerings. Every model examined was vulnerable to imperceptible perturbation attacks. We believe that the applicability of these attacks should in theory generalize to any text-based NLP model without adequate defenses in place.

We perform a collection of untargeted, targeted, and sponge example attacks across the eight models. The experiments were performed on a cluster of machines each equipped with a Tesla P100 GPU and Intel Xeon Silver 4110 CPU running Ubuntu.

For each class of perturbation, we followed Algorithm 1 and found that the optimization converged quickly, thus choosing a population size of 32 with a maximum of 10 iterations in the genetic algorithm. Increasing these parameters further would likely allow an attacker to find even more effective perturbations; i.e. our experimental results obtain a lower bound.

For the objective functions used in these experiments, invisible characters were chosen from a set including ZWSP, ZWNJ, and ZWJ[7]; homoglyphs sets were chosen according to the relevant Unicode technical report [**?**]; reorderings were chosen from the sets defined using Algorithm 2; and deletions were chosen from the set of all non-control ASCII characters followed by a BKSP[8] character. We define the unit value of the perturbation budget as one injected invisible character, one homoglyph character replacement, one `Swap` sequence according to the reordering algorithm, or one ASCII-backspace deletion pair.

We have published a command-line tool written in Python to conduct these experiments as well as the entire set of adversarial examples resulting from these experiments.[9] We have also published an online tool for validating whether text may contain imperceptible perturbations and for generating random imperceptible perturbations.[10]

In the following sections, we describe each experiment in detail.

B. Machine Translation: Integrity

For the machine translation task, we used an English-French transformer model pre-trained on WMT14 data [**?**] published by Facebook as part of Fairseq [**?**], Facebook AI Research's open source ML toolkit for sequence modeling. We utilized the corresponding WMT14 test set data to provide reference translations for each adversarial example.

For the set of integrity attacks, we crafted adversarial examples for 500 sentences and repeated adversarial generation for perturbations budgets of 0 through 5. Each example took, on average, 432 seconds to generate.

For the adversarial examples generated, we compare the BLEU [**?**] scores of the resulting translation against the reference translation in Figure 6. We also provide the Levenshtein distances between these values in Appendix Figure 14, which

[7]Unicode characters U+200B, U+200C, U+200D
[8]Unicode character U+0008
[9]github.com/nickboucher/imperceptible
[10]imperceptible.ml

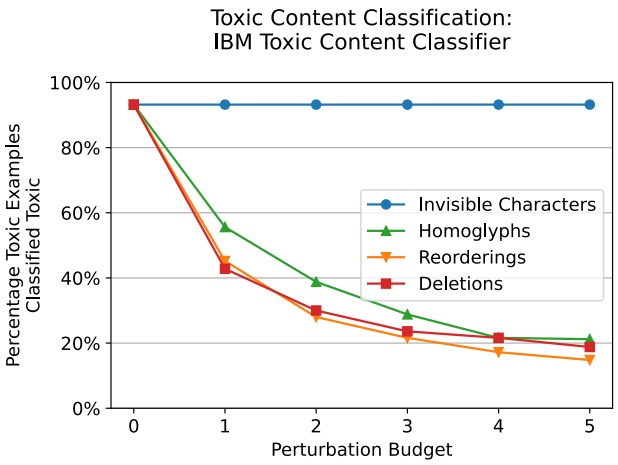

Fig. 8. Percentage of imperceptibly perturbed toxic sentences classified correctly in IBM's Toxic Content Classifier.

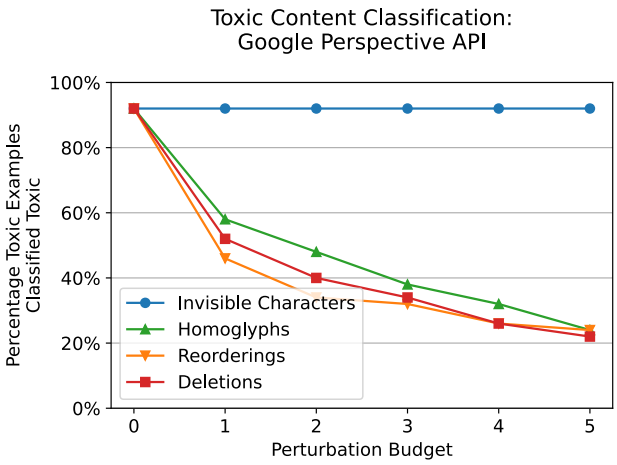

Fig. 9. Percentage of imperceptibly perturbed toxic sentences classified correctly in Google's Perspective API.

increase approximately linearly with reorderings having the largest distance.

