Rapid Homoglyph Prediction and Detection through Visual HitZone Mapping and Comparison

By
Avi Ginsberg

Advisor
Dr. Cui Yu

A thesis submitted in fulfillment of the requirements of the degree of

Master of Science
in
Computer Science

Department of Computer Science and Software Engineering
Monmouth University
West Long Branch, New Jersey
April 2017

Abstract

Character comparison is far from a new concept. For decades, computer science has worked on Optical Character Recognition and handwriting recognition techniques, which are used in a myriad of applications. Much progress has been made in this field, especially with the more contemporary advancements in artificial intelligence. In these cases, we are trying to compare given input to a known set of characters; that is, we want to map image representation of a character to the text it represents. What if we want to do nearly the opposite? What if we want to take a known character and find unknown characters that visually resemble it? This concept has many use cases, most notably in computer/internet security and copyright infringement detection. However, minimal research has been conducted on this topic until now.

Enter: Visual HitZone Mapping and Comparison. This thesis aims to provide a collection of algorithms to enable quick and efficient comparison of glyphs and prediction of homoglyphs with adjustable levels of granularity and percentages of similarity. Such a collection is widely versatile as it can be easily fit to a variety of use-cases right “out-of-the-box”. The proposed technique of Visual HitZones simulates the human behavior of “visual scanning" and allows significant amounts of visual data to be represented in a format that is easily and quickly searchable. Ultimately, the technique allows the algorithms to effortlessly scale and perform accurately on any set of glyphs from any language. Additionally, the Visual HitZones concept employs a pre-processing time-memory trade-off to vastly decrease search and comparison times.

Acknowledgements

I would like to thank my advisor, Dr. Cui Yu, for her guidance in writing this thesis. Her feedback and support has been incredibly helpful.

I would like to thank my family for their love, support, and encouragement throughout the writing of this thesis and for their support of all of my educational goals.

I would like to thank Timothy Gamache for all of his support, encouragement, and insight. I am incredibly grateful for all of his help proofreading. I would like to explicitly thank him for his help with the math used to justify my decision to use static mapper arrays for mapping of pixel number to HitZone map zone during the HitZone map generation process and for his help in rendering the visual representations of this math (Figure 3.13 and 3.14).

I would like to thank Christopher Sladky for his assistance with verifying the computational complexity of the algorithms proposed in this thesis.

I would like to thank all of my friends for their encouragement and support of my educational goals.

Table of Contents

List of Figures

List of Tables

1. Introduction

This chapter provides background information necessary for conceptual understanding of this thesis. It starts by introducing key concepts and progresses to discuss the problem this thesis aims to solve as well as some currently used inadequate solutions to this problem. Finally, the motivation behind the thesis is explained and the approach and aims are detailed.

1.1 Background

Table 1.1: Key Terms and Definitions

Term	Definition
Glyph	A symbol used in written communication. It is used as a synonym for character in this thesis, but it should be noted that glyphs can also be pictographs such as a snowman, sun, or smiley face.
Homoglyph	A glyph that looks the same as, or similar to, another glyph.
Character Code	A unique numeric code issued by the Unicode Consortium that corresponds to a specific glyph concept. It is not tied to a specific rendering of the glyph, as this is dependent on font. Character Codes are usually written as hexadecimal numbers.
Unicode Consortium	An international non-profit organization working to standardize computer character sets.
Font	A collection of data for visually rendering glyphs and their corresponding character codes. Usually, all glyphs within a font are tailored to a certain artistic style. Commonly known examples of fonts include “Arial” and “Times New Roman.”
Granularity	A measure of level of detail.

1.1.1 Unicode and Glyph Basics

The Unicode Consortium is an international organization dedicated to "developing, maintaining, and promoting software internationalization standards and data, particularly the Unicode Standard, which specifies the representation of text in all modern software products and standards." [1] Unicode has created unique character codes for all standardized glyphs and supports the majority of the worlds' languages. The current version of the Unicode Standard, Unicode 9.0, provides character codes for 128,172 glyphs[2] and supports over 900 languages and script sets.[3] Because The Unicode Consortium is internationally recognized as the standardizing body for glyphs and the Unicode Standard is utilized by the majority of contemporary computer systems and software, the standard character codes issued by Unicode will be used in this thesis.

It is important to note that character codes issued by Unicode are not tied to a single visual representation of a glyph. They can be said to represent the concept behind the glyph, rather than the glyph itself. From a high-level simplified viewpoint, a computer stores a string of text (string of glyphs) as a series Unicode character codes. When the computer operating system or a piece of software wants to display the text, it looks in the system font library for the font specified by the software or operating system. Within the font, it finds the visual data needed to render each character code into a glyph. Then, it renders the glyphs.

Theoretically, all fonts should display a specific glyph similarly enough that the human reading the text produced by that font's rendering is able to understand the concept behind the glyph. Occasionally, the artistic liberties taken by font designers make it difficult for a human to recognize specific letters or symbols. However, outlandish font

choice is of little importance for this thesis. Fortunately, most typical computer users will not attempt to override their system fonts. Therefore, we are concerned primarily with the relatively few standard fonts that are found on most contemporary computer systems, most of which are reasonably legible.

All fonts have visual differences in their rendering of glyphs. While most popular fonts aim for legibility, due to the similarities in the shapes of various letters and symbols there may be visual similarities between characters in certain fonts. This is font-specific. For example, " I " and " l " (capital letter I and lower case letter L) look very different when rendered in the font "Times New Roman." However, " I " and " l " (capital letter I and lower case letter L) look nearly identical when rendered in the font "Arial." Such visual ambiguities can have unexpected consequences, which we will explore in our discussion of homoglyphs.

1.1.2 Homoglyphs

A homoglyph is a glyph that looks the same, or nearly the same, as another glyph, but is actually a different glyph and is represented by a different character code. Homoglyphs can arise from different situations. The two most common situations are: 1. Two glyphs from the same language appear visually similar in certain fonts due to similar structures of the character/symbol (as described in the previous section in the case of a capital I and lowercase L appearing to be identical in the Arial font) and 2. Two glyphs from different languages have visually similar renderings. This is exemplified by examining a list of homoglyphs for the "Latin" capital letter A (character code 0x41), which is the A used in English (as well as Spanish, French, and Italian):

Homoglyph Character	Character Code	Description
А	0x410	CYRILLIC CAPITAL LETTER A
Α	0x391	GREEK CAPITAL LETTER ALPHA
Ά	0x386	GREEK CAPITAL LETTER ALPHA WITH TONOS
Ά	0x1fbb	GREEK CAPITAL LETTER ALPHA WITH OXIA
Ὰ	0x1fba	GREEK CAPITAL LETTER ALPHA WITH VARIA
Ạ	0x1ea0	LATIN CAPITAL LETTER A WITH DOT BELOW

Figure 1.1: Homoglyphs of the "Latin" Letter Capital A (Character Code 0x41), Rendered in Times New Roman Font.

Clearly, in the Times New Roman font, the "Cyrillic Capital Letter A" and the "Greek Capital Letter Alpha" both appear identical to a Latin Capital letter A. The remaining 4 homoglyphs in the list look very similar, but have additional markings. When attention is called to the markings, they are very apparent. But, on a small screen such as a smartphone screen, a screen with lower resolution, or even in print, these marks are easy to miss. To an untrained eye, these marks may simply look like dirt on the screen, be mistaken for a quotation mark, or be missed altogether.

1.2 The Problem (and Inadequate Solutions)

The existence of homoglyphs creates an interesting problem. Because homoglyphs look similar, if not identical, but have different character codes, they can be used to construct look-alike strings of text. Nefarious individuals can use these look-alike strings of text to craft a myriad of deception-based attacks. For example, consider an attacker who wants to create a fake PayPal mobile application designed to steal a user's credentials (and later their money). Assuming a PayPal app already exists for the given platform, submitting an app to the platform's app store with the name "PayPal" would

likely be flagged and rejected, as the name is already taken. However, if the attacker replaces the first or second letter “P” (“Latin Capital Letter P” - character code 0x50) with a “P” (“Cyrillic Capital Letter Er” - character code 0x420), the app name still appears identical to PayPal, but may not be flagged as a duplicate app and ultimately may be published in the platform’s app store and used to steal user data.

Such look-alike string attacks are increasingly common. They affect website domain names, app stores, and almost any computerized system in which users are able to submit content. Because these homograph-based attacks are so common and have such great power to deceive, the Unicode Consortium has dedicated several sections of their latest technical report on security considerations to this issue.[4]

Unfortunately, the solutions suggested by Unicode amount to little more than “use extreme caution when working with characters from mixed language sets.”[4] Ultimately, this type of approach is implemented as a set of restrictions which attempt to prevent nefarious individuals from crafting homoglyph-based attacks at the expense of restricting characters available to legitimate users.

A highly visible example of character restrictions is the way the Google Chrome web browser displays characters in domain names. Chrome uses a combination of whitelists and an algorithm based on the Unicode Technical Standard #39 “Moderately Restrictive” character filtering recommendation set to allow only certain combinations of characters to be displayed in website domain name. Any combination of glyphs that are not approved per this policy, which includes many symbols and domain names containing characters from multiple different languages, will result in the entire domain

name being displayed in a code format, known as Punycode, instead of the intended glyphs.[5,6] For example, in the Chrome URL bar, the domain name "i♥icecream.com" would be displayed as "xn--iicecream-0g3f.com". This specific case does little to improve security and prevents a potentially great domain name from being used.

Even more restrictive examples are commonplace. Some domain name registries, such as the registry for Macedonia's country code domain .mk, allow only English letters, numbers, and the hyphen character.[8] This type of policy is successful at preventing many homoglyph URL attacks, but it is severely limiting and decidedly unfriendly to non-English-speaking users.

Clearly, restrictive approaches are inadequate solutions for preventing homoglyph based attacks in our contemporary globalized society. The presence of internet-enabled devices continues to grow around the world, as does our overall reliance on the internet and computerized technology. Users in different countries should be allowed to use their native languages in computerized systems and everyone should be able to use globally recognized symbols such a heart (♥). Instead of restricting the usage of glyphs, I propose we intelligently detect homoglyphs and only disallow true threats. Why punish the legitimate user when we can detect and thwart the attacker instead?

1.3 Motivation and Aims

This thesis was motivated by the aforementioned problems associated with homoglyph characters. It aims to:

1. Propose a system of algorithms for predicting homoglyphs that is fast, efficient, and scalable.
2. Propose a method for calculating visual similarity between glyphs.
3. Provide test results and analysis demonstrating the validity of the above.
4. Explore potential use cases in different sectors.
5. Discuss algorithm implementation considerations that may affect performance.

1.4 Approach

When a human looks at two glyphs side by side in an attempt to determine if they are different, they are likely to scan over the first glyph with their eyes noting the overall shape and features. They will then compare their memory of the first glyph to the overall shape and features of the second glyph. If the glyphs are vastly different, this first pass comparison should suffice, and they will conclude the two characters are indeed different. If the characters are very similar or the same, they are likely to re-examine each character more than once, each time attempting to compare more fine details until they finally conclude a character is indeed visually the same or different. The approach taken in this thesis is inspired by, and in some ways attempts to simulate, this type of human visual comparison behavior.

A high-level view of the approach is as follows. First, we extract the list of glyphs supported by a given font. For each glyph, we create four HitZone maps, one with 16 zones, one with 64 zones, one with 100 zones, and one with 256 zones.

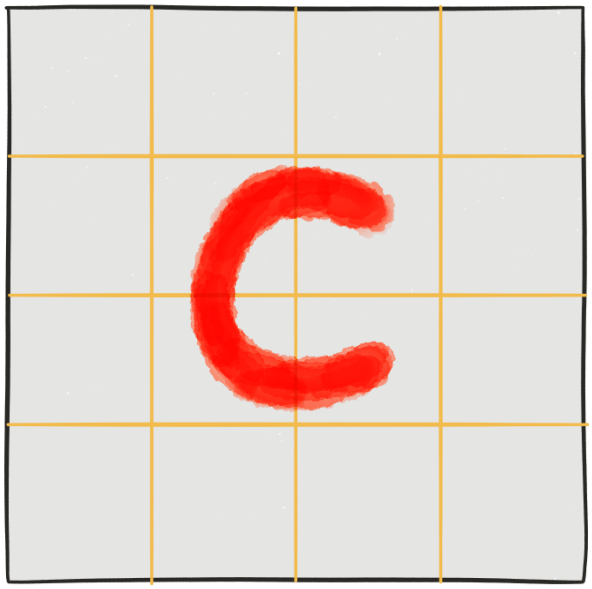

Figure 1.2: Visual Representation of a 16 Zone HitZone Map of the Letter C

To create a HitZone Map, we center the glyph and then divide the glyph into 16, 64, 100, or 256 even zones. Figure 1.2 depicts what a 16 zone HitZone map of the letter C would look like. We then assign a number to each zone, as depicted in Figure 1.3.

