METHODS OF DETECTING AND RECOVERING AFTER HOMOGLYPH ATTACKS

Roman TOLSTOSHEYEV

Baku Higher Oil School, Baku, Azerbaijan

The problem of homoglyph attacks has become significant nowadays. Due to the increasing range of characters in modern encoding systems, similar symbols are being added. This tendency causes problems in distinguishing characters, which hackers exploit in spoofing attacks. In this research, an attempt is made to examine the problem of homoglyph attacks and provide an overview of practical solutions, such as different algorithms. Additionally, a solution to address the issue of spoofing attacks using the Levenshtein edit distance algorithm, widely used in spell checking, will be provided. The proposed solution, which utilizes a backend application written in the Java programming language with the Spring technology stack, can detect homoglyphs and attempt to replace them with normal characters. Additionally, a desktop client for the application is also presented.

Key words: *homoglyph, UTF-encoding, obfuscation, spoofing, dynamic programming.*

1. NTRODUCTION

The main topic of this work is to analyze current solutions against homoglyph attacks and evaluate their positive and negative aspects and propose a new solution. Chapter 2 provides a brief description of the history and prevalence of encoding systems, as well as a full description of homoglyphs and their examples, to establish the genesis of the research problem. Chapter 3 depicts and classifies the security issues regarding homoglyph attacks. The systematization given in this chapter helps identify where this kind of attack is most likely to be applied.

Chapter 4 attempts to analyze previously proposed solutions to the problem, comparing their merits and demerits. This chapter also provides implementations of the main algorithms in the Java programming language. Chapter 5 describes a suggested method to oversee the problem, divided into two sub-chapters. This chapter aims to propose a solution that meets all requirements and is more effective than existing solutions by considering one or more specific traits. Additionally, that chapter introduces a solution called "Ambiglyph", which was specially created during this research to solve the homoglyph attack problem. The proposed solution implements the aspects mentioned in earlier chapters, providing a comprehensive description of "Ambiglyph", its main components, and their working principles.

2. CHARACTER ENCODING STANDARDS

Encoding systems play a crucial role in information exchange. The problem of encoding characters starts from the epoch of communication via telegraph, when people started to encode their text messages using electric signals (McEnery and Xiao:2005, p.47). The most popular was encoding using Morse code which used short and long electric signals for encoding letters, digits and punctuation marks and long absence of signal to separate characters.

With the appearance of computers, the issue of encoding messages became acute. Data was to be encoded using a sequence of bits that were related to the presence and absence of signal. Hence, binary logic of computers with "long signal" incommutability led to new encoding systems were to be invented.

Early encoding systems used 5 and then 6 bits to encode each character; however, it was insufficient and 7-bits encoding systems were in use. The most noticeable example of this kind of system is the American Standard Code for Information Interchange or ASCII announced in 1963. ASCII is used to encode all litters of the Latin English alphabet, digits, punctuation marks and special characters. ASCII was adopted by all computer manufacturers of that time and turned into standard ISO 646 by the International Organization of Standardization (ISO) in 1972.

The significant demerit of ASCII character was that it was adopted only for English letters and did not include native characters of some other countries and languages. The solution of that time was to introduce a new set of characters. The series of ISO standards ISO-8858-X and usage of 8-bit encoding were turned to solve this problem. Additionally, in some languages like Chinese, it was impossible to encode all necessary characters using 8 bits.

The large variety of ASCII character tables created a significant problem in information exchange. Sometimes, it was not practical to process text due to the lack of supported ASCII tables of a software or its regional settings. In other cases, text files can be damaged in case of using inappropriate ASCII encoding format. Furthermore, other encoding systems were still in use: JIS (Japan), GB (China), GOST (USSR), EUC (Unix), CP (IBM, Microsoft), and many more. Most of them we incompatible with one another, creating obstacles in decoding.

UTF-8 (Q-Success:2024). Modern character encoding standards try to cover as many practical symbols used by humans as possible to be more versatile according to the variety of natural and formal languages. This versatility can cause problems related to the visual identity of

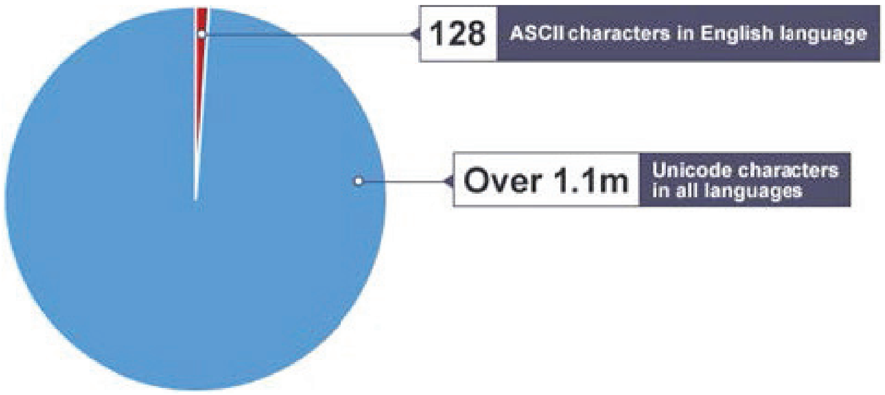


Fig. 1 Comparison of ASCII and Unicode character sets

The establishment of Unicode in early 1989 played a key role in addressing these problems. Unicode encoding system gives a chance to store a character in several bytes and allows people to enlarge characters set in case of necessity with saving backward compatibility with ASCII. Unicode standard updates regularly and adds new symbols. The latest version of Unicode for May 2024 is Unicode 15 (Unicode inc.:2021). The comparison of ASCII and Unicode characters set is given in Figure 1.

