TrueSeeing: Defensive software against text-encoding based invisible attacks

Chunyu Jiang*, Guanyuming He*, Hanyu Zhang*, Kemu Xu*, Yifei Du* and Yilu Shi*
*School of Advanced Technology
Xi'an Jiaotong-Liverpool University, Suzhou, Jiangsu, China
Email: {chunyu.jiang20, guanyuming.he20, hanyu.zhang20, kemu.xu20, yifei.du20, yilu.shi20}@student.xjtlu.edu.cn

***Abstract*—Unicode is one of the most prominent text encoding standards used in modern computer systems. Its wide range of capabilities makes it possible for attackers to create sets of visually indistinguishable, binarily different texts encoded in Unicode. Most text-based systems, such as compilers, text-based generative AIs, and search engines, can be attacked by exploiting this vulnerability. In this paper, we present TrueSeeing, a program that can expose such invisible, text-based attacks to the naked eyes of a human user for corrections. Additionally, the neutralised text can be cryptographically signed by the system to ensure that it may never be modified again to recreate the attacks.**

***Index Terms*—Unicode, text-based, text-encodings, computer systems security**

I. Introduction

Texts, like all other data, are stored in binary in most computer systems. Therefore, to display them to a human user, that is, to transform them to the visible characters of a human-readable language, there must be some standard that defines how such characters are converted into binary data, and vice versa.

Unicode [1] is a dominating text encoding standard. A survey of W3Techs in 2024 [2] reports that one of Unicode's encoding formats, `UTF-8`, is used by 98.2% of investigated World-Wide-Web sites. One of the goals of Unicode is to provide a universal character encoding system for computers. In particular, Unicode needs to support the encoding of characters in many different languages. As a consequence, the Unicode standard is fairly complex, consisting of a total of 149,813 characters and various rules governing how the characters are interpreted and rendered, as of version 15.1.0 [1].

While the rich capabilities of Unicode suggest a unified system to present the vast set of human-readable characters, they also enable attacks in the form of different binary data encoded as imperceptibly distinct characters. For instance, The Cyrillic alphabet has many very similar characters to those in the Latin alphabet (`U+0071`[1] for Latin 'q' and `U+51B` for Cyrillic 'q'), and in practice those are usually rendered the same. Another example involves a clever exploit of special Unicode control (format) characters that can hide, reverse order of, insert, or delete some other characters in their context. Whereas a text editor usually interprets them correctly, other programs in the same software system may not, resulting in different meanings for the same text data.

As will be demonstrated later in this paper, such attacks can maliciously alter the functions of otherwise correct software systems, from search engines like Google and Bing, text-based generative AIs like ChatGPT, to compilers and interpreters like gcc and the Python interpreter. These tools each has a different yet crucial part in modern computer systems, so the consequences of such attacks could be devastating.

a) Our contribution: In this project, we identified aforementioned shortcomings of Unicode and text-based software systems using it, and revealed state-of-the-art attacks resulting from the weaknesses. Subsequently, we proposed a piece of software in defence of such attacks and implemented it.

b) Outline: The remainder of this paper is arranged in this order: In section II, fundamental mechanisms in Unicode are explained. We illustrate how problems of the mechanisms can result in four categories of attacks and present state-of-the-art examples. In section III, we formally define what security policy our software achieves under what threat model. Then, its implementation is briefly described. In section IV, we present the implementation details of the software (results of implementing the software) and showcase what it can achieve. In section V, we discuss our contributions and the limitations of our work. Also, we contemplate what works can follow ours in the future. In section VI, we compare our work with similar studies from different perspectives, considering similar attack examples, similar defence mechanisms, and more. At length, we conclude the paper in section VII.

II. Background

A. How Unicode works

According to the Unicode standard [1], characters in Unicode are contained in a continuous subset of the Integers: $[0, 17 \times 2^{16}) \subset \mathbb{Z}$, called the *code space*, in which each character is mapped to an integer called a *code point*. A code point is usually denoted by `U+` preceding the value of the integer in hexadecimal. For example, The Latin capital letter 'A' is mapped to the code point `U+0041`.