C. Machine Translation: Availability

In addition to attacks on machine-translation model integrity, we also explored whether we could launch availability attacks. These attacks take the form of sponge examples, which are adversarial examples crafted to maximize inference runtime.

We used the same configuration as in the integrity experiments, crafting adversarial examples for 500 sentences with perturbation budgets of 0 to 5. Each example took, on average, 420 seconds to generate.

Sponge-example results against the Fairseq English-French model are presented in Figure 7, which shows that reordering attacks are by some ways the most effective. Levenshtein distances are also provided in Appendix Figure 15. Although the slowdown is not as significant as Shumailov et al. achieved by dropping Chinese characters into Russian text [**?**], our attacks are semantically meaningful and will not be noticeable to human eyes.

D. Machine Translation: MLaaS

In addition to the integrity attacks on Fairseq's open-source translation model, we performed a series of case studies on two popular Machine Learning as a Service (MLaaS) offerings: Google Translate and Microsoft Azure ML. These experiments attest to the real-world applicability of these attacks. In this setting, translation inference involves a web-based API call rather than invoking a local function.

Due to the cost of these services, we crafted adversarial examples targeting integrity for 20 sentences of budgets from 0 to 5 with a reduced maximum evolution iteration value of 3.

The BLEU results of tests against Google Translate are in Appendix Figure 16 and against Microsoft Azure ML in Appendix Figure 17. The corresponding Levenshtein results can be found in Appendix Figures 18 and 19.

Interestingly, the adversarial examples generated against each platform appeared to be meaningfully effective against the other. The BLEU scores of each service's adversarial examples tested against the other are plotted as dotted lines in Appendix Figures 16 and 17. These results show that imperceptible adversarial examples can be transferred between models.

E. Toxic Content Detection

In this task we attempt to defeat a toxic-content detector. For our experiments, we use the open-source Toxic Content Classifier model [**?**] published by IBM. In this setting, the adversary has access to the classification probabilities emitted by the model.

For this set of experiments, we craft adversarial examples for 250 sentences labeled as toxic in the Wikipedia Detox Dataset [**?**] with perturbation budgets from 0 to 5. Each example took, on average, 18 seconds to generate.

IBM Toxic Content Classification perturbation results can be seen in Figure 8. Homoglyphs, reorderings, and deletions effectively degrade model performance by up to 75%, but, interestingly, invisible characters do not have an effect on model performance. This could be because invisible characters were present in the training data and learned accordingly, or, more likely, the model uses a tokenizer which disregards the invisible characters we used.

F. Toxic Content Detection: MLaaS

We repeated the toxic content experiments against Google's Perspective API [**?**], which is deployed at scale in the real world for toxic content detection. We used the same experiment setting as in the IBM Toxic Content Classification experiments, except that we generated adversarial examples for 50 sentences. The results can be seen in Figure 9.

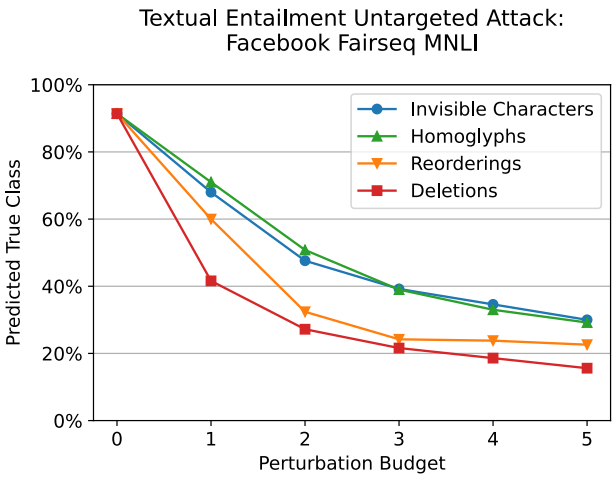

Fig. 10. Untargeted accuracy of Fairseq MNLI model with imperceptible perturbations