Unicode has several implementations; the most popular ones are UTF-8 and UTF-16. 97% of websites around the world use

some symbols and the probability of mismatching them (Miller:2013, p.4). Modern encoding systems such as Unicode use more than 140,000 symbols in version 14, with about 6,000 concerns regarding its visual similarity and symbol understanding (Unicode inc.:2021).

The poor contrast in the visual representation of some characters can depend on the fonts used in their display (without considering the quality, technical parameters, and settings applied to a display device) (The MITRE Corporation:2017). However, commonly used fonts can display several different symbols with extremely limited differences or even without them, proving to be an issue.

More formally, symbols that resemble one another are called homoglyphs. A homoglyph can be a full copy of another character or resemble it. For instance, the Latin character "O" (UTF-16 code is \u004f) can be mismatched with the Cyrillic "O" (\u041e) and the digit zero "0" (\u0030). In the first case, it is almost impossible to distinguish the difference between the two characters, while in the second case, it is possible in most instances (Miller:2013, p.4). As an example, possible homoglyphs of letter "A" are provided on Figure 2.

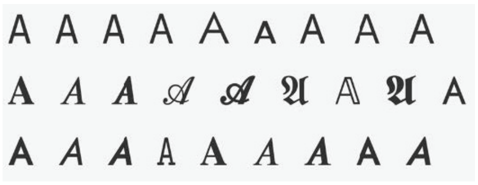


Fig. 2 Homoglyhs of a capital letter "A"

3. SECURITY ISSUES OF HOMOGLYPHS

Homoglyphs can be widely used for various malicious purposes, especially in cyberspace (The MITRE Corporation:2017) (Woodbridge et al.:2018, p.22). For example, consider a PC running on Windows OS. Suppose there is a normal Windows system process called "svchost.exe." An attacker can disguise a malicious process with the name "svch0st.exe." Without attentiveness, it can sometimes be difficult to detect a process with a non-standard or peculiar name that can indicate maliciousness (Woodbridge et al.:2018, p.22). In the case of "svchost.exe" where Cyrillic "O" is used, the possibility of detecting the attacker's process without special tools decreases considerably.

The usage of homoglyphs in this manner can be specified as a spoofing attack (Woodbridge et al.:2018, p.22). A spoofing attack is an attack where an attacker tries to modify some data in such a way that it becomes hard or impossible to identify these modifications (Malwarebytes:2024). The aim of spoofing attacks is usually to gain access to other data or to infect the system itself. There are several types of spoofing attacks (Balaban:2020):

- *ARP Spoofing*: In this kind of attack, an attacker sends an ARP without a request being sent. The idea of the attack is to update ARP tables of devices in such a way that all traffic will go through the attacker's PC.
- *MAC Spoofing*: A type of attack where a hacker changes their own MAC address or masquerades to access a network with access restrictions.

- *IP Spoofing*: Spoofing an IP address to impersonate another computer.
- *DNS Cache Poisoning (DNS Spoofing):* Changing values in the DNS cache to redirect to sites with different domains.
- *Email Spoofing*: A method of disguising malicious emails using the trust given by the address or content of the mail. These emails can use identical or similar domains, sender identifications, and signatures.
- *Website Spoofing*: Creating an exact copy of a site for malicious purposes, which can even be located at a similar address.
- *Caller ID Spoofing*: An attack based on replacing a phone number, ID, or identity to deceive the receiving person.
- *Text Message Spoofing*: A technique of substituting the sender's phone number with letters or numbers to deceive the recipient.
- *Extension Spoofing*: A method of adding or changing file extensions to make them executable.
- *GPS Spoofing*: Sending manually generated or edited GPS signals to the antenna of a victim to affect the navigation device's behavior.
- *Facial Spoofing*: Bypassing face recognition authentication/ identification using a photo or any kind of graphical picture of the victim.

The meaning of homoglyphs allows them to be applied in all types of spoofing attacks (excluding facial spoofing). However, they are not always effective. If the corrupted data from a spoofing attack is checked by a computer, it will at once give a different result, which is understandable. An example of obfuscated words differentiation is given in Figure 3.