Binary data is converted to code points through one of Unicode's encoding formats. Popular formats include `UTF-8`, `UTF-16`, and `GB18030`.

The characters are further categorised into *graphical* characters that have a visual appearance and *format* characters that

[1]As will be discussed in section II-A, the `U+` notation is used to denote Unicode *code points*.

are invisible but affect how some surrounding characters are rendered.

Because graphical characters contain many visually similar ones and format characters are often handled differently by programs that present texts to humans and those process texts internally, there is ample opportunity to craft visually indistinguishable texts with different characters, which we will describe in the next subsection.

B. Attack Methods

Exploits of individual Unicode characteristics exist. For example, some studies [3], [4], [5], [6] investigate specifically into homoglyph (characters that correspond to visually indistinguishable glyphs) attacks targeting Internet domain names (webpage links) and file names; and Alvi *et al.* [7] examined how to detect homoglyphs in plagiarism detection systems.

However, not until 2022 had N. Boucher *et al.* created a taxonomy of such attacks, which they describe as "imperceptible perturbations" of Unicode-encoded texts, in their research [8, p. 1991] of applying such attacks to NLP (Natural Language Processing) systems. They summarise the classes of the attacks as:

> "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) **Reordering**: 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." [8, p. 1991],

where they employ the attacks to "perturb input to a model" (NLP model). In this paper, we consider the application of such attacks for perturbing input to any text-based software system.

C. Attack Examples

In this subsection, state-of-the-art research on successful attacks using the above methods to target real-world software systems will be demonstrated.

In section II-B, we already mentioned several instances before N. Boucher and R. Anderson created the taxonomy of such invisible text-based attacks. In fact, one particular form of the attacks, homoglyphs, has long been steadily used in impersonating domain names for at least 10 years, as summarised by G. Simpson *et al.* [9].

Following their creation of the taxonomy, N. Boucher and R. Anderson continued to realise concrete attack instances on important software systems. In 2023, they employed such attacks to implant an invisible trojan in some source code of nearly all programming languages [10] by making the source code interpreted differently by a compiler and by human eyes (text editors). For example, they could alter the control flow of a function, comment out code, and change strings all without a programmer's notice. Later that year, together with I. Shumailov *et al.* in another study [11], they proved that by providing malicious search query texts that are visually the same as some other harmless texts, neutral and uncompromised search engines (e.g. Google and Bing) could be misled into displaying search results that the attacker wants the user to see. This discovery suggests that it is possible for an adversary to apply the technique in a misinformation campaign, or for an evil government to censor truth and brainwash its citizens.

Apart from N. Boucher and R. Anderson's works, a few other research can be found using such attacks. A. Dionysiou and E. Athanasopoulos [12] suggested an ATGF (Adversarial Text Generation Frameworks) that can mislead the classification results of NLP (Natural Language Processing) models with the concerned attacks. A. Compagno *et al.* proposed [13] how attackers can use non-printable (invisible) characters in Unicode to conceal information, in order to prevent the C& C (Command and Control) channel of their botnets from being detected.

III. Methodology

In this section, how TrueSeeing works will be illustrated. First, the security policy (goals) and threat model are formally defined. Then, how the software achieves the policy given the threat model is described.

A. Security Policy

Suppose an English user of a computer needs to use various software systems that take input in texts encoded in Unicode. Assume the texts that the user inputs into the software systems may come from any source, but the user will visually check the texts with his eyes.

The security policy of TrueSeeing is that, provided the user runs TrueSeeing on the input text, *all* possible attacks originated from the aforementioned four attack classes in section II-B will be successfully exposed to the user visually. The software will then allow the user to remove all attacks injected in the text (the implementation is specified in section III-C). After all the attacks are removed, the software enables the user to cryptographically sign the neutralised text to prevent any possible editing to it in the future.

B. Threat Model

First, the threat model assumes that an adversary cannot alter the function of the hardware as well as any software system the user uses, including the operating system and TrueSeeing. However, assume an adversary has infinite knowledge about any of the software systems, including their complete specifications and source code. Besides, assume an adversary has a thorough understanding of the Unicode standard. Moreover, assume that all such systems are free of program bugs, but possibly have only partial implementation of the Unicode standard.