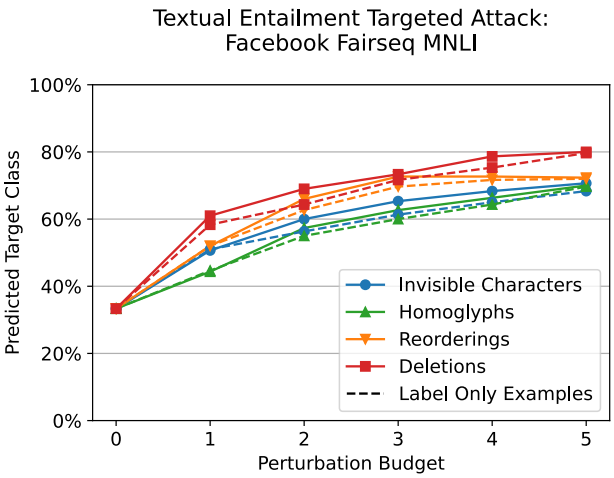

Fig. 11. Targeted accuracy of Fairseq MNLI model with imperceptible perturbations

G. Textual Entailment: Untargeted

Recognizing textual entailment is a text-sequence classification task that requires labeling the relationship between a pair of sentences as entailment, contradiction, or neutral.

For the textual-entailment classification task, we performed experiments using the pre-trained RoBERTa model [**?**] fine-tuned on the MNLI corpus [**?**]. This model is published by Facebook as part of Fairseq [**?**].

For these textual-entailment integrity attacks, we crafted adversarial examples for 500 sentences and repeated adversarial generation for perturbation budgets of 0 through 5. The sentences used in this experiment were taken from the MNLI test set. Each example took, on average, 51 seconds to generate.

The results from this experiment are shown in Figure 10. Performance drops significantly even with a budget of 1.

H. Textual Entailment: Targeted

We repeated the set of textual-entailment classification integrity experiments with targeted attacks. For each sentence, we attempted to craft an adversarial example targeting each of the three possible output classes. Naturally, one of these classes is the correct unperturbed class, and as such we expect the budget $= 0$ results to be approximately 33% successful.

Due to the increased number of adversarial examples per sentence, we crafted adversarial examples for 100 sentences and repeated adversarial generation for perturbation budgets of 0 through 5.

The results can be seen in Figure 11. These attacks were up to 80.0% successful with a budget of 5.

In the first set of targeted textual entailment experiments, we permitted the adversary to access the full set of logits output by the classification model. In other words, the differential evolution algorithm had access to the probability value assigned to each possible output class. We repeated the targeted textual entailment experiments a second time in which the adversary had access to the selected output label only, without probability values. These results are plotted as a dotted line in Figure 11, and were up to 79.6% successful with a budget of 5. Label-only attacks appear to suffer only a slight disadvantage, and even this diminishes as perturbation budgets increase.

I. Named Entity Recognition: Targeted

In addition to the Textual Entailment experiments, we also ran targeted attack experiments against the Named Entity Recognition (NER) task. We used a BERT [**?**] model [**?**] fine-tuned on the CoNLL-2003 dataset [**?**], which at the time of writing was the default NER model on HuggingFace [**?**]. We defined our attack as successful if one or more of the output tokens was classified as the target label, due to the fact that imperceptible perturbations typically break tokenizers and thus result in variable-length perturbed NER model outputs. We used the first 500 entries of the CoNLL-2003 test data split targeting each of the four possible labels using the same attack parameters as the prior experiments.

The attacks were up to 90.2% successful with a budget of 5 depending on the technique selected, although invisible characters had no effect on this model.

The results are visualized in Appendix Figure 20.

J. Sentiment Analysis: Targeted

In addition to Textual Entailment and NER, we also ran targeted attack experiments against the sentiment analysis task. We used a DistilBERT [**?**] model [**?**] fine-tuned on the Emotion dataset [**?**] published on HuggingFace [**?**]. We used the first 500 entries of the test data split of the Emotion dataset targeting each of the six possible labels using the same attack parameters as the prior experiments.