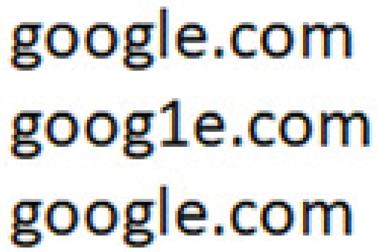

google.com
goog1e.com
gооglе.com

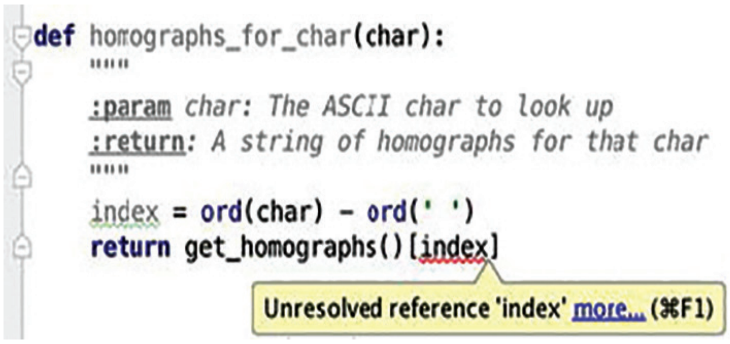
Fig. 3 Example for spoofed with homoglyphs web address. First is normal domain, second – “l” was replaced with “1”, third – English “o” and “e” were replaced with Cyrillic letters.

From the above-mentioned list, we can drop ARP, MAC, and GPS spoofing, but in the case of manual checking without a special device, these attacks can still be dangerous. The rest (from 4 to 9) become a standard place for spoofing attacks. In DNS spoofing, homoglyphs can be used to edit a web address so that the difference between the original DNS record and the edited one is barely noticeable (Unicode inc.:2024).

The same technique can be used in the creation of phishing sites during a website spoofing attack. From the victim's side, it is sometimes impossible to visually distinguish the difference between the original domain name and the spoofed one. This problem is related to the International Domain Name System (IDN) (Opera Team:2017) like depicted on Figure 4. In some browsers, safety and security systems can detect this kind of spoofing. However, the efficiency of these systems is not absolute.

One more example of the usage of homoglyphs in malicious activities is file corruption (McDowell:2011). Configuration files are widely popular in modern software and operating systems. A suitable example is grub.cfg in Linux-based operating systems where the GRUB loader is used, or the file called hosts on machines running on Windows. Both files have human-readable characters and are edited manually. Furthermore, these files are crucial for the corresponding operating systems, and their functionality strongly depends on these files.

Homoglyphs can be easily used to corrupt necessary configuration files and partially or fully cause a computer to malfunction. Fixing these symbols or restoring file data manually can be time-consuming since all homoglyphs (including hidden ones) must be found and replaced with proper symbols, or the whole configuration file must be rewritten. An example of broken code is given in Figure 5.



```
def homographs_for_char(char):
    """
    :param char: The ASCII char to look up
    :return: A string of homographs for that char
    """
    index = ord(char) - ord(' ')
    return get_homographs()[index]
```

Unresolved reference 'index' more... (⌘F1)

Fig. 4 Static analyzer inside IDE recognizes spoofed variable as a new uninitialized variable. (https://habr.com/ru/post/385697)

Another application of homoglyph attacks is the concealment of plagiarism. Homoglyphs can be used to visually preserve the meaning and legibility of words but cause a plagiarism detection system to fail to recognize them as the same word. Hence, two texts that are visually and semantically identical to humans will be recognized as different by a computer (Alvi et al.:2017, p.669).

4. RESEARCH METHODOLOGY

One of the most common ways to detect homoglyphs is to use Levenshtein or Hamming edit distance algorithms and prepare dictionary (Woodbridge et al.:2018, p.22). A word is to be checked by editing distance and compared with a beforehand prepared dictionary. The same approach is used in spell checking algorithms (Lhoussain et al.:2015, p.127).

Levenshtein algorithm–one of the classical examples of Dynamic Programming algorithms (Allison: 1999) used in Computer Science to compute the minimum number of operations required to transform string to string or conversely. The idea of the algorithm is to divide the assumed editing process into three types:

- Insertion of a symbol
- Deletion of a symbol
- Substitution of a symbol

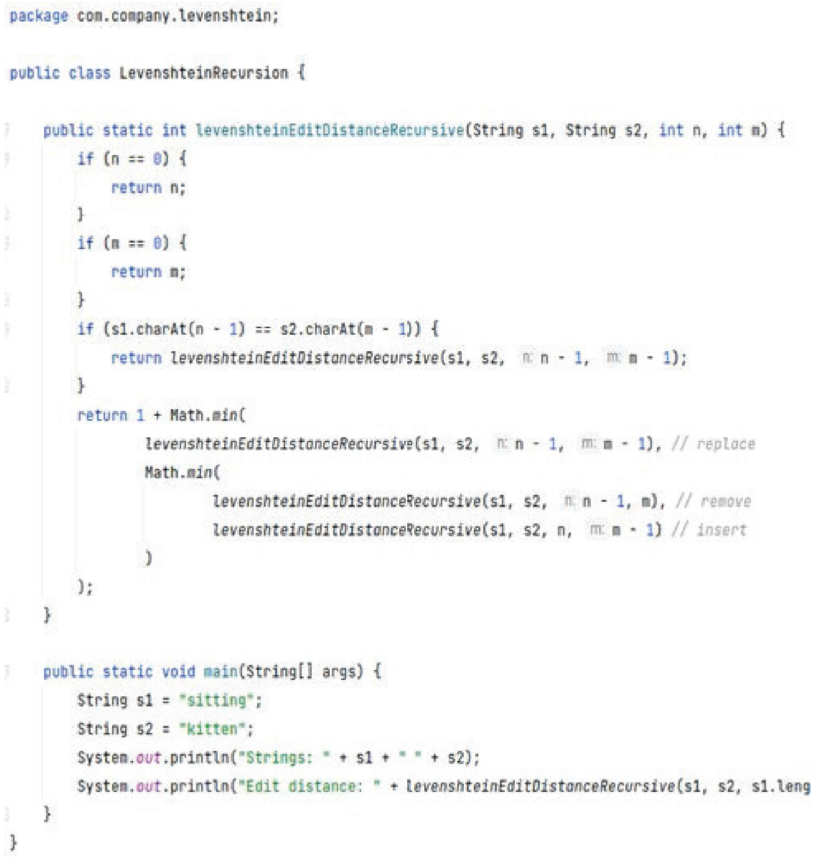
Overall, the recurrent definition of Levenshtein distance function can be presented as on formula 1.