It is presumed that the texts used by the user may come from any source, including those directly controlled by an adversary. For example, a text may be copied by the user from a malicious social media post, or transmitted from an

unknown source through the Internet to the user's computer. Because an adversary is not allowed to alter anything locally on the user's computer, in this threat model, providing any text data to the user is considered the only way an adversary can have influence on the user.

Equivalently, one may assume that an adversary has complete control over any text the user may take into his software systems. Nevertheless, once the text has entered the user's machine, the adversary will lose *all* control and the user will *always* visually inspect the text to check if the visual of the text is harmless.

C. How TrueSeeing Works

Because the user is assumed to use English, TrueSeeing will only consider two encoding formats of Unicode, `UTF-8` and `UTF-16`. When the software is launched, it will ask the user to choose from the two.

After that, it prompts the user to provide the text for examination. Because the threat model disallows an adversary to modify any text within the user's system, the software will assume that the user stores a text obtained from external sources either in the system clipboard (e.g. the user copies an external text), or in a local file (e.g. external text is transmitted through a network and received as a file in the user's machine). After the user selects where the text is stored, the software loads the text into the system as-is (i.e. in binary).

Then, the system maps the binary data into Unicode code points that represent characters, according to the user-selected encoding format. The system stores the converted code points in memory, and processes them as follows in two different pipelines (named `p1` and `p2`).

- `p1` happens exactly as the Unicode standard specifies. All the format characters are effective, and the graphical characters are rendered as Unicode intends them to be rendered.
- In contrast, in `p2`, the system will divide the characters into four categories.
  1) (Cat1) English graphic characters that can be encoded in another text-encoding standard, ASCII (American Standard Code for Information Interchange), in the version [14] standardised by the International Organization for Standardization. They contain all English characters in lowercase and uppercase, as well as all punctuation. Table I shows all the characters and their code points in ASCII vs. Unicode.
  2) (Cat2) All other Unicode graphic characters.
  3) (Cat3) The harmless Unicode format characters, `HT` (Horizontal Tabulation), `LF` (Line Feed), `CR` (Carriage Return). `HT` is used for tabulation to align characters to the same horizontal position. `LT` and `CR` indicate the end of a line. All of the three characters also have their counterparts in ASCII.
  4) (Cat4) All other Unicode format characters, which are potentially harmful.

  Here, all format characters lose their ability entirely. Instead, they are displayed on the screen as if they were normal graphic characters. To distinguish the categories, each is rendered in a different colour, on a different background.

char	code	char	code	char	code
				␣	0x20
!	0x21	”	0x22	#	0x23
$	0x24	%	0x25	&	0x26
’	0x27	(	0x28	)	0x29
*	0x2A	+	0x2B	,	0x2C
-	0x2D	.	0x2E	/	0x2F
0 – 9 : 0x30 – 0x39					
:	0x3A	;	0x3B	<	0x3C
=	0x3D	>	0x3E	?	0x3F
@	0x40				
A – Z : 0x41 – 0x5A					
[	0x5B	\	0x5C	]	0x5D
^	0x5E	_	0x5F	‘	0x60
a – z : 0x61 – 0x7A					
{	0x7B	|	0x7C	}	0x7D
~	0x7E				

TABLE I
ENGLISH GRAPHIC CHARACTERS IN ASCII AND UNICODE

The results of `p1` and `p2` are displayed in parallel on the screen by TrueSeeing. The user is able to edit in the area for `p2`, where he can delete any character or insert an English graphic character by typing on an English keyboard. The change will be made on the code points stored in the memory, so it will be reflected in both `p1` and `p2`.

When the system detects that none of the code points in memory is from categories 2) and 4), it will deem the text as benign, and then enables the user to cryptographically sign the text.

The signing is performed using a digital signature, where the software will employ a public-key cryptography algorithm. A pair of public key and private (secret) key (k_p, k_s) will be generated and stored within the software, but also within the user's reach. Because no adversary is assumed to be able to break into the user's system, the private key is safe from everyone except the user himself. The user can distribute the public key k_p to any recipient of the data signed. For which algorithm the software uses, see Section IV-A4.