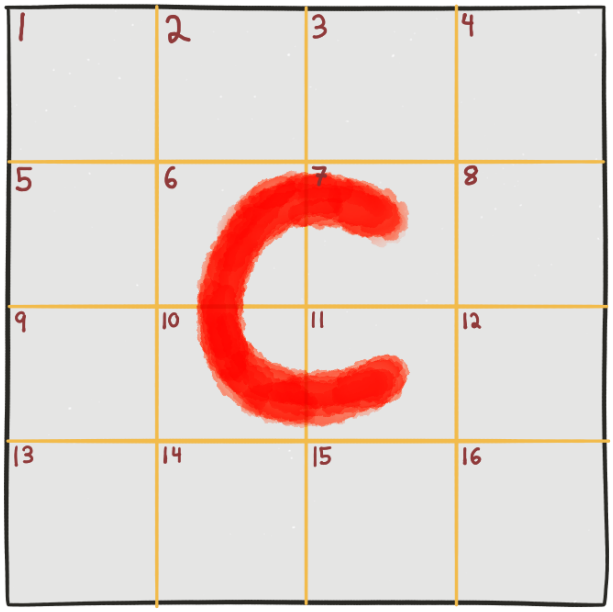

Figure 1.3: Visual Representation of Numbering a 16 Zone HitZone Map of the Letter C

Any zone that is not empty is recorded as a “hit.” Any zone that is empty is recorded as a “miss.”

By creating four HitZone maps, each with an increasing number of smaller zones, we establish multiple levels of granularity, or precision, which are useful in the comparison and prediction algorithms. For each character, we store the four HitZone maps, the character code, and the font’s name in a database.

In order to calculate the similarity percentage of two glyphs for a given font at a given level of granularity, we retrieve the associated HitZone maps for both of the glyphs. We when compare the number and location of hits to determine a percentage of similarity.

In order to predict homoglyphs of a certain similarity percentage for a given glyph, we retrieve one or more HitZone maps for the glyph and then use the “hits” to retrieve from the database all other glyphs with the same base hit profile. Finally, we use the similarity percentage calculation to filter out glyphs that are not greater than or equal to the specified percentage of similarity.

2. Literature Review

2.1 Optical Character Recognition

Any discussion bordering on the topic of character/glyph similarities would be incomplete without at least examining its relationship to, and any overlap with, Optical Character Recognition (OCR) technologies. Very early on in this thesis, it was theorized that OCR techniques could be used to detect homoglyphs. The idea was to abuse weaknesses in common OCR techniques, that is, to input glyphs into OCR algorithms and see which glyphs were incorrectly recognized as other glyphs. It was quickly decided this approach was flawed from multiple angles, namely: 1. it would not help us to predict multiple homoglyphs for a given glyph, and 2. it was essentially a test of the accuracy of a given OCR algorithm and did little to empirically and concretely measure glyph similarity. Hence, this idea was dispensed with.

During the early phases of research, various OCR algorithms and techniques were explored. There have been many advances in OCR due to the advances in Artificial Intelligence. Neural networks are able to recognize typed and hand-written glyphs with excellent accuracy.[9] The k-Nearest Neighbor algorithm also yields excellent results.[10,11] There are two major problems with attempting to apply the aforementioned techniques/algorithms to the homoglyph problem. First, these methods are designed to be increasingly accurate. In predicting homoglyphs, we desire measurable inaccuracy. We want to be able to define a specific amount of inaccuracy, or a "tolerance range," and find all visually similar glyphs within that range. Second, these methods do nearly the opposite of what we are attempting to do. OCR is designed to map unknown glyphs given

as user input to a pre-defined set of glyphs it is trained to recognize. Instead, we want to map a known glyph to any number of unknown glyphs that are similar in appearance.

2.2 Homoglyph Attacks

Homoglyph attacks, or attacks in which look-alike characters are in some way used to trick someone, are incredibly powerful and dangerous. While they receive the most attention in the field of web security, they are by no means limited to this field. For example, a homoglyph attack could be used in academia to bypass plagiarism detection software: an unscrupulous student could obtain an assignment, paper, or other document, and replace several of the characters with homoglyphs before submitting it as his own. To demonstrate this concept, the following example plagiarized paragraph was taken from http://smallseotools.com/plagiarism-checker.

> The road of yellow brick is an element in the novel The Wonderful Wizard of Oz by L. Frank Baum, with additional such roads appearing in The Marvelous Land of Oz and The Patchwork Girl of Oz. The 1939 film The Wizard of Oz, based on the novel, gave it the name by which it is better known, the Yellow Brick Road (it is never referenced by that title in the original novel). In the later film The Wiz, Dorothy has to find the road, as the house was not deposited directly in front of it; in the novel and the 1939 film, Dorothy's house is placed directly in front of the road.

The characters “T”, “h”, “e”, and “o” were replaced with homoglyphs, resulting in the following paragraph. This new paragraph looks identical to the old paragraph:

> The road of yellow brick is an element in the novel The Wonderful Wizard of Oz by L. Frank Baum, with additional such roads appearing in The Marvelous Land of Oz and The Patchwork Girl of Oz. The 1939 film The Wizard of Oz, based on the novel, gave it the name by which it is better known, the Yellow Brick Road (it is never referenced by that title in the original novel). In the later film The Wiz, Dorothy has to find the road, as the house was not deposited directly in front of it; in the novel and the 1939 film, Dorothy's house is placed directly in front of the road.

The original paragraph was submitted to three different plagiarism detection websites: https://www.grammarly.com/plagiarism-checker,

http://smallseotools.com/plagiarism-checker, and http://www.quetext.com. All three websites detected plagiarism:

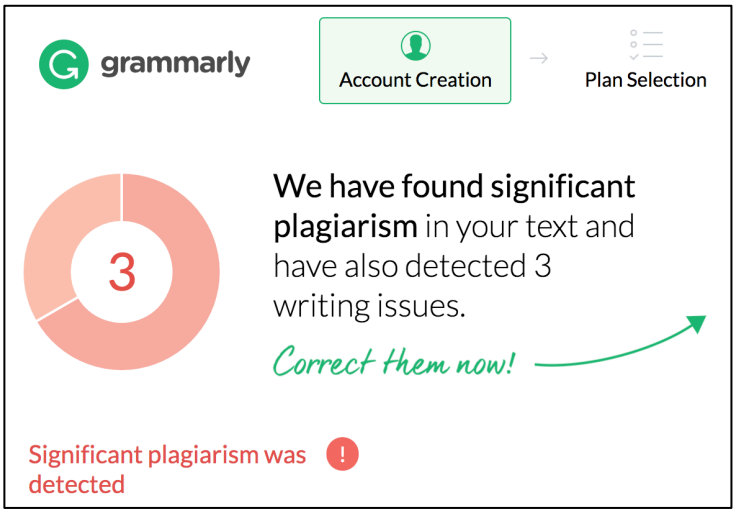

Figure 2.1: Plagiarism Detected by Grammarly

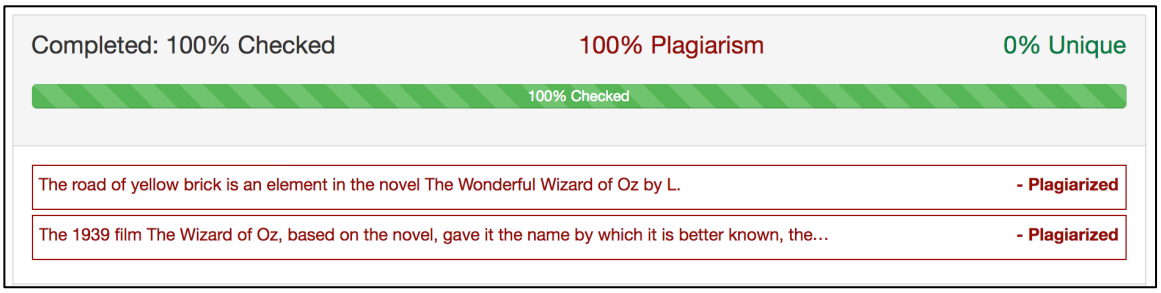

Figure 2.2: Plagiarism Detected by SmallSEOTools

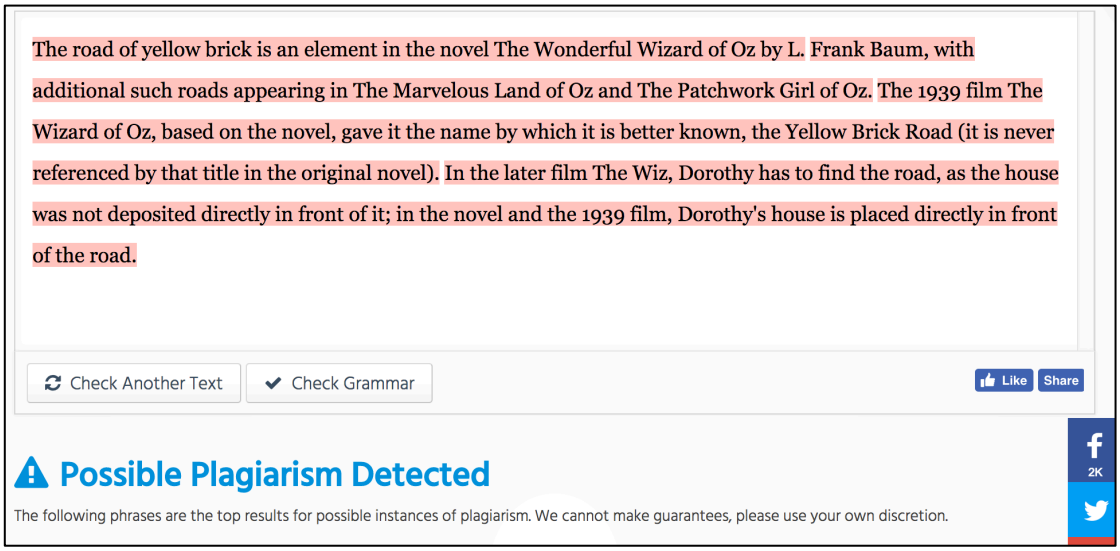

Figure 2.3: Plagiarism Detected by Quetext

Afterwards, the paragraph containing the homoglyph characters was submitted to the same three plagiarism detection websites. All three sites completely failed to detect plagiarism.

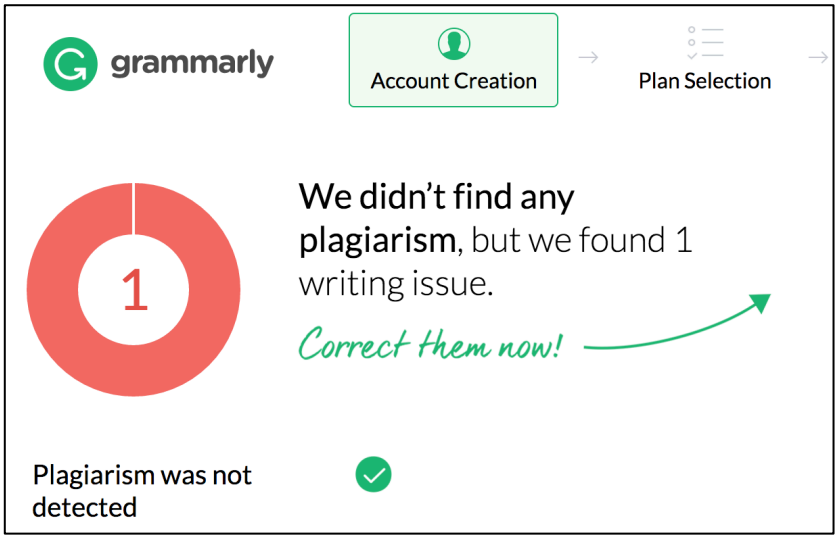

Figure 2.4: Plagiarism Detection Failure – Grammarly

Figure 2.5: Plagiarism Detection Failure – SmallSEOTools

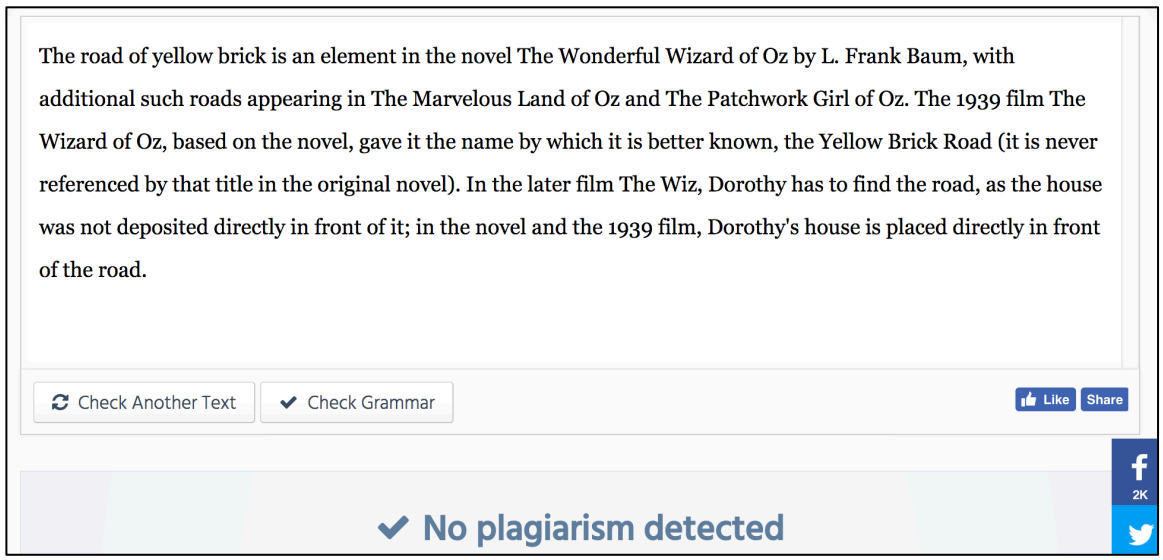

Figure 2.6: Plagiarism Detection Failure – Quetext

Clearly, homoglyph attacks are a significant problem for fields other than internet security. However, within the field of internet security, homoglyph attacks pose a grave danger: they can be employed in various types of phishing scams to steal sensitive information or to gain access to protected resources and information. As previously discussed, Unicode is acutely aware of such attacks along with their implications, and has made a series of recommendations on restrictive policies designed to prevent such attacks.[4]