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j), & if\ \min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1, \\ lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & otherwise \end{cases} \quad (1)$$

where *a* and *b* are strings to be compared and *i* and *j* are indices of their characters. $lev_{a,b}$(i,j) stands for the Levenshtein distance between substrings a_0:a_i and b_0:b_j. The base of the recursion is a situation where the string has length equals to 0 and the value of the function in this case will be length of the second string. Otherwise, we just move forward through the string with *i* and *j* being steadily increased. During this movement, the value of the function will calculate on each increase of both indices. Each time the value of function will be calculated by choosing smallest value of the function with smaller *i* and *j* that was calculated before and by adding 1. Adding 1 means that we must change manipulation with a symbol.

Explaining calculations more in more detail, $lev_{a,b}(i\text{-}1, j)$ means the absence of the i^{th} symbol in the string a or deletion of that symbol, $(i,j\text{-}1)$ means absence of j^{th} symbol in b or insertion of symbol b_j to string a. $lev_{a,b}$ $(i\text{-}1,j\text{-}1)$ means that we remove both corresponding symbols from both strings which means that we can add another mutual one. The last step is needed only in case of inequality of *a_i* and *b_j*. It is possible to write Java code like in Figure 5. Also, the ready realization is available in Apache Commons Text package Java (Apache:2020). Memory and time complexity is for both realizations is $O(n^2)$.

In contrast, the Hamming editing distance algorithm uses a simpler approach. In this algorithm, during the iterating over string the number of characters that differ is counted. Hence, the only constraint in the algorithm is that string should have equal sizes for reliable results. Realization is in Figure 6.



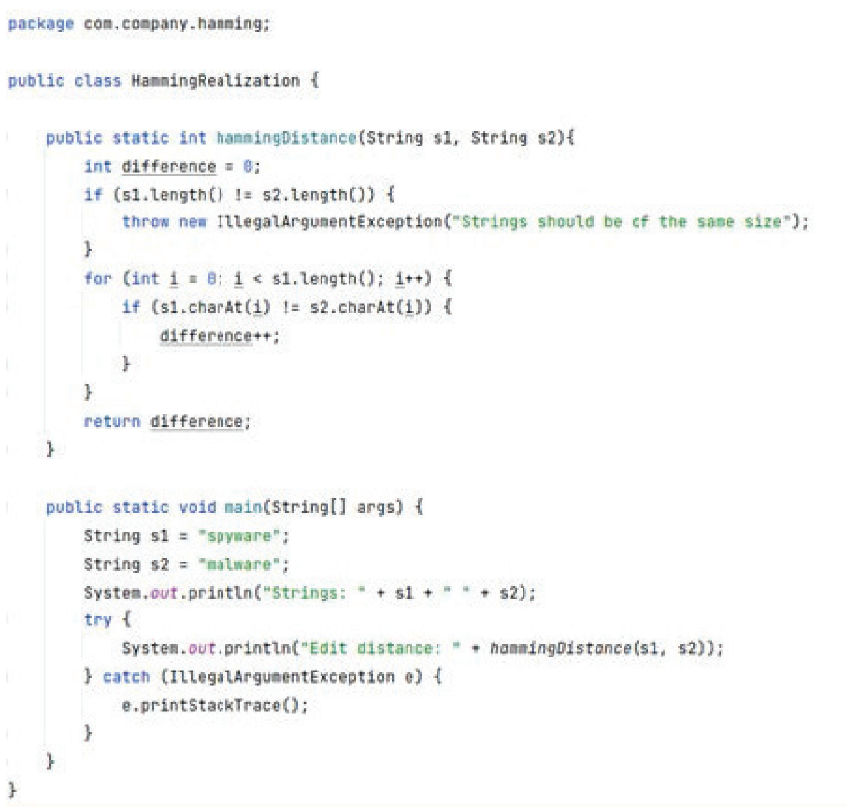
```
package com.company.levenshtein;

public class LevenshteinRecursion {

    public static int levenshteinEditDistanceRecursive(String s1, String s2, int n, int m) {
        if (n == 0) {
            return n;
        }
        if (m == 0) {
            return m;
        }
        if (s1.charAt(n - 1) == s2.charAt(m - 1)) {
            return levenshteinEditDistanceRecursive(s1, s2,  n: n - 1,  m: m - 1);
        }
        return 1 + Math.min(
                levenshteinEditDistanceRecursive(s1, s2,  n: n - 1,  m: m - 1), // replace
                Math.min(
                        levenshteinEditDistanceRecursive(s1, s2,  n: n - 1, m), // remove
                        levenshteinEditDistanceRecursive(s1, s2, n,  m: m - 1) // insert
                )
        );
    }

    public static void main(String[] args) {
        String s1 = "sitting";
        String s2 = "kitten";
        System.out.println("Strings: " + s1 + " " + s2);
        System.out.println("Edit distance: " + levenshteinEditDistanceRecursive(s1, s2, s1.leng
    }
}
```