Digital signature created using a public-key algorithm has been proven to be able to ensure three important security properties of the data signed:

- **Integrity**: the data is not modified after being signed.
- **Authenticity**: a recipient can be confident that the data signed is from the user.
- **Non-repudiation**: the user cannot argue that the data is not originated from him.

Provided that the user does not leak the private key k_s from his system, the three properties can then be achieved.

IV. IMPLEMENTATION DETAIL & RESULTS

In this section, first we describe the implementation details as the result of executing our implementation plan of the software. Then, the software itself is demonstrated.

A. Implementation detail

1) Importing text: As mentioned in Section III-C, users can import text either from the clipboard or by uploading a file. We use system APIs to read binary data from the two places into a buffer. This buffer is then converted into a string of Unicode code points using the encoding format selected by the user. Then, the text is sent to p1 and p2.

p1 will process the text as Unicode specifies, and its result will be displayed in a box on the top of the GUI, which is titled "Original Text". p2 will detect potentially harmful characters, and its result is displayed with different colours in another GUI box under the previous one, which is titled "True Text".

2) Detecting and colouring dangerous characters: In p2, the system iterates through all Unicode code points and classifies them into the four distinct categories defined in Section III-C.

Using the `repr()` function in Python, any code point that is not in Cat1 is converted into its real binary representation before being displayed on the "True Text" area. The type of each is determined by checking which category the binary value belongs to, and the corresponding colour for the foreground and that for the background are then applied to the GUI:

- Cat1 code points are displayed as black characters on a white background.
- Cat2 code points are displayed as dark blue characters on a light blue background.
- Cat3 code points are displayed as dark green characters on a light green background.
- Cat4 code points are displayed as dark red characters on a light red background.

Because some code points cannot be displayed directly, the system will display their corresponding hexadecimal Unicode code with an escaping character \. For example, `\x08` represents the eighth character, Backspace. This is done by Python's `repr()` function. There are two types of special cases, one is for harmless format characters, we use `\n` for line feed, `\r` for carriage return, and `\t` for (horizontal) tabulation. For the second special case, the escape character \, it is represented by `\\` in the True Text area. After that, all characters are added to the True Text area by applying the appropriate font colour and background colour according to their categories. The exact display settings are shown in Section IV-B.

3) Correcting true text: Section III-C demands that the user is only able to edit in the "True Text" GUI. Users may delete the harmful characters or add harmless ones. The True Text area will be bound with an edit-checking function, a benign checking function and a callback function. The edit-checking function first detects each user input to the True Text area, as it will only accept harmless characters from the keyboard. If the input character belongs to the potentially harmful characters, then it will be rejected. For harmless characters,

- ASCII graphical character can be directly inserted via the keyboard.
- If instead the character is a format character (i.e. in Cat3), then it will be read in the escaped forms, namely `\n`, `\r` or `\t`.

To convert the content in the "True Text" area back to code points in the buffer, the `eval()` function is called. This function finds every code point represented as a \ plus a code number, and interprets them into the corresponding code point.

If, after an update, the system finds that none of the code points in the buffer belongs to Cat2 and Cat4, the string is marked as benign.

4) Signing the text: Once the code points are marked as benign, the system enables them to be signed as a digital signature. The software uses a digital signature algorithm based on RSA that was first introduced by M. Bellare and P. Rogaway in 1996 [15]. It is called the "*Full-domain-hash scheme*" in the paper.

The first step is the same as the usual RSA — generation of public and private keys. The Python library `Crypto` is used to generate key pairs. This generates

$$k_p = (N, e), \quad k_s = (N, d),$$

where k_p is the public key, and k_d is the private key. The key length is set to 2048 bits to have enough security.

Then, we need to choose a hash function. Recall the name is a full-domain-hash scheme, which means the domain of the hash function has to be the full domain of the RSA function, that is, $\mathbb{Z}_N^*$ [15, Section 1]. That paper suggests that in practice SHA-1 can be used [15, Section 3], but considering that it was published several years ago, we choose a more secure hash function, H = SHA-256, using the Python method `Cryto.SHA256`.