Hewlett-Packard apparently agreed with the view stated in this thesis that the restrictive approach is suboptimal. They applied for, and in 2014 were granted, a patent

for a Homoglyph Detection System for website domain names.[12] This patent proposes to use a system in which a questionable website address is run through a glyph-matching module to generate possible similar looking website addresses. These addresses are then checked with a DNS query to determine if the questionable site could be an attempt to trick users looking for a real site. The patent fails to describe the exact inner-workings of the glyph-matching module – simply stating that one should be used and suggesting it may use OCR methods.[12] The system of algorithms proposed by this thesis could work well for glyph-matching in this type of detection system.

In addition to large companies and standard setting bodies, computer security researchers are also concerned with homoglyph-related attacks. In 2012, veteran internet security researcher and blogger Adrian Crenshaw released a highly comprehensive article in which he explores the various ways in which homoglyphs can be used to construct phishing attacks, with an emphasis on email and social media.[13] Phishing attacks are obviously a major concern for the field of internet security. However, a fringe issue that is brushed up against by Mr. Crenshaw's article is potentially just as harmful: filter evasion. The implications of this are not explored in his article, but one can imagine a myriad of situations in which the ability to circumvent text filters would be problematic. For example, in an online game designed for children, one may be able to use homoglyphs to bypass chat filters, allowing them to say inappropriate things. Alternatively, homoglyphs may be used to bypass message filters in online auction sites or market places, enabling criminals to provide alternative contact information in an attempt to move a transaction outside of the site and ultimately scam legitimate buyers or sellers.

Homoglyph-based attacks are clearly dangerous, significant, and have wide reaching consequences. They affect many different fields in as many ways as motivated attackers can dream up. As our global reliance on technology increases, the situation stands to worsen. As such, it is incredibly advantageous to be able to detect and thwart this type of abuse.

2.3 Homoglyph Detection

Detection of homoglyphs is a difficult problem. How does one claim that two glyphs are visually similar? Moreover, how can a computer make this determination? A 2011 paper about finding homoglyphs from the Department of Electrical and Computer Engineering at the University of Alberta suggests that glyph similarity is not an absolute characteristic. Rather, it is a spectrum that depends largely on the human perceiving the two glyphs.[14] This thesis largely adopts this viewpoint.

The Alberta paper also proposes a method for calculating similarity between glyphs. It proposes the use of compressed and standardized values based on Kolmogorov Complexity theory, which claims the distance between two objects is the difference between the lengths of the shortest programs that can produce those objects on a Turing machine.[14] The experimental results demonstrated by this method indicate that it is able to generate numeric values that roughly approximate a general measure of similarity between two glyphs. This method does not directly compare structure of glyphs. While these similarity measures are a step in the direction of accurate detection, similarity that ignores base structure of glyphs leaves much to be desired. The approach taken in this

thesis recognizes this shortcoming and attempts to rectify it by proposing a method of calculating similarity that is much simpler than Kolmogorov Complexity and takes visual character structure into account.

3. Algorithms and Techniques

The overall goal of the algorithms and techniques presented in this thesis is to develop a system in which homoglyphs can be accurately detected and predicted. Scalability and versatility were chief concerns throughout the research and development process. The discussion is broken into two sections for each algorithm/technique: theory and implementation. The theory section aims to provide insight into development and inner-workings, while the implementation section provides pseudocode and notes on the specific implementation in the test software developed for this thesis. Python 2.7.8 and MySQL 5.6.34 were used for implementation.

3.1 Visual HitZone Mapping

3.1(a) Theory

In order to develop a system in which large numbers of glyphs can be compared quickly, a time-memory tradeoff was employed. It was decided that the best approach was to pre-process, encode, and store each glyph in a database in a way that made comparison of thousands of glyphs based on visual structure simple and fast. This was accomplished by creating "Visual HitZone Maps."

The model of a Visual HitZone Map was arrived at by introspection. When visually examining two glyphs to check for similarity, a human is apt to first glance at each glyph, quickly looking at different areas, or zones, of each glyph to get a "big picture view." They will continue to look at each glyph in greater detail, that is, paying attention to smaller and smaller areas, or zones, on each glyph, until they make a decision about the similarity. At a high level, the algorithms in this thesis attempt to mimic this

behavior. In order for this type of comparison to be performed, glyphs must be broken up into multiple areas, or zones, that can be compared. Accordingly, we arrive at the concept of a HitZone map.

The HitZone map model proposes the creation of four different maps for each glyph. These maps are identified by their "level," which reflects their amount of granularity, or precision. A map with a higher granularity level represents the glyph to a greater degree of precision. A level 1 HitZone map should consist of 16 zones (4 rows and 4 columns). A level 2 HitZone map should consist of 64 zones (8 rows and 8 columns). A level 3 HitZone map should consist of 100 zones (10 rows and 10 columns). A level 4 HitZone map should consist of 256 zones (16 rows and 16 columns). This is depicted in Figures 3.1 – 3.4, in which the letter A has been broken up in to zones according two the 4 different levels of granularity.

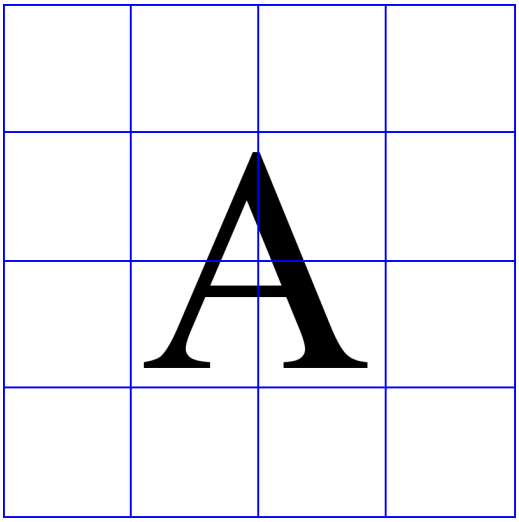

Figure 3.1: Zones for HitZone Map of Letter A at Granularity Level 1 (4x4 Zones)

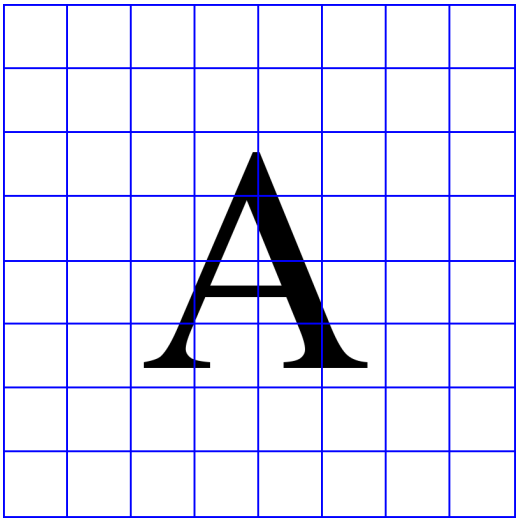

Figure 3.2: Zones for HitZone Map of Letter A at Granularity Level 2 (8x8 Zones)

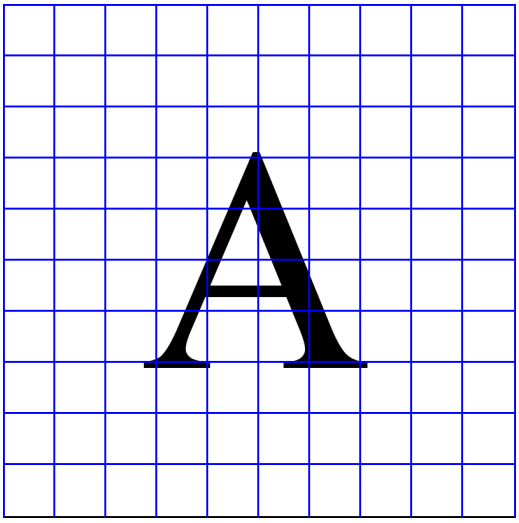

Figure 3.3: Zones for HitZone Map of Letter A at Granularity Level 3 (10x10 Zones)

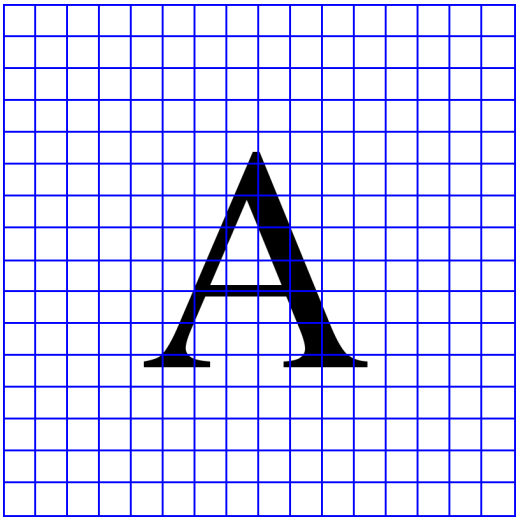

Figure 3.4: Zones for HitZone Map of Letter A at Granularity Level 4 (16x16 Zones)

All maps are of the same size and contain the same number of pixels. The increase in number of zones between maps is due to smaller zone size. Each zone in each HitZone map stores either a “hit” or a “miss.” In order to calculate these values, a visual representation of each glyph is rendered. The visual representation is divided into zones as prescribed above. Any zone that is completely white is recorded as a miss. Any zone that is not completely white is recorded as a hit. In this manner, we derive maps with

different levels of granularity that approximate the basic structure of the glyph. A visual representation of this concept can be seen in Figures 3.5-3.8. The color green is used to represent hits and the color red is used to represent misses. This design allows the glyph to later be compared against other glyphs in a manner similar to the behavior of a human performing the comparison.