Fig. 5 Levenstein edit distance recursive realization



```
package com.company.hamming;

public class HammingRealization {

    public static int hammingDistance(String s1, String s2){
        int difference = 0;
        if (s1.length() != s2.length()) {
            throw new IllegalArgumentException("Strings should be of the same size");
        }
        for (int i = 0; i < s1.length(); i++) {
            if (s1.charAt(i) != s2.charAt(i)) {
                difference++;
            }
        }
        return difference;
    }

    public static void main(String[] args) {
        String s1 = "spyware";
        String s2 = "malware";
        System.out.println("Strings: " + s1 + " " + s2);
        try {
            System.out.println("Edit distance: " + hammingDistance(s1, s2));
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 6 Realization of Hamming distance algorithm

Overall, if the distance between the given word and a similar word from the dictionary is greater (or lower) than the predefined precision and contains a character that can be a homoglyph, this word will be counted as spoofed.

As follows from the main idea of the method, the effectiveness of these algorithms is limited by the predefined precision and the existence of the word in the dictionary. This will be the main drawback of the given method. Another demerit of using conventional algorithms is their technical inability to detect all types of phishing at once.

Describing the positive sides, in cases where the word exists in the dictionary and the precision is absolute, the method provides a 100% detection rate. Memory and time complexity is $O(n)$.

Another approach is the usage of Machine Learning to detect words that have homoglyphs. One implementation is to convert a word to a vector image and find the distance between the vector-graphical representation of the given word and those from a dictionary of confusable characters or Siamese

Networks (Woodbridge et al.:2018, p.22). Considering the disadvantages of the Machine Learning method itself, such as computing performance requirements (Flair:2021) and the ability to poison the data, some implementations of this approach can achieve a higher true-positive rate than conventional algorithms (improvement from 13% to 45%).

Another point that should be mentioned is that a possible solution should not only aim to detect homoglyphs but also to recover files after an attack. Since the recovery process is a problem in homoglyph attacks, this fact should be considered. Detection methods should also consider finding spoofing in specific terminology or abbreviations of a company or enterprise.

The possible comparison using matrix table can be represented as in Table 1.

5. SUGGESTED FRAMEWORK

The main problem of homoglyph detection lies in the inability to predict which word a homoglyph can indicate a spoofed word, or if it is used due to the spelling of a particular word in a language. Fortunately, getting the list of homoglyphs is not a significant problem since detecting pairs of similar symbols is easier because the number of symbols in an encoding system is limited and smaller than that of words in a natural language. Thus, the main weak point of any algorithm can be related to a database of words or data needed to detect a spoofed word. The key function of the most effective solution will be the ability to intensively enrich our own database while considering all possible contexts. Another feature that can be implemented is the recovery of obfuscated words using

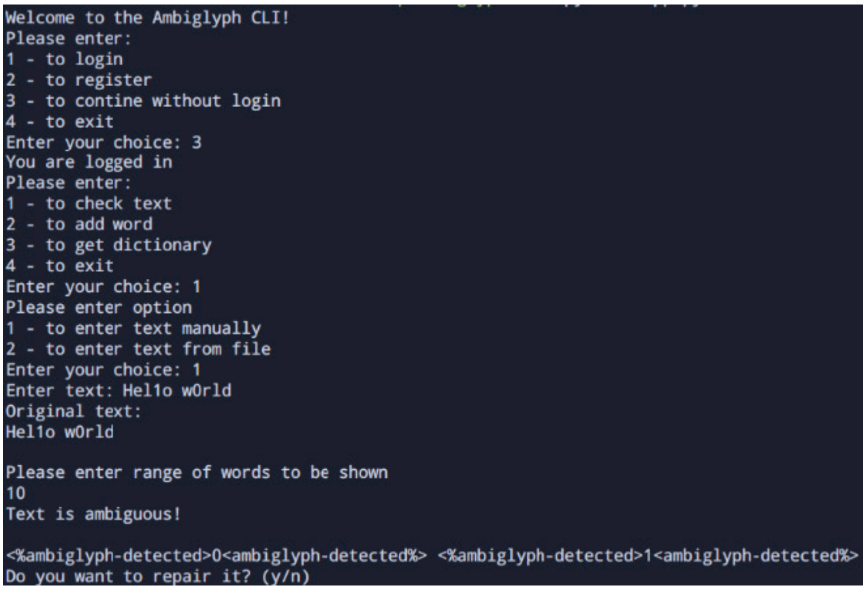
Table 1

Features	Levenshtein edit distance	Hamming edit distance	Machine Learning approach
Detecting words with different lengths	+	-	+
100% accuracy	+		
Less computing power needed	+		

any word-guessing approach.

As one implementation of a possible solution, special software using the Levenshtein algorithm and linear search can be created. A detached backend server will play the role of the main actor in detecting obfuscated text and attempting recovery, with a client-side model in use. The backend server will store all necessary data, such as a database of homoglyphs and a dictionary of words. When the server receives a request with text, all words within it will be checked using the Levenshtein distance to determine similarity with words in the dictionary in case a homoglyph exists. This algorithm has proven to be the most sophisticated in this field in terms of accuracy and time complexity (Hardesty:2015). Thus, it will be used. If words are detected, then linear search takes its turn. Potential words according to the Levenshtein edit distance will be found in the dictionary and given as suggestions to the user for replacement. Of course, the number of suggestions can be significant depending on the precision defined in the Levenshtein algorithm (maximal edit distance between given and suggested words). Since the proper edit distance to be chosen can vary depending on the homoglyph used in spoofing a word, the user must supply boundaries that specify the maximum number of suggestions to be given.

The next problem solved by the solution should be the inclusion of specific words strictly related to the context, for example, of a company or an organization. The main dictionary