The signing and verifying functions, $Sign$ and $Verify$, are defined as decrypting the hashed message and encrypting the signature, respectively [15, Section 3]:

$$Sign(M) = H^d(M) \bmod N$$

$$Verify(M, x) = \begin{cases} 1 & \text{if } x^e \bmod N = H(M) \\ 0 & \text{otherwise} \end{cases}$$

M. Bellare and P. Rogaway have proven that if $\mathcal{RSA}$ is (t', ϵ')-secure, then the signature scheme will be $(t, q_{sig}, q_{hash}, \epsilon)$-secure [15, Section 2.2].

B. Software demonstration

The user interface of TrueSeeing is built with the Python package `Tkinter`. The main windows has three main panels: *Encoding Format*, *Text Examination*, and *Signature Production*. These panels guide users through the process of text examination and signature generation. In addition, we provide informative tips and warning messages where necessary.

1) Encoding format GUI: In this panel (Figure 1), users can choose between two encoding formats, UTF-8 and UTF-16, using radio buttons. The default selection is UTF-8. A "Select" button at the bottom allows users to confirm their choice and proceed with the selected encoding format.

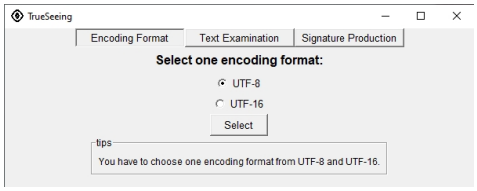

Fig. 1. The Unicode encoding format GUI panel

2) Text examination GUI: The Text Examination panel (Figure 2) is divided into "*Original Text*" and "*True Text*" areas. Users can load suspicious text to be displayed in the two areas by two buttons "*Load from system clipboard*" and "*Load from a local file*".

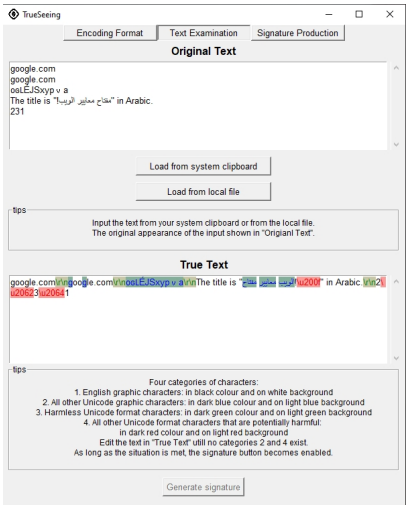

Fig. 2. The text examination GUI panel

Users can directly edit the text in the "True Text" area. The "Generate Signature" button remains disabled until all potentially harmful characters are removed.

3) Digital signature GUI: After the user clicks on the "Generate Signature" button, he is taken to the Signature Production panel (Figure 3), where the generated digital signature is displayed in a text box in hexadecimal.

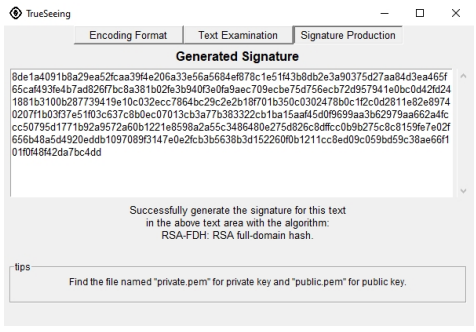

Fig. 3. Digital signature GUI panel

V. Discussion

In this section, we will summarise the abilities of TrueSeeing and discuss its limitations. Simultaneously, we will propose possible future improvements to these limitations.