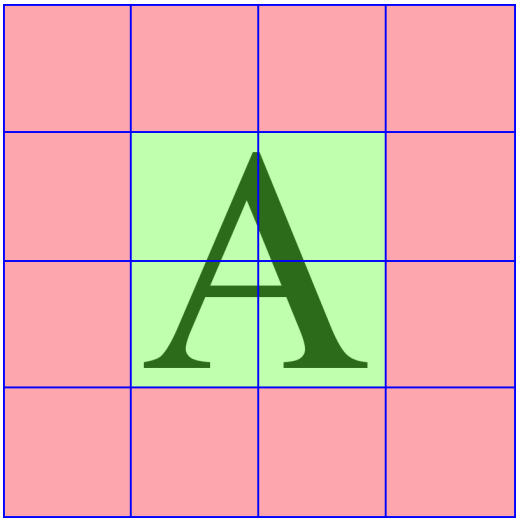

Figure 3.5: Depiction of "Hits" and "Misses" on a Level 1 HitZone Map for Letter A

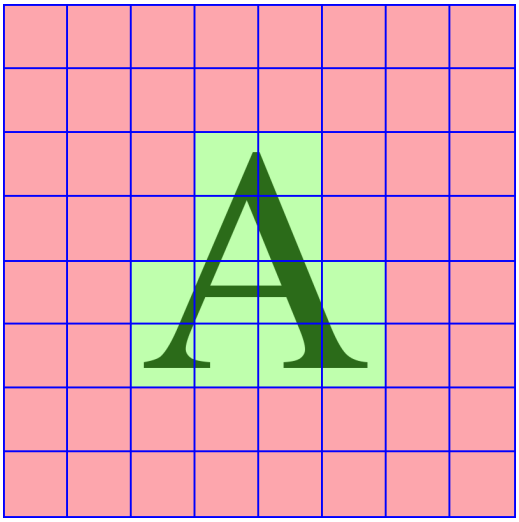

Figure 3.6: Depiction of "Hits" and "Misses" on a Level 2 HitZone Map for Letter A

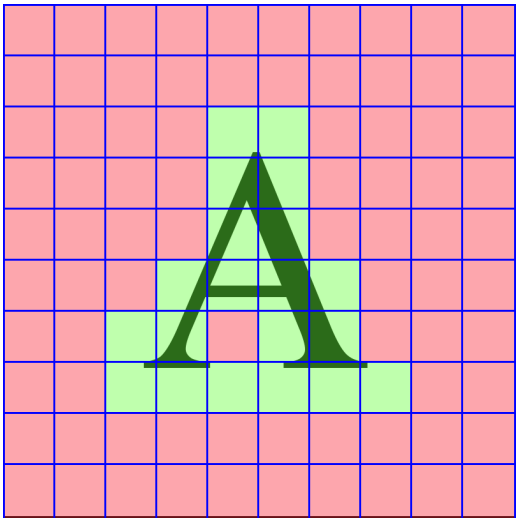

Figure 3.7: Depiction of "Hits" and "Misses" on a Level 3 HitZone Map for Letter A

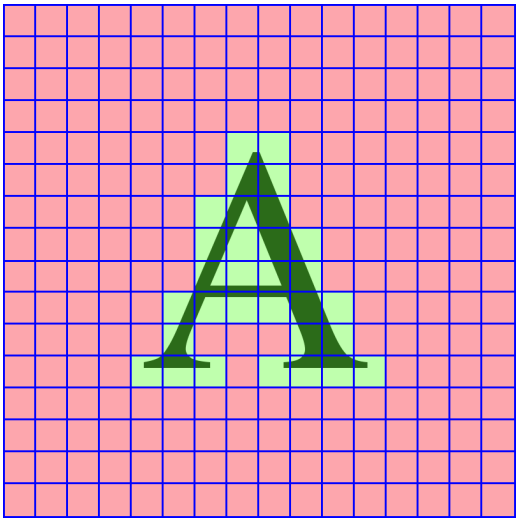

Figure 3.8: Depiction of "Hits" and "Misses" on a Level 4 HitZone Map for Letter A

In order to represent HitZone maps efficiently, an array of zeroes and ones should be used*. Each zone on the HitZone map should be given an identifying index number. The top left-most zone should be assigned the index number zero. The row should be traversed from left to right and each zone should be assigned a number one larger than the zone to its left. When the end of the row is reached, the next number should be assigned to the left-most zone of the row below it. A visual representation of this zone index numbering for a level 1 HitZone map can be seen in Figure 3.9.

*Note: If storage space is very scarce, it should be possible to represent HitZone map data as an inverted index in which only index numbers of zones with hits are recorded. The comparison and prediction algorithms would need to be adjusted slightly to support this storage format. For most fonts, this should provide a high level of compression (> 50% space savings) with a mild performance impact for comparison and search.

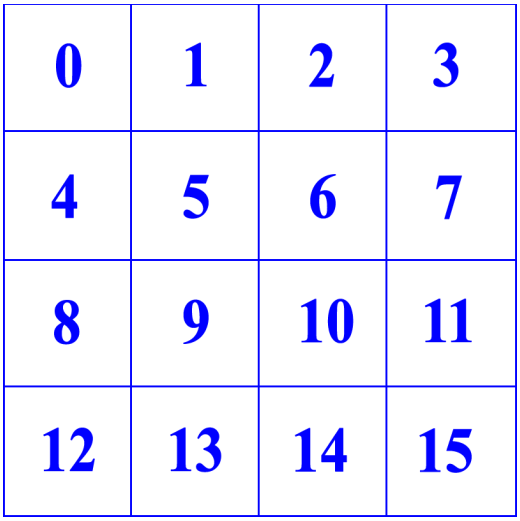

Figure 3.9: Visual Depiction of Index Numbering for a Level 1 HitZone Map

For each zone that contains a "hit," store a 1 in the array block with the corresponding index number. For each zone that contains a "miss," store a zero in the array block with the corresponding index number. This can be seen in Figures 3.10, 3.11, and 3.12 below.

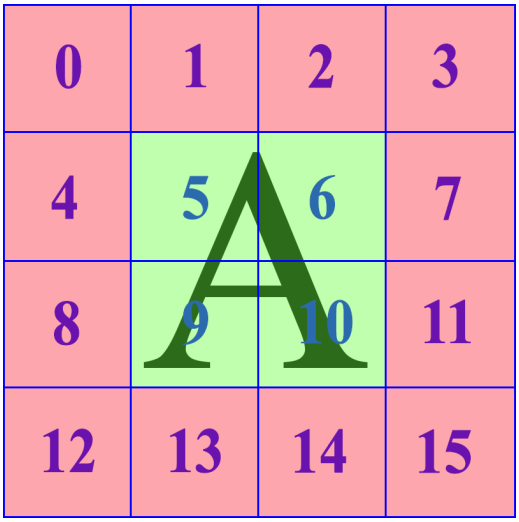

Figure 3.10: Letter A with Overlay of Visual Depiction of Index Numbering for a Level 1 HitZone Map

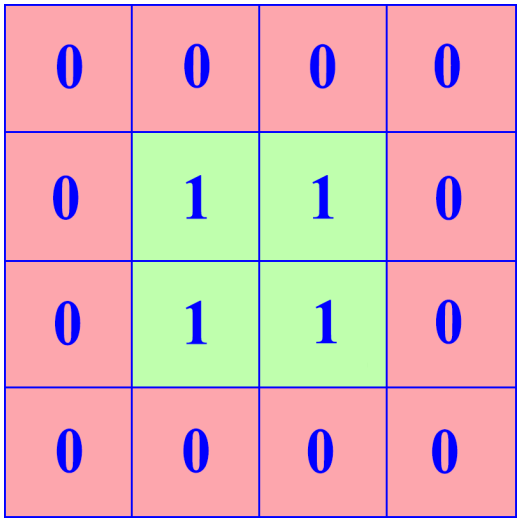

Figure 3.11: Visual Representation of "Hits" and "Misses" from Figure 3.10 Represented as 1s and 0s

Array Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value:	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0

Figure 3.12: Visual Representation of Array Storing HitZone Map Data from Figure 3.11

3.1(b) Implementation

The process of encoding HitZone maps and storing them into our database is as follows. First, a database it set up with four tables, one for each level. Each table should have a column for CharacterCode, a column for Font, and a column for each of the zones in the HitZone map at that level. For example, the table for Level 1 would have a "CharacterCode" column, a "Font" column, and 16 columns for the 16 zones in the level 1 HitZone map. The table should be indexed on CharacterCode.

After the database is created, we must gather the fonts we want to use to populate the database. For each font, a table of the characters that the font supports should be extracted. If you are using OpenType fonts on a Unix based system, this can be done using the TTX tool, which is a command line tool for converting from OpenType fonts to

XML. The command *ttx -t cmap <font_name>.ttf* will output an XML file with a "character table," which is a table of all the glyphs contained in the font.

Following the character table extraction, we render each glyph onto a canvas of 80 pixels by 80 pixels. It is unnecessary to save the image data to a file. We want to generate an array with 6400 blocks, with each block corresponding to a pixel of the rendered glyph, with the top left-most pixel at array index 0 and the pixels being traversed from left to right and rows being traversed top to bottom. Each array block should contain the RBG value of the corresponding pixel. This can be done in python using the PIL library's *draw.text* function and a font size of 50, along with the *getdata* function. It is important to render the glyph in the center of the canvas so that it is not truncated at an edge of the canvas. To calculate the (x,y) coordinate needed to render the glyph in the center of the canvas, the following formulae can be used:

$$Center\ x = \frac{Canvas\ Width\ in\ Pixels - Glyph\ Width\ in\ Pixels}{2}$$

$$Center\ y = \frac{Canvas\ Height\ in\ Pixels - Glyph\ Height\ in\ Pixels}{2}$$

Canvas Width and Height should both be 80 pixels. This size canvas is sufficiently large to provide enough resolution for glyph comparison without unnecessarily wasting storage space.

After the glyph has been rendered in the center, we need to calculate its "true size" and re-render it so it is truly centered on the canvas. The phenomenon is due to white space that is "baked" into the visual data of the glyph in the font. In order to obtain the true size of the glyph, iterate through the 6400 block array of visual data that was just generated for the glyph. This iteration can be accomplished with a pair of nested for loops,

each loop counting from 0 to 79, and a counter, *pos*, that increments by 1 on each iteration of the inside for loop. We also want to create a four variables: one to store the smallest x coordinate where there is a non-white pixel (*smallest_x*), one to store the smallest y coordinate where there is a non-white pixel (*smallest_y*), one to store the largest x coordinate where there is a non-white pixel (*largest_x*), and one to store the largest y coordinate where there is a non-white pixel (*largest_y*). The values of *smallest_x* and *smallest_y* should be initialized to a number larger than 6400. The values of *largest_x* and *largest_y* should be initialized to 0. The counter of the outer for loop represents the y coordinates of pixels in the image and the counter of the inner for loop represents the x coordinates of the pixels in the image. The tertiary counter, *pos*, represents the index of the pixel as stored in the image data array. Each pixel should be checked for RBG values of (255, 255, 255). If the values are anything else, we have detected a non-white pixel. When we detect a non-white pixel, we compare the current x and y coordinates to the variables that hold the smallest x and y coordinates and the largest x and y coordinates. If the current value of x is smaller than the value stored in *smallest_x*, we replace the value with our current value of x. Similarly, if the current value of x is larger than the value stored in *largest_x*, we replace the value with our current value of x. The same comparisons and replacements are done for our y variables. In this manner, we are iterating through each pixel of the image and finding the top-most, bottom most, left-most, and right-most pixels that are non-white. Hence, at the termination of this nested for loop, we have variables with values representing the true top, bottom, left, and right of the glyph. *smallest_x* can be subtracted from *largest_x* and *smallest_y* can be subtracted from *largest_y* to arrive at the glyph's true width and height,

in pixels, respectively. It should be noted at the end of the nested for loop, if the values of *smallest_x*, *smallest_y*, *largest_x*, and *largest_y* are unchanged from their initial values, we have detected an "invisible character." These are present in many font sets and are used for typesetting and various other purposes. When an invisible character is detected, it is safe to discard the image data and move on to processing the next character.

After we have calculated the true width and height of the glyph, we can calculate x and y offsets to be used in conjunction with the original glyph size to find the true coordinates needed to center the glyph on the canvas. The formulae for calculating offsets and the true center x and y are as follows:

$$x\ offset = Original\ Glyph\ Width\ in\ Pixels - Real\ Glyph\ Width\ in\ Pixels$$

$$y\ offset = Original\ Glyph\ Height\ in\ Pixels - Real\ Glyph\ Height\ in\ Pixels$$

$$True\ Center\ x = \frac{Canvas\ Width\ in\ Pixels - Original\ Glyph\ Width\ in\ Pixels - x\ offset}{2}$$

$$True\ Center\ y = \frac{Canvas\ Height\ in\ Pixels - Original\ Glyph\ Height\ in\ Pixels - y\ offset}{2}$$

The "True Center x" is actually the point on the x-axis of the canvas that we want to begin rendering the glyph image data so that the glyph appears truly centered. The "True Center y" is the same, but for the y-axis. Using the true center x and y coordinates we re-render the glyph in the center of the canvas.

Once we have re-rendered the glyph, it is time to calculate the "hits" and "misses" for each level HitZone map. In order to determine which zones have "hits," we must first establish a way to associate pixels and zones. As previously described, we represent image data as a 6400 block array of pixels starting with the top left-most pixel at index 0 and then traversing pixels from left to right and rows from top to bottom. When we

overlay the indexed zones from our HitZone maps and look for the relationship between zone number and pixel number, we arrive at a highly complex periodically discontinuous piecewise function. For example, for a level 1 HitZone map (16 zones), the function would be discontinuous every 20 pixels. Figure 3.13 shows what this function would look like if we zoom in on zones 1-4. Figure 3.14 shows what the function would look like across all 16 zones. This function gets increasingly complex as granularity level increases.