```
Welcome to the Ambiglyph CLI!
Please enter:
1 - to login
2 - to register
3 - to contine without login
4 - to exit
Enter your choice: 3
You are logged in
Please enter:
1 - to check text
2 - to add word
3 - to get dictionary
4 - to exit
Enter your choice: 1
Please enter option
1 - to enter text manually
2 - to enter text from file
Enter your choice: 1
Enter text: Hel1o w0rld
Original text:
Hel1o w0rld

Please enter range of words to be shown
10
Text is ambiguous!

<%ambiglyph-detected>0<ambiglyph-detected%> <%ambiglyph-detected>1<ambiglyph-detected%>
Do you want to repair it? (y/n)
```

Fig. 7 Ambiglyph CLI

can be enlarged with words specific to the company. Of course, some of these words can be considered potentially dangerous information if published; therefore, an accounting system is to be introduced. All users will be able to log in using their credentials and use the enlarged dictionary. They can add or remove words to their own data storage.

Talking about client applications, which will do all main interactions with the user. It was decided to use Command Line Interface (CLI) as it will be more comfortable and convenient to use with configuration files or plain text files. An interface is shown in Figure 7.

The integration between CLI application and backend application is done by REST API technology (Figure 8). The server side is fully written in Java programming language using Spring technology stack. Ready realization of Levenshtein edit distance algorithm is taken from Apache Commons Text package as a more versatile one. As a database server the MySQL database is used. CLI application is fully written in Python programming language. This application is compatible with macOS and Linux-based operation systems.

The name "Ambiglyph" is given to the software. Both server (Ambiglyph Server) and client (Ambiglyph CLI) applications are available on GitHub platform.

With regards to the client-server architecture, the responsibilities of both sides are separated. The server side will only process incoming text and find suggestions. Also, it can manage Create, read, update and drop (CRUD) operations on word and users by playing a role of authentication server and segregating privileges. Spring security module and authentication is managed by JSON Web Tokens (JWT-tokens).

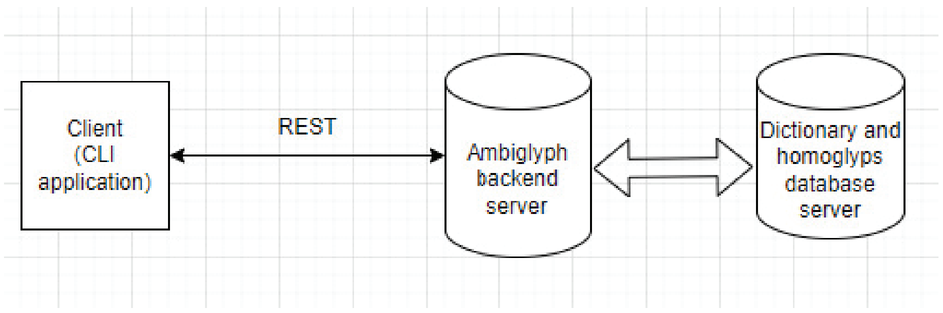
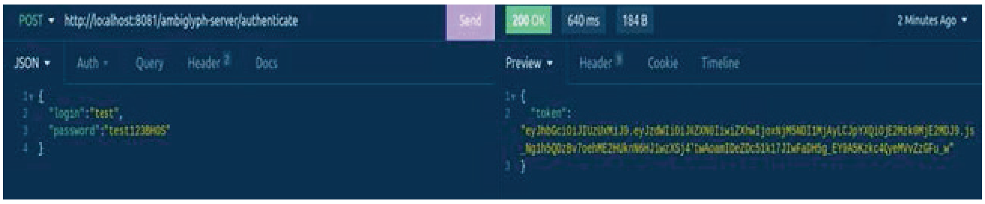


Fig. 8 Client-server structure



http://localhost:8081/ambiglyph-server/authenticate

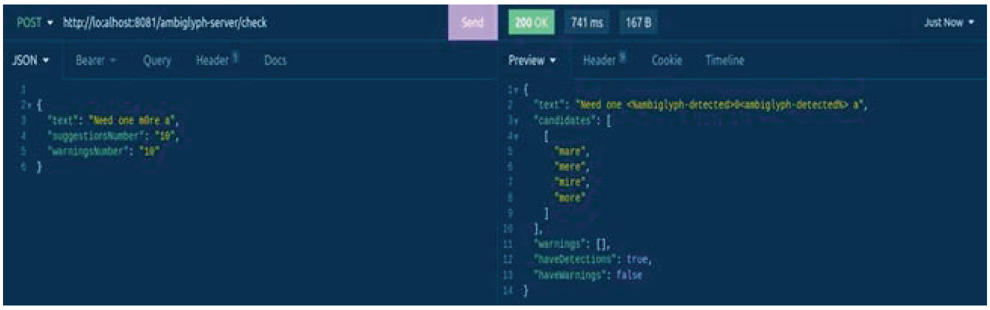
```
{
  "login":"test",
  "password":"test123BH0S"
}
```