TrueSeeing effectively addresses all four classes of attacks, namely invisible characters attacks, homoglyph attacks, reordering attacks, and deletion attacks. It provides users with robust defence against text-encoding based invisible attacks by categorizing them into different harmful levels and rendering characters in four different colours: black, blue, green, and red. This visually distinct representation alerts users to the presence of potential attacks, ensuring they can detect and rectify any attempts to perturb the input to a model using invisible characters. However, its methodology has a few limitations that are welcome to possible improvements in the future:

- **Multi-Lingual Support:** As TrueSeeing currently only supports English texts and two Unicode encodings UTF-8 and UTF-16, expanding its capabilities to support other languages and encoding formats would significantly broaden its utility.
  Although supporting an additional language might be as simple as add characters from category 2) to category 1) in TrueSeeing, some languages may still require careful consideration of linguistic nuances and diverse character sets. In particular, languages like Chinese and Japanese have thousands of graphical characters, and it is possible that some of them may enable new attacks if added blindly into category 1).
- **Expand Scope:** TrueSeeing only targets text-based attacks encoded in Unicode, focusing on specific classes of imperceptible perturbations outlined in existing research. Therefore, it may not able to adequately address other types of vulnerabilities or attacks targeting text-based software systems, should they emerge in the future.
- **Real-Time Protection:** Currently TrueSeeing relies on the user to input a possibly malicious piece of text to it. Acting as an independent software system, it is not able to monitor important text-communication channels in the user's operating system. For instance, it is not able to monitor network traffic and find malicious texts within.
  By integrating it with existing software systems, TrueSeeing could be used actively to intercept and neutralize malicious text inputs before they reach the user. Implementing such real-time monitoring and protection features on top of TrueSeeing would enable continuous defence against evolving text-based attacks.

Overall, TrueSeeing still represents a significant step towards mitigating text-encoding based invisible attacks, offering valuable insights for defending against such vulnerabilities. However, continued research and development are necessary to address its limitations and further enhance its effectiveness in safeguarding text-based systems against evolving threats.

VI. Related Works

In this section, studies that are related to our project are discussed. We start by presenting different defence measures against investigated attacks, and then describe attacks that are not the same but similar to the investigated ones.

A. Related defence measures

A family of attack methods in the form of (3) in section II-B are recorded as a CVE record (The CVE Program [16] aims to "identify, define, and catalogue publicly disclosed cybersecurity vulnerabilities."), CVE-2021-42574 [17], which identifies the possible attacks using "Bidirectional" control characters in Unicode.

A lot of industry software dealing with source code now has implemented defensive mechanisms against CVE-2021-42574. For instance, GitHub will issue a warning if it finds such characters inside source code [18]; in Microsoft Visual Studio 2022 Version 17.0.3, the issue is reported as "addressed", as they have enabled the editor to detect and render such control characters [19]. The GNU Compilers Collection (GCC) now allows users to specify `-Wbidi-chars=[none|unpaired|any|ucn]` so that the compiler can "*warn about possibly misleading UTF-8 bidirectional control characters in comments, string literals, character constants, and identifiers*" [20].

These industry solutions work in a very similar way as our software, but they

1) Only address one specific family of such attacks: CVE-2021-42574.
2) Unlike our independent TrueSeeing, they are integrated into individual software systems.

Because ultimately the vulnerability enabling such attacks originates from the different interpretations of the same Unicode texts by different programs, N. Boucher *et al.* proposed that OCR (optical character recognition) technology can be used to defend against such attacks by using OCR to recreate the text from what is visually displayed [8, sec. VII.E]. This way there is only one interpretation of a given Unicode text.

The OCR defence strategy is a strong measure against nearly all types of such attacks. However, compared with our software, they suffer from:

1) Accuracy and ambiguity. OCR systems cannot 100% capture texts from pictures. In addition, they cannot distinguish homoglyphs that are *exactly the same*.
2) Speed. OCR systems may employ machine learning models, which are naturally slower than TrueSeeing's simple algorithm.
3) Their own vulnerabilities. Research has demonstrated that they are vulnerable to hardly perceptible attacks [21].

B. Similar attack strategies

Although the taxonomy of attacks used in this paper covers a broad range of attack methods, there are also different ways to exploit a text-based software system. Moreover, similar attack techniques have been applied beyond the scope of text-based systems.

1) Similar attack on different systems: Since texts are not the only visual elements human users see on a computer screen, it is not surprising that imperceptible attacks exist to exploit other visual elements of a computer system.

Like Unicode, there are also several standards or formats for the binary representation of images, such as PNG (Portable Network Graphics) and JPEG (Joint Photographic Experts Group). These standards are likely to possess vulnerabilities like Unicode does, thus enabling imperceptible attacks on them.