The attacks were up to 79.2% successful with a budget of 5 depending on the technique selection, although invisible characters also had no effect on this model.

The results are visualized in Appendix Figure 21.

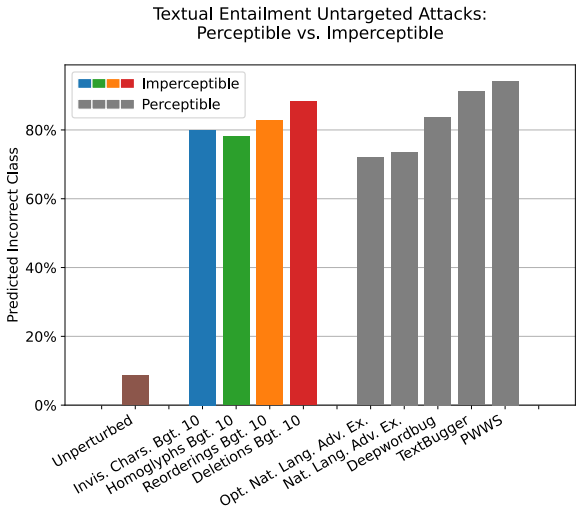

Fig. 12. Perceptible and imperceptible attack success rates against Facebook Fairseq RoBERTa MNLI.

K. Comparison with Previous Work

We selected five attack methods described in prior adversarial NLP work to compare with imperceptible perturbations. Of immediate note is that all prior work results in visually perceptible perturbations whereas imperceptible perturbations have no visual artifacts.

Despite this, we leveraged tooling provided by TextAttack [**?**] to compare all four classes of imperceptible perturbations against TextBugger [**?**], DeepWordBug [**?**], Probability Weighted Word Saliency [**?**], Natural Language Adversarial Examples [**?**], and an optimized version of Natural Language Adversarial Examples [**?**].

The results, shown in Figure 12, indicate that with a budget of 10, imperceptible perturbations have similar adversarial efficacy as the existing perceptible methods. Moreover, the imperceptible budget could be arbitrarily increased without visual effect for even better adversarial performance.

VII. Discussion

A. Ethics

We followed departmental ethics guidelines closely. We used legitimate, well-formed API calls to all third parties, and paid for commercial products. To minimize the impact both on commercial services and CO_2 production, we chose small inputs, maximum iterations, and pool sizes. For example, while Microsoft Azure allows inputs of size 10,000 [**?**], we used inputs of less than 50 characters. Finally, we followed standard responsible disclosure processes.

B. Experimental Interpretation

Applying imperceptible perturbations drastically degrades the performance of all models examined, representing NLP tasks including machine translation, textual entailment classification, toxic content detection, named entity recognition, and sentiment analysis. Every performance metric, whether BLEU translation score, percentage correct classification, or average inference time, was degraded relative to no perturbations (budget=0), with degradation growing as the perturbation budget was increased.

The only exception was invisible character attacks against toxic content, NER, and sentiment analysis models, which had no effect; this is likely indicative of invisible characters being present in training data, or the tokenizer for these models ignoring the chosen invisible characters. For every other technique/model combination, however, there is a clear relationship between increased imperceptible perturbations and decreased model performance.

To make translation quality loss more concrete, we provide an example of varying BLEU scores in Appendix B.

C. Search Engine Attack

Discrepancies between encoded bytes and their visual rendering affect searching and indexing systems. Search engine attacks fall into two categories: attacks on searching and attacks on indexing.

Attacks on searching result from perturbed search queries. Most systems search by comparing the encoded search query against indexed sets of resources. In an attack on searching, the adversary's goal is to degrade the quality or quantity of results. Perturbed queries interfere with the comparisons.

Attacks on indexing use perturbations to hide information from search engines. Even though a perturbed document may be crawled by a search engine's crawler, the terms used to index it will be affected by the perturbations, making it less likely to appear from a search on unperturbed terms. It is thus possible to hide documents from search engines "in plain sight." As an example application, a dishonest company could mask negative information in its financial filings so that the specialist search engines used by stock analysts fail to pick it up.