```
{
  "token":
  "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJ0ZXN0IiwiZXhwIjoxNjM5NDI1MjAyLCJpYXQiOjE2Mzk0MjE2MDJ9.js_Ng1h5QDzBv7oehME2HUkrN6HJ1wzXSj4'twAoamIDeZDc51k17JIwFaDH5g_EY0A5Kzkc4QyeMVvZzGFu_w"
}
```

Fig. 9 Retrieval of JWT-token after successful login

The current main REST API features can be depicted like this. Firstly, it is needed for users to have an account in the system. In test example, user with login "test" will be used. It must be authorized for future actions; hence users must obtain JWT-token from the server (Figure 9). Then, the user can send necessary requests (Figure 10).

However, it is not possible to detect spoofed words if it is used only by one company or organization (Figure 11).

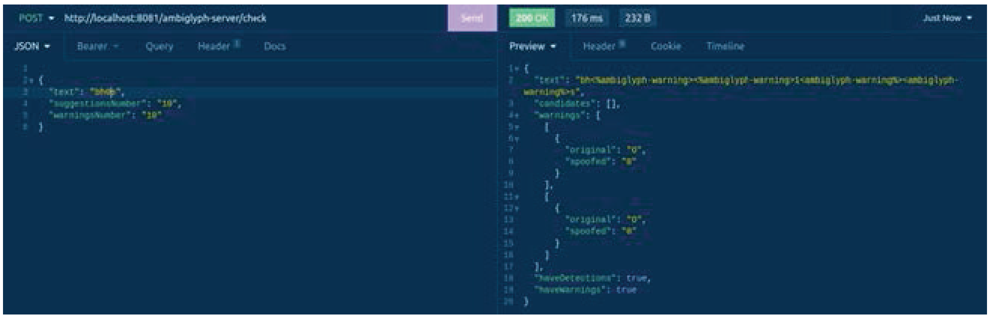


http://localhost:8081/ambiglyph-server/check

```
{
  "text": "Need one m0re a",
  "suggestionsNumber": "10",
  "warningsNumber": "10"
}
```

```
{
  "text": "Need one <%ambiglyph-detected>0<ambiglyph-detected%> a",
  "candidates": [
    [
      "mare",
      "mere",
      "mire",
      "more"
    ]
  ],
  "warnings": [],
  "haveDetections": true,
  "haveWarnings": false
}
```

Fig. 10 Obfuscated word "more" ("m0re") is detected, and suggestions are provided.

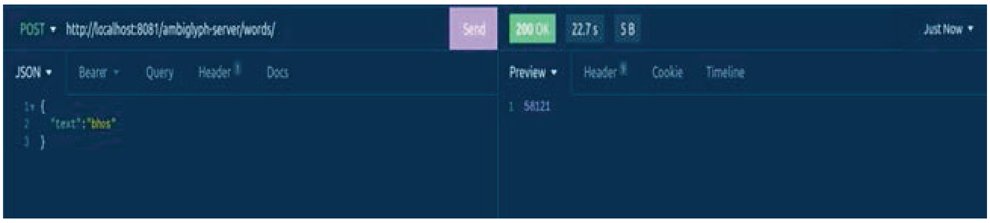


http://localhost:8081/ambiglyph-server/check

```
{
  "text": "bh0s",
  "suggestionsNumber": "10",
  "warningsNumber": "10"
}
```

```
{
  "text": "bh<%ambiglyph-warning><%ambiglyph-warning>1<ambiglyph-warning%><ambiglyph-warning%>s",
  "candidates": [],
  "warnings": [
    [
      {
        "original": "O",
        "spoofed": "0"
      }
    ],
    [
      {
        "original": "O",
        "spoofed": "0"
      }
    ]
  ],
  "haveDetections": true,
  "haveWarnings": true
}
```

Fig. 11 Word "bh0s" is recognized as warning, no suggestions are provided.

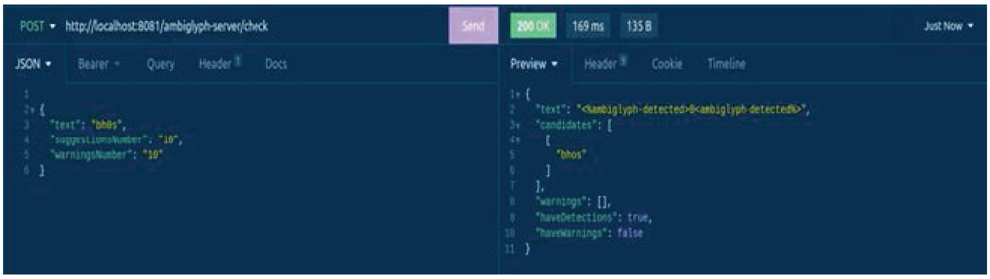


http://localhost:8081/ambiglyph-server/words/