For example, numerous studies have been dedicated to improving the robustness and imperceptibility of image watermarks [22], [23], [24], [25]. Image watermarking is the technique of inserting watermark information into an image; a robust watermark can preserve the information during many different image transformations; and an imperceptible watermark cannot or is very hard to spot by a human. While the technique itself is quite neutral, there are malicious use cases of it in different attacks. For instance, an evil company may insert an employee's information imperceptibly into the user interface of the company's communication software. When the employee takes a screenshot of some evidence of the company's wrongdoings, the evil company will then be able to track down who the employee is and take revenge on him.

2) Different attacks on text-based systems: If we overlook the imperceptibility of attacks, then there are an enormous amount of adversarial studies on text-based systems. In recent years, a large amount of efforts [26], [27], [28], [29], [30] has been devoted to attacking text-based *deep neural networks*, which are the foundation of deep learning. These attacks are often called *adversarial examples* in the domain of machine learning.

Apart from the imperceptible attacks investigated in the paper and perceptible attacks mentioned above, a noteworthy study discovered attacks that nearly imperceptibly alter the input text [21]. Adding a hat to 'a' to get 'â' is nearly imperceptible in a large amount of text, but the study has proven that it can have a significantly damaging effect on some image-to-text systems like OCR and ViT.

VII. Conclusion

This report first investigates the vulnerabilities inherent in the Unicode standard. As Unicode serves as a universal character encoding system for computers, weaknesses in its specification inadvertently open the door to attacks which craft visually similar but semantically distinct texts. These attacks, including the manipulation of invisible characters, homoglyphs, reordering, and deletions, pose significant threats to the integrity and functionality of vital software systems, from search engines to compilers and interpreters.

To address the potential challenge and threat, this paper proposes TrueSeeing, a defensive software designed to combat these attacks. It exposes such originally invisible attacks to human users visually. By categorising characters into different

groups and rendering them accordingly, TrueSeeing enables users to identify and neutralize potential attacks embedded within texts. Furthermore, the software offers the additional security feature of cryptographically signing the neutralised text using RSA-FDH, ensuring its integrity, authenticity, and non-repudiation.

In summary, TrueSeeing is a noteworthy piece of defensive software against text-encoding based invisible attacks, providing users with the tools necessary to safeguard against malicious manipulations of text data. As technology continues to evolve, solutions like TrueSeeing will play a crucial role in ensuring the integrity and security of digital systems. Because this is an independent system, future works can integrate it into existing text-based system to create a safer text-based software pipeline.

REFERENCES

[1] The Unicode Consortium, *Unicode Standard,* 15.1.0, The Unicode Consortium, 2023, accessed 2 April 2024. [Online]. Available: https://www.unicode.org/versions/Unicode15.1.0/

[2] W3Techs. (2024) Usage of character encodings broken down by ranking. Accessed 2 April 2024. [Online]. Available: https://w3techs.com/technologies/cross/character_encoding/ranking

[3] L. J. Sern, Y. G. Peng David, and C. J. Hao, "Phishgan: Data augmentation and identification of homoglyph attacks," in *2020 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*, 2020, pp. 1–6.

[4] A. Ginsberg and C. Yu, "Rapid homoglyph prediction and detection," in *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, 2018, pp. 17–23.

[5] Y. Lu, M. K. K, N. Mohammed, and Y. Wang, "Homoglyph attack detection with unpaired data," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 377–382. [Online]. Available: https://doi.org/10.1145/3318216.3363337

[6] J. Woodbridge, H. S. Anderson, A. Ahuja, and D. Grant, "Detecting homoglyph attacks with a siamese neural network," in *2018 IEEE Security and Privacy Workshops (SPW)*, 2018, pp. 22–28.

[7] F. Alvi, M. Stevenson, and P. Clough, "Plagiarism detection in texts obfuscated with homoglyphs," in *Advances in Information Retrieval*, J. M. Jose, C. Hauff, I. S. Altıngovde, D. Song, D. Albakour, S. Watt, and J. Tait, Eds. Cham: Springer International Publishing, 2017, pp. 669–675.