D. Attack Potential

Imperceptible perturbations derived from manipulating Unicode encodings provide a broad and powerful class of attacks on text-based NLP models. They enable adversaries to:

- Alter the output of machine translation systems;
- Evade toxic-content detection;
- Invisibly poison NLP training sets;
- Hide documents from indexing systems;
- Degrade the quality of search;
- Conduct denial-of-service attacks on NLP systems.

Perhaps the most disturbing aspect of our imperceptible perturbation attacks is their broad applicability: all text-based NLP systems we tested are susceptible. Indeed, any machine learning model which ingests user-supplied text as input is theoretically vulnerable to this attack. The adversarial implications may vary from one application to another and from one model to another, but all text-based models are based on encoded text, and all text is subject to adversarial encoding unless the coding is suitably constrained.

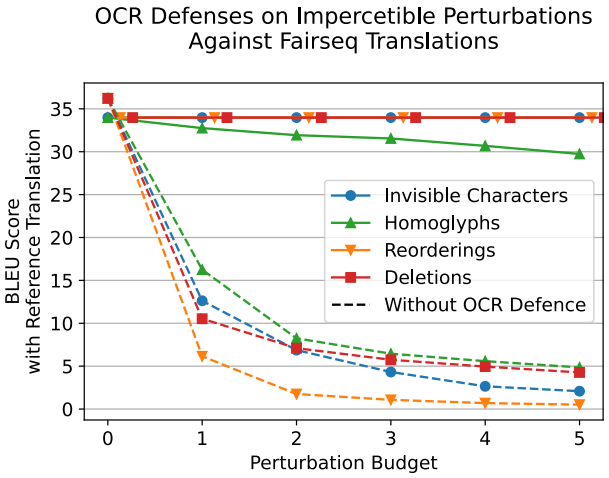

Fig. 13. Evaluation of OCR defense against imperceptible perturbations.

E. Defenses

Given that the conceptual source of this attack stems from differences in logical and visual text encoding representation, one catch-all defense is to render all input, interpret it with optical character recognition (OCR), and feed the output into the original text model. This technique is described more formally in Appendix Algorithm 3. Such a tactic functionally forces models to operate on visual input rather than highly variable encodings, and has the added benefit that it can be retrofitted onto existing models without retraining.

To evaluate OCR as a general defense against imperceptible perturbations, we reevaluated the 500 adversarial examples previously generated for each technique against the Fairseq En→FR translation model. Prior to inference, we preprocessed each sample by resolving control sequences in Python, rendering each input as an image with Pillow [**?**] and Unifont [**?**], and then performing OCR on each image with Tesseract [**?**] fine-tuned on Unifont. The results, shown in Figure 13, indicate that this technique fully prevents 100% of invisible character, reordering, and deletion attacks while strongly mitigating the majority of homoglyph attacks.

Our experimental defense, however, comes at a cost of 6.2% lowered baseline BLEU scores. This can be attributed to the OCR engine being imperfect; on some occasions, it outputs incorrect text for an unperturbed rendering. Similarly, it misinterprets homoglyphs at a higher rate than unperturbed text, leading to degraded defenses with the increased use of homoglyphs. Despite these shortcomings, OCR provides strong general defense at a relatively low cost without retraining existing models. Further, this cost could be decreased with better performing OCR models.

The accuracy and computational costs of retrofitting existing models with OCR may not be acceptable in all applications. We therefore explore additional defenses that may be more appropriate for certain settings.

1) Invisible Character Defenses: Generally speaking, invisible characters do not affect the semantic meaning of text, but relate to formatting concerns. For many text-based NLP applications, removing a standard set of invisible characters from inference inputs would block invisible character attacks.

If application requirements do not allow discarding such characters, tokenizers must include them in the source-language dictionary to create non-`<unk>` embeddings.

2) Homoglyph Defenses: Homoglyphs are perhaps the most challenging technique against which to defend. Functionally speaking, the OCR defense attempts to map unusual homoglyphs to their more common counterparts, thus increasing the likelihood that they are present in the NLP model's dictionary.