```
{
  "text":"bhos"
}
```

```
58121
```

Fig. 12 Adding word “bhos” to the user's words database.



http://localhost:8081/ambiglyph-server/check

```

{
  "text": "bh0s",
  "suggestionsNumber": "10",
  "warningsNumber": "10"
}
```

```
{
  "text": "<%ambiglyph-detected>0<ambiglyph-detected%>",
  "candidates": [
    [
      "bhos"
    ]
  ],
  "warnings": [],
  "haveDetections": true,
  "haveWarnings": false
}
```

Fig. 13 Word “bh0s” recognized as obfuscated word “bhos.”

By default, Ambiglyph uses an open database of words (Corncob Lowercase Dictionary) that consists of about 58,000 commonly used words. Of course, there is no word “bhos” or spoofed version “bh0s”. But, by using homoglyph database (Homoglyphs) it is still possible to send a warning that the symbol “0” can be for obfuscation. Anyway, it is still possible to add the word “bhos” to evaluate user’s database (Figure 12).

Now, the obfuscation can be detected (Figure 13).

By contrast, the client side is only responsible for reading files, chunking them and sending them to the server. After the server responds, the client tries to restore a text according to the suggestions provided by the server and choice of user. Also, it can add unfamiliar words to the users’ dictionary or remove them. CLI ensures versatileness of the application usage, especially when graphic user interface (GUI) is not accessible.

6. CONCLUSION

Overall, modern encoding systems are quite rich in terms of the symbols they can stand for; however, this leads to the indistinguishability of some similar characters, known as homoglyphs. This vulnerability is actively exploited by hackers to perform several types of spoofing attacks. The problem needs to be addressed using various techniques, such as classical algorithms or

Machine Learning approaches. Nevertheless, it is still impossible to guarantee absolute detection and recovery of texts corrupted by homoglyphs. This issue is related to the imperfections of current algorithms and challenges in creating and supporting sufficient databases. Recovery techniques also suffer from these problems.

The solution of guessing obfuscated words using the Levenshtein edit distance, which is commonly used in spell checking, and dividing dictionary databases into users who can extend them with their own terminologies and words used in the context of their organization or enterprise, can be considered a successful attempt to solve the problem of detecting homoglyph attacks. Additionally, linear search on given suggestions can also be efficient for recovery after homoglyph attacks.

However, there are still negatives that interfere with making the solution sophisticated. The first is the requirement to always be connected to a database server, which is sometimes not possible, for example, due to the absence of an internet connection. Another negative is the human factor related to the choice of words to be replaced by the server suggestions. Users can mistakenly replace a spoofed word with an inappropriate option. The same issue arises with updating a user's database. Furthermore, a lack of specific words used by a user can hinder the process of proper word guessing by the server, resulting in the inability to recover a word. Lastly, the Command Line Interface of the client application may not be suitable or user-friendly for some users.

REFERENCES

[1] Allison, L., Dynamic Programming Algorithm (DPA) for Edit-Distance, 1999. Available: https://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/Edit.

[2] Ambiglyph CLI, Ambiglyph CLI, Available: https://github.com/IZOBRETATEL777/ambiglyph-cli.

[3] Ambiglyph Server, Ambiglyph Server, Available: https://github.com/IZOBRETATEL777/ambiglyph-server.

[4] Apache, Commons Text, 2020. Available: https://commons.apache.org/proper/commons-text/.

[5] Balaban, D., 11 Types of Spoofing Attacks Every Security Professional Should Know About, March 24, 2020. Available: https://www.securitymagazine.com/articles/91980-types-of-spoofing-attacks-every-security-professional-should-know-about.

[6] Corncob Lowercase Dictionary,

corncob_lowercase.txt, Available: http://www.mieliestronk.com/corncob_lowercase.txt.

[7] Flair, Data, Advantages and disadvantages of machine learning language, 2021.

[8] Homoglyphs, Homoglyphs, Available: http://homoglyphs.net/.

[9] Lhoussain, Aouragh Si, Gueddah, Hicham, and YOUSFI, Abdellah, Adaptating the levenshtein distance to contextual spelling correction, *International Journal of Computer Science and Applications*, Vol. 12, No. 1, pp. 127-133, 2015.

[10] Malwarebytes, What is a spoofing attack?, Available: https://www.malwarebytes.com/spoofing.

[11] McDowell, M., Understanding Hidden Threats: Corrupted Software Files, March 9, 2011 (updated in 2019). Available: https://us-cert.cisa.gov/ncas/tips/ST06-006.

[12] Cox, L.A., What's Wrong with Risk Matrices? In: *Risk Analysis,* Vol. 28, No. 2, 2008.

[13] Flouris, G.T., Lock D., (2009) *Managing Aviation Projects from Concept to Completion,* Ashgate Publishing Company, Farnham, pp. 304-306.