[8] N. Boucher, I. Shumailov, R. Anderson, and N. Papernot, "Bad characters: Imperceptible nlp attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1987–2004.

[9] G. Simpson, T. Moore, and R. Clayton, "Ten years of attacks on companies using visual impersonation of domain names," in *2020 APWG Symposium on Electronic Crime Research (eCrime)*, 2020, pp. 1–12.

[10] N. Boucher and R. Anderson, "Trojan source: Invisible vulnerabilities," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6507–6524. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/boucher

[11] N. Boucher, L. Pajola, I. Shumailov, R. Anderson, and M. Conti, "Boosting big brother: Attacking search engines with encodings," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 700–713. [Online]. Available: https://doi.org/10.1145/3607199.3607220

[12] A. Dionysiou and E. Athanasopoulos, "Unicode evil: Evading nlp systems using visual similarities of text characters," in *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–12. [Online]. Available: https://doi.org/10.1145/3474369.3486871

[13] A. Compagno, M. Conti, D. Lain, G. Lovisotto, and L. V. Mancini, "Boten elisa: A novel approach for botnet c& c in online social networks," in *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015, pp. 74–82.

[14] The International Organization for Standardization, *ISO/IEC 646:1991, ISO 7-bit coded character set for information interchange*, 1991, reviewed 2020, accessed 4 April 2024. [Online]. Available: https://www.iso.org/standard/4777.html

[15] M. Bellare and P. Rogaway, "The exact security of digital signatures-how to sign with rsa and rabin," in *Advances in Cryptology — EUROCRYPT '96*, U. Maurer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 399–416.

[16] The CVE Program, "About the CVE program," 2024. [Online]. Available: https://www.cve.org/About/Overview

[17] ——, "CVE-2021-42574," Nov. 1 2021. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-42574

[18] The GitHub Team, "Warning about bidirectional Unicode text," 2021. [Online]. Available: https://github.blog/changelog/2021-10-31-warning-about-bidirectional-unicode-text/

[19] Microsoft Visual Studio Team, "Visual studio 2022 version 17.0.3," 2021. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/releases/2022/release-notes-v17.0

[20] The Free Software Foundation, Inc., "3.8 options to request or suppress warnings," 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#index-Wbidi-chars_003d

[21] N. Boucher, J. Blessing, I. Shumailov, R. Anderson, and N. Papernot, "When vision fails: Text attacks against vit and ocr," 2023.

[22] J. Abraham and V. Paul, "An imperceptible spatial domain color image watermarking scheme," *Journal of King Saud University - Computer and Information Sciences*, vol. 31, no. 1, pp. 125–133, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157816301483

[23] K. Prabha and I. Shatheesh Sam, "An effective robust and imperceptible blind color image watermarking using wht," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 6, Part A, pp. 2982–2992, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157820303384

[24] P. W. Adi, Y. P. Astuti, and E. R. Subhiyakto, "Imperceptible image watermarking based on chinese remainder theorem over the edges," in *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, 2017, pp. 1–5.

[25] F. Rahimi and H. Rabani, "A visually imperceptible and robust image watermarking scheme in contourlet domain," in *IEEE 10th INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING PROCEEDINGS*, 2010, pp. 1817–1820.

[26] S. Liu, N. Lu, W. Hong, C. Qian, and K. Tang, "Effective and imperceptible adversarial textual attack via multi-objectivization," *ACM Trans. Evol. Learn. Optim.*, mar 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3651166

[27] J. Dai, C. Chen, and Y. Li, "A backdoor attack against lstm-based text classification systems," *IEEE Access*, vol. 7, pp. 138 872–138 878, 2019.

[28] 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)*, 2018, pp. 50–56.

[29] S. Samanta and S. Mehta, "Towards crafting text adversarial samples," 2017.

[30] B. Liang, H. Li, M. Su, P. Bian, X. Li, and W. Shi, "Deep text classification can be fooled," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, ser. IJCAI-2018. International Joint Conferences on Artificial Intelligence Organization, Jul. 2018. [Online]. Available: http://dx.doi.org/10.24963/ijcai.2018/585