This mapping could be specified by model designers; a well-designed mapping of less-common homoglyphs to their most common counterparts applied prior to inference would have a similar effect to a high-performing OCR model. However, creating such a mapping is a daunting task, as the Unicode specification is immense. Automated techniques, such as previously depicted in Figure 4, may help to create these mappings.

3) Reordering Defenses: For some text-based NLP models with a graphical user interface, reordering attacks can be prevented by stripping all Bidi control characters as the input is displayed to the active user. In other settings, it may be more suitable to throw a warning for Bidi control characters.

A more general solution, however – and one that works for applications without a graphical user interface – is to apply the Bidi algorithm to resolve Bidi control characters and coerce the logical order of text to match the order in which it would be visually rendered.

4) Deletion Defenses: We suspect that there may not be many use cases where deletion characters are a valid input into a model. Deletion characters may be resolved prior to inference, or a warning may be fired upon their detection.

VIII. Conclusion

Text-based NLP models are vulnerable to a broad class of imperceptible perturbations which can alter model output and increase inference runtime without modifying the visual appearance of the input. These attacks exploit language coding features, such as invisible characters and homoglyphs. Although they have been seen occasionally in the past in spam and phishing scams, the designers of the many NLP systems that are now being deployed at scale appear to have ignored them completely.

We have presented a systematic exploration of text-encoding exploits against NLP systems. We have developed a taxonomy of these attacks and explored in detail how they can be used to mislead and to poison machine-translation, toxic content detection, textual entailment classification, NER, and sentiment analysis systems. Indeed, they can be used on any text-based ML model that processes natural language. Furthermore, they can be used to degrade the quality of search engine results and hide data from indexing and filtering algorithms.

We propose a variety of defenses against this class of attacks, and recommend that all firms building and deploying text-based NLP systems implement such defenses if they want their applications to be robust against malicious actors.

Acknowledgment

This work was supported by DARPA (through the GARD program), CIFAR (through a Canada CIFAR AI Chair), by NSERC (under the Discovery Program, and COHESA strategic research network), and by a gift from Intel. We also thank the Vector Institute's sponsors. Ilia Shumailov was supported with funds from Bosch-Forschungsstiftung im Stifterverband. We would also like to thank Adelin Travers for help with natural language annotation, Markus Kuhn and Zakhar Shumaylov for help with Unicode magic, and Darija Halatova for help with visualizations.

Appendix

A. *Machine Translation Fairseq Levenshtein Distances*

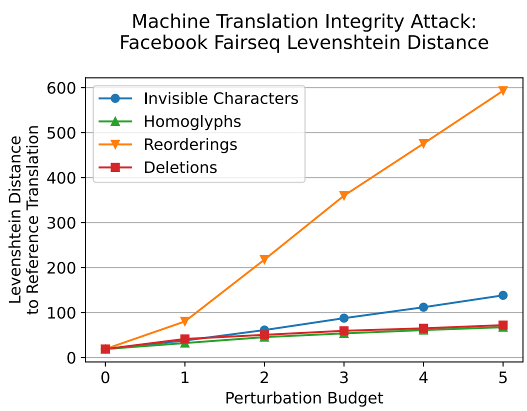

Fig. 14. Levenshtein distances between integrity attack imperceptible perturbations and unperturbed WMT data on Fairseq EN-FR model

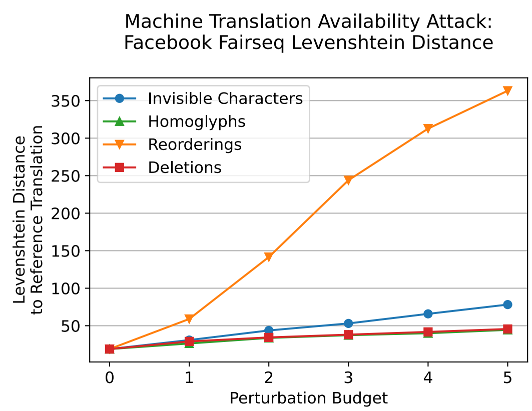

Fig. 15. Levenshtein distances between availability attack imperceptible perturbations and unperturbed WMT data on Fairseq EN-FR model

B. *Example BLEU Scores*

TABLE III
BLEU SCORES ACROSS VARYING INVISIBLE CHARACTER BUDGETS FOR THE INPUT "AND I THINK ABOUT MY FATHER." WITH REFERENCE TRANSLATION "ET JE PENSE À MON PÈRE." ON THE FAIRSEQ WMT14 EN→FR MACHINE TRANSLATION MODEL.