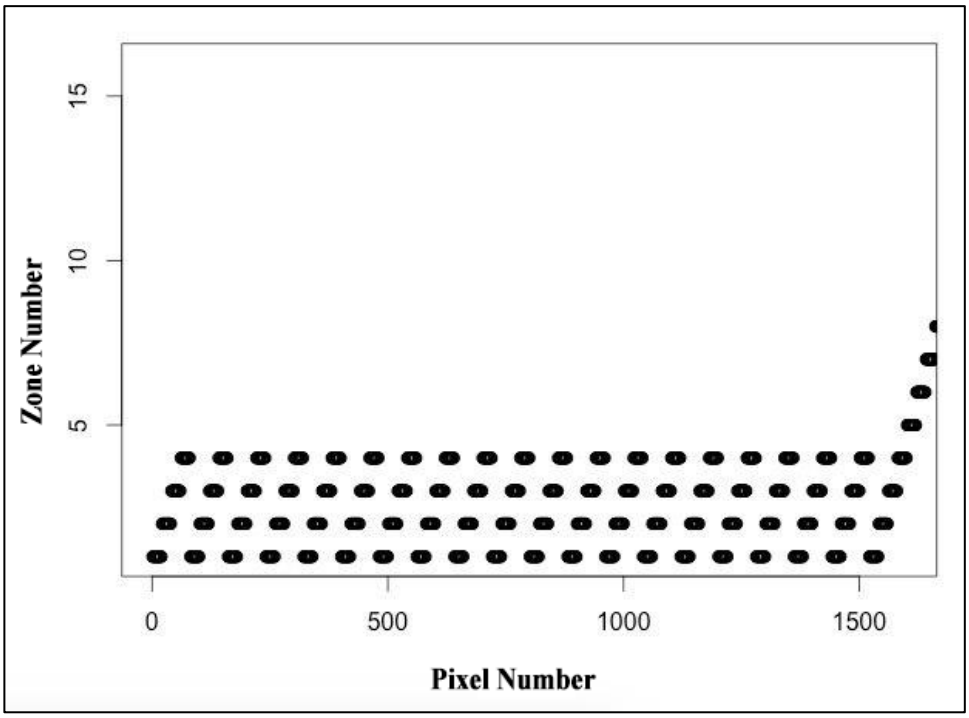

Figure 3.13: Visual Representation of Piecewise Function for Mapping Pixel Number to Zone Number at Granularity Level 1 (Zoomed in on Zones 1-4)

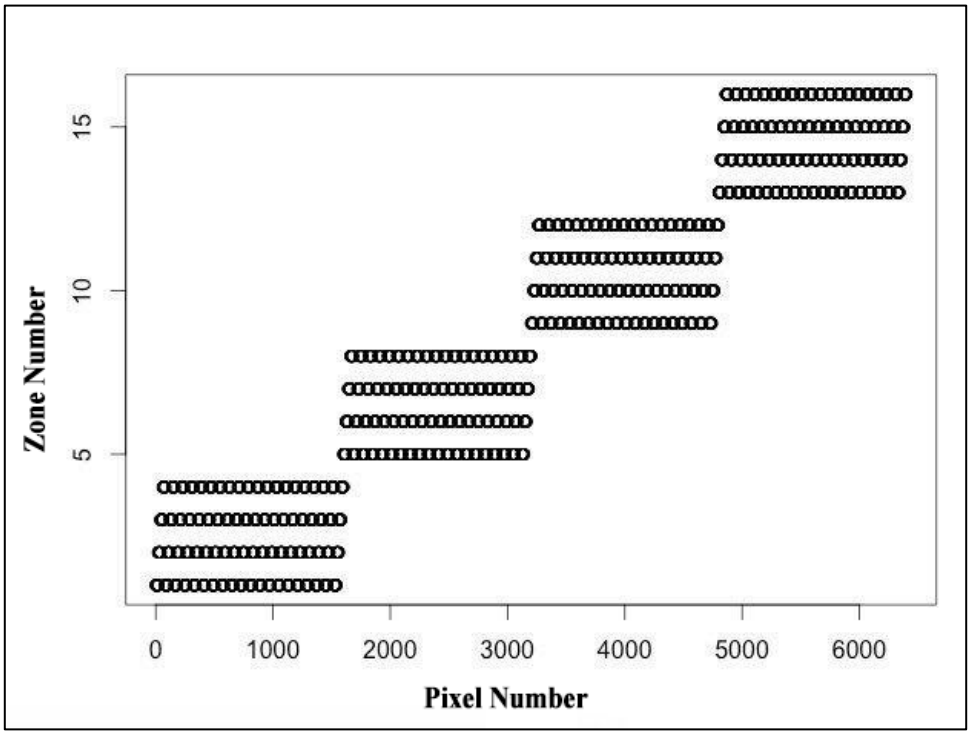

Figure 3.14: Visual Representation of Piecewise Function for Mapping Pixel Number to Zone Number at Granularity Level 1 (Showing all 16 Zones)

Due to the computational complexity of mapping pixel number to zone number it did not make sense to create a mapping function. Instead, static 6400 block arrays were generated for each HitZone map level, which map image data array pixel number to HitZone map zone number. The array index corresponds to the pixel number and the value corresponds to the zone number. These arrays were serialized and are loaded at runtime. They are referenced to map pixel number to zone number when a hit is detected, as will be described next. In order to build these arrays the following algorithm was developed:

Algorithm 1 GeneratePixelZoneMapperArray

Input: $NumZones$

1: $SqrtNumZones \Leftarrow \sqrt{NumZones}$
2: $PixelsPerZoneRow \Leftarrow 80/SqrtNumZones$
3: $PixelsPerZoneColumn \Leftarrow PixelsPerZoneRow$
4: $zone \Leftarrow 1$
5: $zbase \Leftarrow 0$
6: $pixel \Leftarrow 0$
7: CREATE Array $PxMap$ of size 6400
8: **for** $row = 0$ to 79 **do**
9: **for** $z = 1$ to $SqrtNumZones + 1$ **do**
10: **for** $p = 0$ to $PixelsPerZoneRow$ **do**
11: $currentpx \Leftarrow zbase + z$
12: $PxMap[pixel] \Leftarrow currentpx$
13: INCREMENT $pixel$
14: **if** $(row + 1)\%PixelsPerZoneColumn = 0$ **then**
15: $zbase = zbase + SqrtNumZones$
16: **end if**
17: **end for**
18: **end for**
19: **end for**
20: **return** $PxMap$

Figure 3.15: Pseudocode for Algorithm to Generate Array that Maps Pixel Number to Zone Number

The algorithm presented in Figure 3.15 should be used to generate mapper arrays for granularity levels 1-4, providing 16, 64, 100, and 256 as input for *NumZones*, respectively. Once this has been done, we iterate through each glyph in the character table for each font. For each glyph in each character table, we render the truly centered version of the glyph as previously described. We then create four arrays of lengths 16, 64, 100, and 256 to store HitZone maps at the 4 different granularity levels. We pre-populate all blocks in these 4 arrays with the value 0. We then iterate through each pixel in the image data array for the glyph, looking for "hits" by detecting non-white pixels in the same manner as described in the discussion about rendering the glyph in the center of the canvas. Each time a hit is detected, we reference the mapper-arrays to find the zone number associated with that pixel for each level of granularity. We then set the

corresponding block in each of our 4 HitZone map storage arrays to a value of 1. Once we have calculated all four HitZone maps for the glyph, we store the HitZone map values into our database along with the character code for the glyph and the current font. We then move on to the next glyph. The pseudocode for this process for a single font is as follows:

Algorithm 2 GenerateHitZoneMapsStoreDB

Input: $FontCharMap$, $FontData$, $FontName$
Input: $PixelZoneMapperArrayLevel1$, $PixelZoneMapperArrayLevel2$
Input: $PixelZoneMapperArrayLevel3$, $PixelZoneMapperArrayLevel4$

```
Array CC ⇐ EXTRACT CharCodes from FontCharMap
for i = 0 to count(CC) do
    CurrentGlyph ⇐ CC[i]
    img ⇐ New Image Canvas of size 80x80 pixels
    Set font renderer to use font FontName, FontData, font size 50
    glyph_pixel_size_x ⇐ Get_Size_x(CurrentGlyph)
    glyph_pixel_size_y ⇐ Get_Size_y(CurrentGlyph)
    glyph_center_x ⇐ (80 − glyph_pixel_size_x)/2
    glyph_center_x ⇐ (80 − glyph_pixel_size_y)/2
    On img at (glyph_center_x, glyph_center_x) render CurrentGlyph
    pixel_list = Get_Image_Data(img)
    pos ⇐ largest_x ⇐ largest_y ⇐ 0
    smallest_x ⇐ smallest_y ⇐ 6500
    for y = 0 to 79 do
      for x = 0 to 79 do
        if pixel_list[pos] != Color White then
          if x < smallest_x then
            smallest_x ⇐ x
          end if
          if y < smallest_y then
            smallest_y ⇐ y
          end if
          if x > largest_x then
            largest_x ⇐ x
          end if
          if y > largest_y then
            largest_y ⇐ y
          end if
        end if
        INCREMENT pos
      end for
    end for
    if smallest_x = 6500 || smallest_y = 6500 || largest_x = 0 || largest_y = 0 then
      Skip to next CC[i] in FOR LOOP
    end if
    real_width ⇐ largest_x − smallest_x
    real_height ⇐ largest_y − smallest_y
    x_offset ⇐ glyph_pixel_size_x − real_width
    y_offset ⇐ glyph_pixel_size_x − real_width
    true_glyph_center_x ⇐ (80 − glyph_pixel_size_x − x_offset)/2
    true_glyph_center_y ⇐ (80 − glyph_pixel_size_y − y_offset)/2
    On img at (true_glyph_center_x, true_glyph_center_x) render CurrentGlyph
```

```
43:    CREATE Array Level1Hits of size 16, Initialize all values to 0
44:    CREATE Array Level2Hits of size 64, Initialize all values to 0
45:    CREATE Array Level3Hits of size 100, Initialize all values to 0
46:    CREATE Array Level4Hits of size 256, Initialize all values to 0
47:    pixel_list = Get_Image_Data(img)
48:    for pixel = 0 to 6399 do
49:       if pixel_list[pos] != Color White then
50:          Level1Hits[PixelZoneMapperArrayLevel1[pixel] − 1] ⇐ 1
51:          Level2Hits[PixelZoneMapperArrayLevel2[pixel] − 1] ⇐ 1
52:          Level3Hits[PixelZoneMapperArrayLevel3[pixel] − 1] ⇐ 1
53:          Level4Hits[PixelZoneMapperArrayLevel4[pixel] − 1] ⇐ 1
54:          STORE in Database Table Level1: FontName, CharCode, Level1Hits
55:          STORE in Database Table Level2: FontName, CharCode, Level2Hits
56:          STORE in Database Table Level3: FontName, CharCode, Level3Hits
57:          STORE in Database Table Level4: FontName, CharCode, Level4Hits
58:       end if
59:    end for
60: end for
```

Figure 3.16: Pseudocode for Algorithm to Create HitZone Maps for All Glyphs in a Font and Store in Database

3.2 Visual HitZone Comparison

3.2.1 Glyph Similarity Calculation with Granularity Adjustment

3.2.1(a) Theory

As previously discussed, the HitZone map generation algorithm generates HitZone maps for 4 different levels of granularity for each glyph. This glyph similarity calculation algorithm operates on a pair of HitZone maps, one from each glyph to be compared. As such, glyphs can be compared at different levels of granularity depending on the HitZone maps supplied as input. It should be noted that this algorithm is designed to compare HitZone maps of the same granularity level.

When we choose a level of granularity for comparison, we are essentially saying, "I want to compare these two glyphs at this level of precision." Higher levels of granularity will result in more precise comparisons of similarity. The level of similarity

desired strongly depends on the use case. For example, if a person is purely interested in the percentage of similarity between two glyphs, a high granularly level, such as 3 or 4, should be used. Conversely, when the similarity calculation is being used to filter out glyphs that are of a low similarity level, such as when this algorithm is used as part of the "Homoglyph Prediction with Seekback" algorithm (described in Section 3.2.3), a lower level of granularity may be desired in order to get a large set of potentially similar glyphs for future, more detailed comparison.

The glyph similarity algorithm works as follows. HitZone maps for two glyphs are given as input. The algorithm iterates through both HitZone maps simultaneously and compares each pair of zones at each zone index. It identifies and keeps count of three categories: number of zones where both glyphs have a hit (Common Hits), number of zones where only glyph 1 has a hit (Glyph 1 Unique Hits), and number of zones where only glyph 2 has a hit (Glyph 2 Unique Hits). A visual representation of this can be seen for two example level 1 granularity HitZone maps in Figure 3.17.

HitZone Map Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Glyph 1:	0	0	0	1	0	1	1	0	0	1	1	0	1	0	0	0
Glyph 2:	0	0	1	0	0	1	1	0	0	1	1	0	0	1	0	0
Common Hits: 4												Glyph 1 Unique Hits: 2			Glyph 2 Unique Hits: 2	

Figure 3.17: Comparison of Two Example Glyph HitZone Maps at Level 1 Granularity

We can then calculate the percentage of similarity by comparing the number of common hits with the sum of the common and unique hits. This comparison works because HitZone maps take glyph structure into account by design. The comparison in Figure 3.17 is essentially like laying Glyph 2 on top of Glyph 1 and observing which zones "stick out." The amount of "stick out" compared to the amount of similarity gives

us a percentage measure of how similar the glyphs are. For example, if we lay the glyphs represented in Figure 3.17 on top of each other, we can see that zones 5, 6, 9, and 10 are the same. Glyph 1 has 2 zones (3 and 12) that "stick out" and Glyph 2 has 2 zones (2 and 13) that "stick out." A total of 4 zones that "stick out," compared with the 4 zones that are the same, shows us that half of the total zone with hits in them are different. Thus, we can say that the glyphs are 50% similar. The following formula can be used to calculate the percentage of similarity:

$$Percentage\ Similarity = \lfloor \frac{Common\ Hits}{Common\ Hits + Glyph\ 1\ Unique\ Hits + Glyph\ 2\ Unique\ Hits} * 100 \rfloor$$

3.2.1(b) Implementation

Pseudocode for the implementation of this algorithm is as follows. Note that the inputs *Glyph1_HitZone_Map* and *Glyph2_HitZone_Map* must be HitZone Maps of the same level of granularity.

Algorithm 3 CalcGlyphSimilarity

Input: $Glyph1\_HitZone\_Map$, $Glyph2\_HitZone\_Map$

```
1:  zones = count(Glyph1_HitZone_Map)
2:  similar_zones ⇐ M1_hit_zones ⇐ M2_hit_zones ⇐ 0
3:  for x = 0 to zones do
4:     if Glyph1_HitZone_Map[x] = 1 && Glyph2_HitZone_Map[x] = 1 then
5:        INCREMENT similar_zones
6:     else if Glyph1_HitZone_Map[x] = 1 && Glyph2_HitZone_Map[x] = 0 then
7:        INCREMENT M1_hit_zones
8:     else if Glyph1_HitZone_Map[x] = 0 && Glyph2_HitZone_Map[x] = 1 then
9:        INCREMENT M2_hit_zones
10:    end if
11: end for
12: return ⌊similar_zones/(similar_zones + M1_hit_zones + M2_hit_zones) * 100⌋
```

Figure 3.18: Pseudocode for Algorithm to Compute Percentage Similarity Between two Glyph HitZone Maps

3.2.2 Homoglyph Prediction with Granularity Adjustment and Hit Percentage Filtering

3.2.2(a) Theory

In order to predict the homoglyphs of a given glyph, we want to get all glyphs that are potential matches and filter out glyphs that do not meet a specific threshold percentage of similarity. In order to do this, we first get the HitZone map for the given glyph at the desired level of granularity and take note of which zones have hits. We then find all other glyphs that have hits in the same zones. These glyphs are a list of potential homoglyphs.

HitZone Maps were designed so that a database engine could compare them quickly and easily. In order to obtain a list of potential homoglyphs, we retrieve all rows from our database table corresponding to the desired granularity level where there is a 1 in the column corresponding to each zone that has a 1 in our given glyph's HitZone map. A sample SQL query can be seen in Figure 3.19.

```
SELECT * FROM `Level1` WHERE `Font` = 'Arial' AND
 `Z5` = '1' AND `Z6` = '1' AND `Z10` = '1' AND `Z11` = '1'
```

Figure 3.19: Example SQL Query for Obtaining All Potential Homoglyphs in Font Set Arial Matching a Glyph at Level 1 Granularity with Hits in Zones 5, 6, 10, and 11

After we obtain this list of potential homoglyphs, we use the similarity calculation algorithm as described in Section 3.2.1 to compare each potential homoglyph to the given glyph and filter out the potential homoglyphs that do not meet the desired similarity threshold.

3.2.2(b) Implementation

The pseudocode for the homoglyph prediction algorithm can be found below. For this specific implementation, we include *FontName* as a means of retrieving only potential homoglyphs from within a specific font set. This was done because in the majority of cases, we only want to predict homoglyphs for the font set that is being used by the system we are working with. If *FontName* is removed from the database query, the algorithm will predict homoglyphs from all fonts in the database.

Algorithm 4 PredictHomoglyphs

Input: $FontName$, $GranularityLevel$
Input: $PercentageSimilarity$, $CharCode$
1: $CC\_HitZones \Leftarrow$ RETRIEVE HitZones of $CharCode$, $FontName$, $GranularityLevel$ from Database
2: $CC\_HitZone\_Map \Leftarrow$ Process $CC\_Hitzones$ into HitZone Map array
3: $DB\_Query \Leftarrow$ RETRIEVE Entries from Table $GranularityLevel$ Where $Font = FontName$
4: **for** $x = 0$ to $count(CC\_HitZone\_Map)$ **do**
5: **if** $CC\_HitZone\_Map[x] = 1$ **then**
6: Add Filter Clause to DB_Query Specifying Where $Zone\ x = 1$
7: **end if**
8: **end for**
9: $results \Leftarrow$ Get all rows from EXECUTE DB_Query
10: $PotentialHomoglyphs \Leftarrow$ CREATE Array of Size $count(results)$
11: **for** $x = 0$ to $count(results)$ **do**
12: $CC\_Temp \Leftarrow$ PARSE $CharCode$ from $results$
13: $HitZone\_Map\_Temp \Leftarrow$ PARSE and PROCESS HitZone Map array from $results$
14: PUSH $(CC\_Temp => HitZone\_Map\_Temp)$ onto $PotentialHomoglyphs$
15: **end for**
16: $Homoglyphs \Leftarrow$ CREATE Array of Size $count(results)$
17: **for all** $PotentialHomoglyphs$ as $CC\_Temp => HitZone\_Map\_Temp$ **do**
18: $SimCalc \Leftarrow CalcGlyphSimilarity(CC\_HitZone\_Map,\ HitZone\_Map\_Temp)$
19: **if** $SimCalc >= PercentageSimilarity$ **then**
20: PUSH CC_Temp onto $Homoglyphs$
21: **end if**
22: **end for**
23: REMOVE Empty Array Blocks from $Homoglyphs$
24: **return** $Homoglyphs$

Figure 3.20: Pseudocode for Algorithm to Predict Homoglyphs of a Given Glyph, with a Specific Level of Granularity, At or Above a Specific Threshold Percentage of Similarity

3.2.3 Homoglyph Prediction with Granularity Adjustment, Hit Percentage Filtering, and Seekback

3.2.3(a) Theory

The homoglyph prediction algorithm described in Section 3.2.2 has a major limitation: it only finds potential homoglyphs that have hits in at least in the same zones as the given glyph. While this produces highly accurate matches, it is likely to exclude potentially valid matches at higher levels of granularity. In order to remedy this limitation, a modified algorithm with a "seekback" technique is proposed.

The idea behind Seekback is simple. In order to predict homoglyphs at a specific level of granularity, we first compute a list of all homoglyphs at a lower level of granularity. We then compare all glyphs on that list at the desired granularity level, filtering out those that do not meet the desired similarity threshold.

By "seeking back" to a lower level of granularity, we obtain a preliminary list of potential homoglyphs. This preliminary list does not exclude glyphs that do not have hits in the same exact zones as the given glyph at the desired granularity level. By retrieving and comparing the HitZone maps for all of the glyphs on the preliminary list at the desired granularity level, we arrive at a final list of predicted homoglyphs that is much more accurate in terms of inclusivity, and hence remedy the limitation of the initial algorithm.

There may be certain situations in which Seekback is not desired. One such situation may be a case in which only homoglyphs with 100% similarity at a specific level of granularity are desired. Further, the number of levels of granularity to go back for

generating the preliminary list can be varied based on use case implementation requirements.

3.2.3(b) Implementation

The pseudocode for the Homoglyph Prediction Algorithm variant with Seekback can be found in Figure 3.21. Note that this implementation goes back 1 level of granularity from the desired granularity level for the Seekback feature. It is up to the person implementing the algorithm to decide if they want to seek back more than 1 level of granularity, or, if they always want to seek back to a fixed level of granularity. All such variants are valid and could have utility in different use cases.

Algorithm 5 PredictHomoglyphsSeekback
Note: Not to be used at Granularity Level 1

Input: $FontName$, $GranularityLevel$
Input: $PercentageSimilarity$, $CharCode$
1: $CC\_HitZones \Leftarrow$ RETRIEVE HitZones of $CharCode$, $FontName$, $GranularityLevel - 1$ from Database
2: $CC\_HitZone\_Map \Leftarrow$ Process $CC\_Hitzones$ into HitZone Map array
3: $DB\_Query \Leftarrow$ RETRIEVE Entries from Table $GranularityLevel - 1$
4: **for** $x = 0$ to $count(CC\_HitZone\_Map)$ **do**
5: **if** $CC\_HitZone\_Map[x] = 1$ **then**
6: Add Filter Clause to DB_Query Specifying Where $Zone\ x = 1$
7: **end if**
8: **end for**
9: $results \Leftarrow$ Get all rows from EXECUTE DB_Query
10: $PotentialHomoglyphs \Leftarrow$ CREATE Array of Size $count(results)$
11: **for** $x = 0$ to $count(results)$ **do**
12: $CC\_Temp \Leftarrow$ PARSE $CharCode$ from $results$
13: $HitZone\_Map\_Temp \Leftarrow$ PARSE and PROCESS HitZone Map array from $results$
14: PUSH ($CC\_Temp => HitZone\_Map\_Temp$) onto $PotentialHomoglyphs$
15: **end for**
16: $Homoglyphs \Leftarrow$ CREATE Array of Size $count(results)$
17: **for all** $PotentialHomoglyphs$ as $CC\_Temp => HitZone\_Map\_Temp$ **do**
18: $SimCalc \Leftarrow CalcGlyphSimilarity(CC\_HitZone\_Map,\ HitZone\_Map\_Temp)$
19: **if** $SimCalc >= PercentageSimilarity$ **then**
20: PUSH CC_Temp onto $Homoglyphs$
21: **end if**
22: **end for**
23: REMOVE Empty Array Blocks from $Homoglyphs$
24: $CC\_HitZones \Leftarrow$ RETRIEVE HitZones of $CharCode$, $GranularityLevel$ from Database
25: $CC\_HitZone\_Map \Leftarrow$ Process $CC\_Hitzones$ into HitZone Map array
26: $DB\_Query \Leftarrow$ RETRIEVE Entries from Table $GranularityLevel$ Where $Font = FontName$
27: **for** $x = 0$ to $count(Homoglyphs)$ **do**
28: Add Filter Clause to DB_Query Specifying Where $CharCode\ = Homoglyphs[x]$
29: **end for**
30: $results \Leftarrow$ Get all rows from EXECUTE DB_Query
31: $PotentialHomoglyphs \Leftarrow$ CREATE Array of Size $count(results)$
32: **for** $x = 0$ to $count(results)$ **do**
33: $CC\_Temp \Leftarrow$ PARSE $CharCode$ from $results$
34: $HitZone\_Map\_Temp \Leftarrow$ PARSE and PROCESS HitZone Map array from $results$
35: PUSH ($CC\_Temp => HitZone\_Map\_Temp$) onto $PotentialHomoglyphs$
36: **end for**
37: $Homoglyphs \Leftarrow$ CREATE Array of Size $count(results)$
38: **for all** $PotentialHomoglyphs$ as $CC\_Temp => HitZone\_Map\_Temp$ **do**
39: $SimCalc \Leftarrow CalcGlyphSimilarity(CC\_HitZone\_Map,\ HitZone\_Map\_Temp)$
40: **if** $SimCalc >= PercentageSimilarity$ **then**
41: PUSH CC_Temp onto $Homoglyphs$
42: **end if**
43: **end for**
44: REMOVE Empty Array Blocks from $Homoglyphs$
45: **return** $Homoglyphs$

Figure 3.21: Pseudocode for Variant of Homoglyph Prediction Algorithm (Algorithm 4) with 1 Level of Seekback

4. Experimental Study and Analysis

4.1 Algorithm Analysis

4.1.1 Glyph Similarity Calculation

A line-by-line Big-O analysis of the Glyph Similarity Calculation Algorithm (Algorithm 3 – Figure 3.18) can be seen in Figure 4.1 below. The complexity of this algorithm is *O(n)*.

```
 1: O(1)
 2: O(1)
 3:      O(n){
 4:          O(1)
 5:          O(1)
 6:          O(1)
 7:          O(1)
 8:          O(1)
 9:          O(1)
10:          -
11:      }
12: O(1)
```