Budget	BLEU Score	Adversarial Example	Adversarial Translation
0	100	And I think about my father.	Et je pense à mon père.
1	19.3	And I think aU+200Dbout my father.	Et je pense que c' est un bout de course pour mon père.
2	12.4	And I thinkU+200D about my fatU+200Bher.	Et je pense que l' inquiétude au sujet de ma masse adipeuse l' inquiète.
3	1.9	AnU+200Bd I thiU+200Cnk about my fU+200Bather.	L " âme d" une personne ne doit pas être confondue avec l " âme d\' une autre personne.
4	1.9	AnU+200DdU+200C I think aU+200Bbout U+200Bmy father.	Un parent parent parent parent Je pense qu' un parent parent parent parent parent parent parent parent parent parent parent parent
5	0.9	AndU+200D I thiU+200CnkU+200B U+200Babout my fatheU+200Br.	Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Et Ma r. r. r.

C. Machine Translation MLaaS Results

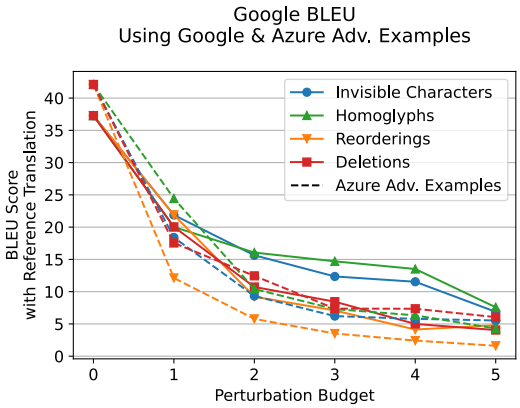

Fig. 16. BLEU Scores of Azure's imperceptible adversarial examples on Google Translate

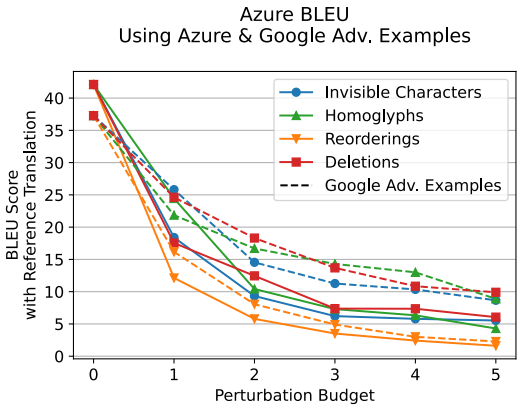

Fig. 17. BLEU Scores of Google Translate's imperceptible adversarial examples on Microsoft Azure

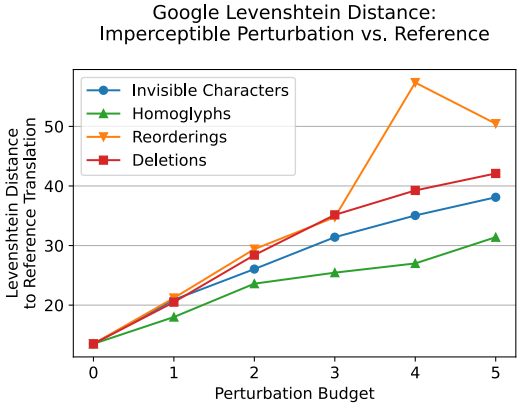

Fig. 18. Levenshtein distances between imperceptible perturbations and unperturbed WMT data on Google Translate's EN-FR model