Figure 4.1: Line-by-line Big-O Analysis of Glyph Similarity Calculation Algorithm (Algorithm 3 - Figure 3.18)

4.1.2 Homoglyph Prediction Algorithm

A line-by-line Big-O analysis of the Homoglyph Prediction Algorithm (Algorithm 4 - Figure 3.20) can be seen in Figure 4.2 below. Assuming the CharCode column of our database is indexed, the un-simplified complexity of the algorithm is *O(m*n + 2*p*n)*, where m is the number of rows in the database table, p is the number of results from the database query looking for potential homoglyph matches, and n is the number of zones in the HitZone map array. Because $p \leq m$, the simplified complexity of this algorithm is *O(m*n)*.

```
1: O(n), where n is the number of elements in the HitZone map array
2: O(n)
3: O(1)
4:      O(n){
5:          O(1)
6:          O(1)
7:          -
8:      }
9: O(m*n), where m is the number of rows in the database table
10: O(1)
11:     O(p){, where p is the number of results from the DB query
12:         O(1)
13:         O(n)
14:         O(1)
15:     }
16: O(1)
17:     O(p){
18:         O(n)
19:         O(1)
20:         O(1)
21:         -
22:     }
23: O(1)
24: O(1)
```

Figure 4.2: Line-by-line Big-O Analysis of Homoglyph Prediction Algorithm (Algorithm 4 - Figure 3.20)

4.1.3 Homoglyph Prediction Algorithm with Seekback

A line-by-line Big-O analysis of the Homoglyph Prediction Algorithm with Seekback (Algorithm 5 - Figure 3.21) can be seen in Figure 4.3 below. Assuming the CharacterCode column of our database is indexed, the un-simplified complexity of the algorithm is $O(m*n + 2*p*n + 3*r*q)$, where m is the number of rows in the database table, p is the number of results from the database query looking for potential homoglyph matches, and n is the number of zones in the HitZone map array at GranularityLevel -1, r is the number of elements in the Homoglyphs array in line after comparing at GranularityLevel -1, and q is the number of elements in the HitZone map array at the desired GranularityLevel. This can theoretically be simplified to $O(max(m*n, r*q))$. However, it is impossible to know the value of r prior to runtime, so this is not a useful bound. We know that $m*q$ is larger than either $m*n$ or $r*q$ because $n < q$ and $m > r$. Thus, for the purpose of utility, we bound this algorithm at $O(m*q)$.

```
1: O(n), where n is the number of elements in the HitZone map array
2: O(n)
3: O(1)
4:     O(n){
5:         O(1)
6:         O(1)
7:         -
8:     }
9: O(m*n), where m is the number of rows in the database table
10: O(1)
11:     O(p){, where p is the number of results from the DB query
12:         O(1)
13:         O(n)
14:         O(1)
15:     }
16: O(1)
17:     O(p){
18:         O(n)
19:         O(1)
20:         O(1)
21:         -
22:     }
23: O(1)
24: O(q), where q is the number of elements in the HitZone map array at the desired Granularity
    Level
25: O(q)
26: O(1)
27:     O(r){, where r is the number of elements in the Homoglyphs array in line 23
28:         O(1)
29:     }
30: O(r*q)
31: O(1)
32:     O(r){
33:         O(1)
34:         O(q)
35:         O(1)
36:     }
37: O(1)
38:     O(r){
39:         O(q)
40:         O(1)
41:         O(1)
42:     }
43: O(1)
44: O(1)
```

Figure 4.3: Line-by-line Big-O Analysis of Homoglyph Prediction Algorithm with 1 Level Seekback (Algorithm 5 - Figure 3.21)

4.2 Accuracy Testing

4.2.1 Glyph Similarity

Figure 4.4 below provides a small sample of input glyphs and the percentage of similarity calculated by the algorithm. Hundreds of characters from different languages were tested. Accuracy of results were assessed by visually examining the rendered versions of the glyphs given as input, mentally assessing an estimated percentage of similarity, and comparing the assessed percentage of similarity with the algorithmically calculated percentage of similarity. This figure shows comparison of glyphs at granularity levels 3 and 4. It can be seen that for glyphs that are visually identical, such as a "Latin Capital Letter A" and a "Greek Capital Letter Alpha" when rendered in Arial or Times New Roman fonts, the percentage of similarity is 100% at both levels of granularity. This contrasts with the results from pairs of glyphs that are "similar but not identical," such as a "Vertical Line" and a "Latin Capital Letter I." In this case, as expected, an increased level of granularity results in decreased percentage of similarity.

Font	Granularity Level	Glyph 1	Glyph 2	Glyph 1 Rendered	Glyph 2 Rendered	Calculated % Similarity
Arial	3	Vertical Line (0x7C)	Latin Capital Letter I (0x49)			100%
Arial	4	Vertical Line (0x7C)	Latin Capital Letter I (0x49)			80%
Times New Roman	3	Vertical Line (0x7C)	Latin Capital Letter I (0x49)			38%
Times New Roman	4	Vertical Line (0x7C)	Latin Capital Letter I (0x49)			34%

Font	Granularity Level	Glyph 1	Glyph 2	Glyph 1 Rendered	Glyph 2 Rendered	Calculated % Similarity
Arial	3	Latin Capital Letter A (0x41)	Greek Capital Letter Alpha (0x391)	A	A	100%
Arial	4	Latin Capital Letter A (0x41)	Greek Capital Letter Alpha (0x391)	A	A	100%
Times New Roman	3	Latin Capital Letter A (0x41)	Greek Capital Letter Alpha (0x391)	A	A	100%
Times New Roman	4	Latin Capital Letter A (0x41)	Greek Capital Letter Alpha (0x391)	A	A	100%
Arial	3	Cyrillic Small Letter Zhe (0x436)	Latin Small Letter X (0x78)	ж	x	77%
Arial	4	Cyrillic Small Letter Zhe (0x436)	Latin Small Letter X (0x78)	ж	x	62%
Times New Roman	3	Cyrillic Small Letter Zhe (0x436)	Latin Small Letter X (0x78)	ж	x	66%
Times New Roman	4	Cyrillic Small Letter Zhe (0x436)	Latin Small Letter X (0x78)	ж	x	35%

Figure 4.4: Example Similarity Algorithm Results

4.2.2 Homoglyph Prediction

Visual observation was used to confirm accuracy for the homoglyph prediction algorithm in a manner similar to that described in Section 4.2.1. Sample results can be seen in Figures 4.5 – 4.7.

Description of Input:

Rendered Glyph	Glyph	Percentage Similarity	Font	Level of Granularity
A	Latin Capital Letter A (0x41)	90	Arial	4

Results:

Glyph	HexChrCode	Glyph Description
Α	0x391	Greek Capital Letter Alpha
Ά	0x386	Greek Capital Letter Alpha with Tonos
А	0x410	Cyrillic Capital Letter A

Figure 4.5: Predicted Homoglyphs of "Latin Capital Letter A", At or Above 90% Similarity, for Font Arial, at Level 4 Granularity

Description of Input:

Rendered Glyph	Glyph	Percentage Similarity	Font	Level of Granularity
o	Latin Small Letter O (0x6f)	100	Arial	4

Results:

Glyph	HexChrCode	Glyph Description
о	0x43e	Cyrillic Small Letter O
ο	0x3bf	Greek Small Letter Omicro

Figure 4.6: Predicted Homoglyphs of "Latin Small Letter O", 100% Similarity, for Font Arial, at Level 4 Granularity

Description of Input:

Rendered Glyph	Glyph	Percentage Similarity	Font	Level of Granularity
━	Hyphen-Minus (0x2D)	75	Courier New	3

Results:

Glyph	HexChrCode	Glyph Description
‒	0x2012	Figure Dash
–	0x2013	En Dash
—	0x2014	Em Dash
―	0x2015	Horizontal Bar
	0xad	Soft Hyphen
━	0x25ac	Black Rectangle
─	0x2500	Box Drawings Light Horizontal
−	0x2212	Minus Sign

Figure 4.7: Predicted Homoglyphs of "Hyphen-Minus" Symbol, At or Above 75% Similarity, for Font Courier New, at Level 3 Granularity

4.2.3 Homoglyph Prediction with Seekback

Visual observation was again used to confirm accuracy for the homoglyph prediction algorithm with Seekback technique. The results can be seen in Figures 4.8 – 4.10. It should be noted that this implementation of the algorithm has 1 level of Seekback, as described in Section 3.2.3.

Description of Input:

Rendered Glyph	Glyph	Percentage Similarity	Font	Level of Granularity
ף	Hebrew Letter Final Pe (0x5e3)	73	Times New Roman	3

Results:

Glyph	HexChrCode	Glyph Description
ɳ	0x19e	Latin Small Letter N with Long Right Leg
ﬓ	0xfb43	Hebrew Letter Final Pe with Dagesh

Figure 4.8: Predicted Homoglyphs of "Hebrew Letter Final Pe", At or Above 73% Similarity, for Font Times New Roman, At Level 3 Granularity (With 1 Level of Seekback)

Description of Input:

Rendered Glyph	Glyph	Percentage Similarity	Font	Level of Granularity
A	Latin Capital Letter A (0x41)	75	Courier New	4

Results:

Glyph	HexChrCode	Glyph Description
А	0x410	Cyrillic Capital Letter A
Α	0x391	Greek Capital Letter Alpha
Ά	0x1fbb	Greek Capital Letter Alpha with Oxia
Ὰ	0x1fba	Greek Capital Letter Alpha with Varia
Ἅ	0x1f0d	Greek Capital Letter Alpha with Dasia and Oxia
Ἂ	0x1f0a	Greek Capital Letter Alpha with Psili and Varia
Ἄ	0x1f0c	Greek Capital Letter Alpha with Psili and Oxia
Λ	0x39b	Greek Capital Letter Lamda
Ἀ	0x1f08	Greek Capital Letter Alpha with Psili
Ἁ	0x1f09	Greek Capital Letter Alpha with Dasia
Ἃ	0x1f0b	Greek Capital Letter Alpha With Dasia and Varia

Figure 4.9: Predicted Homoglyphs of "Latin Capital Letter A", At or Above 75% Similarity, for Font Courier New, At Level 4 Granularity (With 1 Level of Seekback)

Description of Input:

Rendered Glyph	Glyph	Percentage Similarity	Font	Level of Granularity
T	Latin Capital Letter T (0x5e3)	100	Tahoma	4

Results:

Glyph	HexChrCode	Glyph Description
Т	0x422	Cyrillic Capital Letter Te
Τ	0x3a4	Greek Capital Letter Tau

Figure 4.10: Predicted Homoglyphs of "Latin Capital Letter T", 100% Similarity, for Font Tahoma, At Level 4 Granularity (With 1 Level of Seekback)

4.3 Execution Time Performance

The computer used in this thesis was a Macbook Pro (Retina, 15-inch, Mid 2015) with a 2.8GHz Intel Core i7 Processor, 16GB 1600 MHz DDR3 RAM, AMD Radeon R9 M370X 2048MB Graphics Card, and Intel Iris Pro 1536MB onboard graphics. The computer was running macOS Sierra (Version 10.12.3).

In order to measure execution time, the Unix time utility was used. This utility executes and times the runtime of a program. It returns a measure of total time elapsed, as well as two different measures of time spent inside the CPU.[15,16] We are only concerned with the total time elapsed. The results in Figure 4.11 and 4.12 show minimum, mean, median, and maximum runtime values for each algorithm and configuration after 25 trials. The variance in runtime is attributed largely to network traffic fluctuation, as the database is stored on a remote server. This is discussed further in Section 4.4.