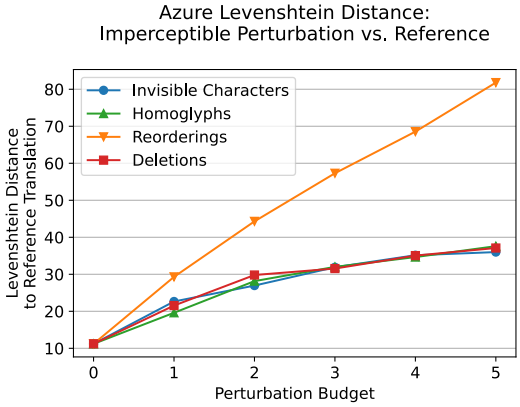

Fig. 19. Levenshtein distances between imperceptible perturbations and unperturbed WMT data on Microsoft Azure's EN-FR model

D. Multi-Class Targeted Classification Results

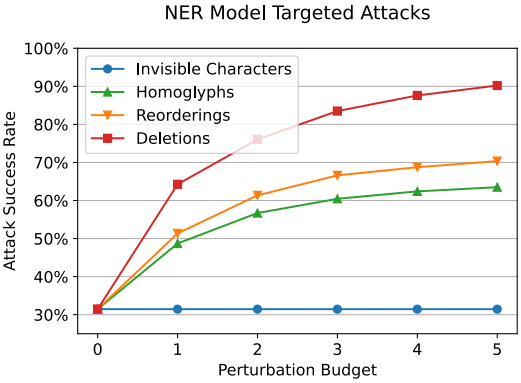

Fig. 20. Attack success rates for targeted Named Entity Recognition attacks against MDZ's CoNLL-2003 model with Imperceptible Perturbations

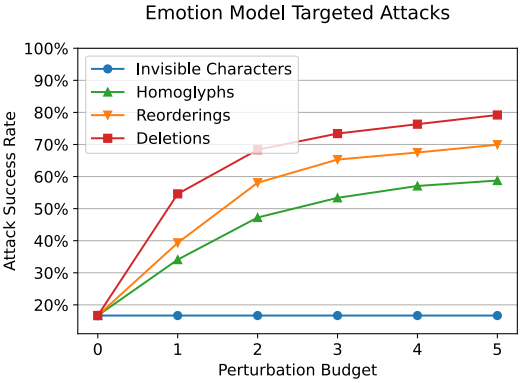

Fig. 21. Attack success rates for targeted sentiment analysis Imperceptible Perturbations attacks against DistilBERT fine-tuned on the Emotion dataset

E. Bidirectional Reordering Algorithm

Algorithm 2: Generation of 2^{n-1} visually identical strings via Unicode reorderings.

Input: `string` x of length n
Result: `Set` of 2^{n-1} visually identical reorderings of x

```
struct { string one, two; } Swap
string PDF := 0x202C, LRO := 0x202D
string RLO := 0x202E, PDI := 0x2069
string LRI := 0x2066

procedure SWAPS (body, prefix, suffix)
  Set orderings := { concatenate(prefix, body, suffix) }
  for i := 0 to length(body)-1 do
    Swap swap := { body[i+1], body[i] }
    orderings.add([prefix, body[:i],
                   swap, body[i+1:], suffix])
    orderings.union(SWAPS(suffix, [prefix, swap], null))
    orderings.union(SWAPS([prefix, swap], null, suffix))
  end for
  return orderings
end procedure

procedure ENCODE (ordering)
  string encoding := ""
  for element in ordering do
    if element is Swap
      swap = ENCODE([LRO, LRI, RLO, LRI,
                     element.one, PDI, LRI,
                     element.two, PDI, PDF,
                     PDI, PDF])
      encoding = concatenate(encoding, swap)
    else if element is string
      encoding = concatenate(encoding, element)
  end for
  return encoding
end procedure

Set orderings := { }
for ordering in SWAPS(x, null, null) do
  orderings.add(ENCODE(ordering))
end for
return orderings
```

F. OCR Defense Algorithm

Algorithm 3: OCR defense technique against imperceptible perturbations via input pre-processing.

Input: model input text $\mathbf{x}$
Result: pre-processed model input text $\mathbf{x}'$

x = resolve_control_chars(x) ▷ Apply Bidi+Deletion
i := render_text(x)
x' := ocr(i)
return x' ▷ Pass output to model