Algorithm:	**SimilarityCalc**	**SimilarityCalc**
Font:	Times New Roman	Times New Roman
Granularity:	Level 3	Level 4
% Similarity:	n/a	n/a
Input 1 Rendered:	|	|
Input 1:	Vertical Line (0x7C)	Vertical Line (0x7C)
Input 2 Rendered:	I	I
Input 2:	Latin Capital Letter I (0x49)	Latin Capital Letter I (0x49)
Trial 1 Time (sec):	0.171	0.171
Trial 2 Time (sec):	0.153	0.177
Trial 3 Time (sec):	0.147	0.175
Trial 4 Time (sec):	0.152	0.172
Trial 5 Time (sec):	0.153	0.179
Trial 6 Time (sec):	0.144	0.166
Trial 7 Time (sec):	0.149	0.17
Trial 8 Time (sec):	0.144	0.176
Trial 9 Time (sec):	0.154	0.174
Trial 10 Time (sec):	0.179	0.183
Trial 11 Time (sec):	0.146	0.166
Trial 12 Time (sec):	0.152	0.171
Trial 13 Time (sec):	0.147	0.175
Trial 14 Time (sec):	0.147	0.184
Trial 15 Time (sec):	0.143	0.174
Trial 16 Time (sec):	0.144	0.17
Trial 17 Time (sec):	0.15	0.181
Trial 18 Time (sec):	0.142	0.164
Trial 19 Time (sec):	0.148	0.169
Trial 20 Time (sec):	0.153	0.175
Trial 21 Time (sec):	0.149	0.178
Trial 22 Time (sec):	0.145	0.171
Trial 23 Time (sec):	0.145	0.184
Trial 24 Time (sec):	0.158	0.167
Trial 25 Time (sec):	0.147	0.171
Min (sec):	0.142	0.164
Mean (sec):	0.15048	0.17372
Median (sec):	0.148	0.174
Max (sec):	0.179	0.184

Figure 4.11: Runtime Statistics for Similarity Calculation Algorithm at Granularity Level 3 and 4

Algorithm:	**Predict**	**Predict**	**PredictSeekback**	**PredictSeekback**
Font:	Courier New	Courier New	Arial	Arial
Granularity:	Level 3	Level 4	Level 3	Level 4
% Similarity:	70	70	70	70
Input 1 Rendered:	–	–	Г	Г
Input 1:	Hyphen-Minus (0x2D)	Hyphen-Minus (0x2D)	Cyrillic Small Letter Ghe (0x433)	Cyrillic Small Letter Ghe (0x433)
Trial 1 Time (sec):	0.696	0.583	0.34	0.391
Trial 2 Time (sec):	0.678	0.664	0.333	0.381
Trial 3 Time (sec):	0.677	0.592	0.355	0.425
Trial 4 Time (sec):	0.696	0.563	0.414	0.428
Trial 5 Time (sec):	0.671	0.651	0.35	0.42
Trial 6 Time (sec):	0.671	0.603	0.336	0.407
Trial 7 Time (sec):	0.698	0.587	0.336	0.396
Trial 8 Time (sec):	0.688	0.588	0.353	0.436
Trial 9 Time (sec):	0.668	0.573	0.375	0.413
Trial 10 Time (sec):	0.684	0.579	0.356	0.436
Trial 11 Time (sec):	0.677	0.64	0.352	0.405
Trial 12 Time (sec):	0.671	0.643	0.351	0.41
Trial 13 Time (sec):	0.674	0.62	0.358	0.481
Trial 14 Time (sec):	0.675	0.674	0.349	0.446
Trial 15 Time (sec):	0.684	0.617	0.348	0.42
Trial 16 Time (sec):	0.683	0.599	0.376	0.45
Trial 17 Time (sec):	0.695	0.591	0.369	0.389
Trial 18 Time (sec):	0.699	0.61	0.381	0.388
Trial 19 Time (sec):	0.675	0.652	0.349	0.405
Trial 20 Time (sec):	0.663	0.578	0.333	0.398
Trial 21 Time (sec):	0.75	0.573	0.332	0.408
Trial 22 Time (sec):	0.67	0.693	0.38	0.401
Trial 23 Time (sec):	0.717	0.605	0.354	0.396
Trial 24 Time (sec):	0.68	0.643	0.358	0.375
Trial 25 Time (sec):	0.699	0.571	0.358	0.379
Min (sec):	**0.663**	**0.563**	**0.332**	**0.375**
Mean (sec):	**0.68556**	**0.61168**	**0.35584**	**0.41136**
Median (sec):	**0.68**	**0.603**	**0.353**	**0.407**
Max (sec):	**0.75**	**0.693**	**0.414**	**0.481**

Figure 4.12: Runtime Statistics for Homoglyph Prediction Algorithm and Homoglyph Prediction Algorithm with Seekback at Granularity Level 3 and 4

4.4 Implementation Notes and Considerations

It should be noted that storing the database on a computer separate from the computer used to execute queries may introduce random delays, as time required to communicate over a local network or the internet will vary widely with network traffic. Reasonable performance *can* however be achieved using a remote database. The implementation in this thesis used a remote database located in a managed professional datacenter with excellent connectivity in Ashburn, Virginia. Discussion of server location and configuration is beyond the scope of this thesis, but if a remote database is used, care should be taken to ensure the database server has excellent connectivity and is optimized for peak performance.

5. Potential Applications

There are a myriad of potential applications for the algorithms and techniques proposed in this thesis. The applications discussed below are purely theoretical. The discussions are intentionally brief and are meant to serve as a starting point for thinking about these concepts.

5.1 Copyright Infringement and General Plagiarism Detection

As we move into a digital age, it is becoming increasingly common for publications and documents that were once printed on paper to be provided, sometimes exclusively, in a digital format. Students and researchers submit papers as PDF or Microsoft Word documents, and authors publish books in the form of eBooks through various platforms such as the Amazon Kindle eBook store. As seen in the earlier discussion of Homoglyph Attacks (Section 2.2), homoglyphs can be used to bypass many plagiarism detection engines. This is clearly a major issue for academia. Taking this concept one step further, one can imagine a case in which an unscrupulous individual obtains copies of eBooks that they did not write and do not have rights to sell, replaces characters within those eBooks with homoglyphs to evade duplicate content detection mechanisms, and illegally sells the eBooks on an online eBook store.

In order to detect such abuse, a system could be built that checks submitted documents for mixed language character sets. For each letter in each word in the electronic document that contains glyphs from mixed languages, the system would use the algorithms discussed in this thesis to predict a list of homoglyphs, and it would then

check that list for homoglyphs in the language that the electronic document is supposed to be written in. Those homoglyphs, if found, would be used to create new words containing only characters of the expected language of the document. Each word would then be checked against a dictionary for validity. If the word is valid, the system would replace the questionable word in the document with the newly calculated one. At the end of this process, the newly generated document would be checked for similarity with other documents using standard plagiarism detection techniques.

5.2 App Store Review Process

At the beginning of Section 1.2 there is a discussion that details the possibility of submitting fake apps to an app store, for nefarious purposes, using homoglyphs to circumvent detection of similar app names. In order to detect such abuse, a similar approach to the one described in Section 5.1 could be used. But, instead of checking against a dictionary, the newly calculated app names could be checked against the current catalog of apps in the app store.

5.3 Domain Name Review – Registry Level

Homoglyph attacks in domain names are incredibly dangerous. They can be used to steal personal information and spread viruses. As discussed in Section 1.2, the current recommendations from Unicode are for web browsers to simply refrain from displaying most domain names with characters from different languages. Further, many domain

name registries do not allow non-English characters at all. The algorithms and techniques proposed in this thesis could be used as a starting point to liberate the world of internet users and developers from such restrictive policies. By predicting and detecting homoglyphs in domain names with mixed language sets, domain registries could check if someone is trying to register a domain name that looks like a domain name which is already in use. Such registration attempts could be denied or flagged for manual review. This would allow the use of domain names such as "i♥icecream.com," while preventing registration of domain names that are visually similar to others, and thus have the potential to be used for malicious purposes.

5.4 Penetration Testing and Exploit Discovery

Many web applications and pieces of software are unable to correctly process characters from languages other than those primarily used by the developer(s) of said application(s). This can be ascribed to many factors, including the complexity of the Unicode character space and poor understanding of code globalization concepts. In the best case, applications simply display a generic glyph, usually a block (□), representing a glyph rendering failure. In the worst cases, some characters can cause crashes, buffer overflows, and other undesired performance, which is occasionally exploitable.

The algorithms presented in this thesis may be useful to both those that want to improve their code and those trying to exploit code, as they provide an easy way to find homoglyphs. Such homoglyphs may be especially useful in exploits or attacks where the attacker must convince another individual to enter specific text.

5.5 Password Security

It is conceivable that the algorithms in this thesis could be used to generate highly secure passwords. For example, if a user wanted to make a secure variant of the password "birthday," they could replace several of the characters with homoglyphs and memorize the languages that the homoglyphs belong to. This would turn the unsecure password "birthday" into an incredibly secure password, as most attackers would not expect the characters in the password. A tool could be built to perform the conversion from the original plain text password to the "homoglyph encrypted" password, where the user enters the plain text password and is presented with dropdown menus below each letter to select the homoglyph language for that character. The tool would then return the "homoglyph encrypted" password for the user to copy and paste into wherever it is needed. If the user uses a password manager this tool would only need to be used once.

6. Conclusion

6.1 Discussion

Technology is rapidly improving and humanity is increasingly reliant on it. As the internet permeates the globe, it is time to recognize that a large percentage of internet and technology users do not speak or write in English as their first language. Denying people the use of characters from their native languages in various areas of technology because of the potential for homoglyph related attacks is a lazy "solution" to this type of problem. Conversely, ignoring the potential for such abuse is irresponsible. As demonstrated in this thesis, it is possible to predict and detect homoglyphs with a high level of accuracy, and contemporary computers are able to perform such computations with ease. It is time for the world to enter a new era of homoglyph security: one in which *detection* and *prevention* of such abuse supplants the denial of use of large portions of the Unicode character space that *may* enable abuse. Further, as previously demonstrated, there are many systems, such as plagiarism detection engines, that perform comparison checks or some type of filtering which can be bypassed using homoglyphs. This thesis should serve as a foundation for improvements necessary to prevent such abuses of these systems.

6.2 Future Directions

There are many potential future directions for this body of work. The HitZone map design allows for a number of variants of the Homoglyph Prediction algorithms to be created. Such variants could attempt to predict homoglyphs based on hit clustering, hit tracing, comparisons based on misses instead of hits, or comparisons based on both hits and misses.

7. References

1. "The Unicode Consortium." [Unicode]. Accessed April 20, 2017. http://www.unicode.org/consortium/consort.html.

2. "Unicode® 9.0.0." [Unicode]. Accessed April 20, 2017. http://www.unicode.org/versions/Unicode9.0.0/.

3. "Languages and Scripts." [Unicode]. Accessed April 20, 2017. http://www.unicode.org/cldr/charts/latest/supplemental/languages_and_scripts.html.

4. "Unicode Security Considerations." UTR #36: Unicode Security Considerations. Accessed April 20, 2017. http://unicode.org/reports/tr36/.

5. "IDN in Google Chrome." The Chromium Projects. Accessed April 20, 2017. https://www.chromium.org/developers/design-documents/idn-in-google-chrome.

6. "Unicode Security Mechanisms." [Unicode]. Accessed April 20, 2017. http://www.unicode.org/reports/tr39.

7. "IDN Display Algorithm." IDN Display Algorithm - MozillaWiki. Accessed April 20, 2017. https://wiki.mozilla.org/IDN_Display_Algorithm.

8. “POLICY for the organization and the management of the top-level Macedonian .MK domain and the top-level Macedonian .MKD domain.” PDF. MACEDONIAN ACADEMIC RESEARCH NETWORK, May 21, 2014. http://registrar.mk/docs/EN_MARnet_policy.pdf

9. VENKATA RAO, N., A.S.C.S. SASTRY, A.S.N. CHAKRAVARTHY, and KALYANCHAKRAVARTHI P. "OPTICAL CHARACTER RECOGNITION TECHNIQUE ALGORITHMS." Journal of Theoretical and Applied Information Technology 83, no. 2 (January 20, 2016). Accessed April 21, 2017.

10. Wang, Wei. "Optical Character Recognition, Using K-Nearest Neighbors." Computing Research Repository, November 7, 2014. Accessed April 21, 2017. https://arxiv.org/abs/1411.1442.

11. Jubair, Mohammad Imrul, and Prianka Banik. "A Simplified Method for Handwritten Character Recognition from Document Image." International Journal of Computer Applications 51, no. 15 (August 2012). Accessed April 21, 2017.

12. Miller, Joseph A. Homoglyph monitoring. US Patent US 20140115704 A1, filed October 24, 2012, and issued April 24, 2014.

13. Irongeek.com. "Out of Character: Use of Punycode and Homoglyph Attacks to Obfuscate URLs for Phishing." Irongeek. April 2012. Accessed April 21, 2017. http://www.irongeek.com/i.php?page=security%2Fout-of-character-use-of-punycode-and-homoglyph-attacks-to-obfuscate-urls-for-phishing.

14. Roshanbin, Narges, and James Miller. "Finding Homoglyphs - A Step towards Detecting Unicode-Based Visual Spoofing Attacks." Web Information Systems Engineering – WISE 2014 Workshops, LNCS, 9051 (October 12-14, 2014). doi:10.1007/978-3-319-20370-6.

15. "What do 'real', 'user' and 'sys' mean in the output of time(1)?" Stackoverflow. Accessed April 23, 2017. http://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1.

16. BSD General Commands Manual. Version 1.6c. Accessed April 23, 2017.