

# Dual Canonicalization: An Answer to the Homograph Attack

James N. Helfrich and Rick Neff

**Abstract**— Phishing attacks have frequently used homographs since they were first described a decade ago. Though a wide variety of tools and techniques have been used to mitigate the effectiveness of these attacks, none have offered a comprehensive solution. This paper describes the homograph attack as a mathematical problem, identifies the various flavors of the attack, and provides a solution applicable to a wide variety of scenarios.

**Index Terms**— Computer security, Pattern recognition

## I. INTRODUCTION

Homophones are two words pronounced the same but having different meanings or spellings (such as “hair” and “hare”). Homonyms are two words spelled the same but having different meanings (such as the noun “sow” and the verb “sow”). Homographs, by contrast, are two words or tokens that appear the same but are encoded differently (such as {0x48, 0x44, 0x4d, 0x4c} and {0x0397, 0x0422, 0xff2d, 0x13DE} for “HTML”). Homographs may be part of a deception attack if they can be used to mislead an observer into believing that an encoding represents something different than it actually does. This type of deception is called a “homograph attack.”

A homograph attack is defined as text “maliciously misspelled by substitution of non-Latin letters” [1] for the purpose of deceiving the viewer into thinking counterfeit text is legitimate. Homograph attacks, also known as Unicode attacks [2] and homograph obfuscation [3], were first formally defined by Gabrilovich and Gontmakher in 2002. However, homograph attacks date back at least a year earlier [4]. Fu, Deng, and Wenxin independently theorized about the possibility of homograph attacks in 2005 [5].

There are two variants of the homograph attack: production of counterfeit text that appears legitimate and production of text recognizable by a human but not by a filter [2]. Both of these variants rely on the homograph property of having multiple encodings producing the same visual representation.

The first variant is commonly used in URL spoofing and phishing attacks. In these cases, the attacker presents to an observer a URL that appears to be directed towards a trusted site when in fact it is directed towards a malicious site. This URL could be schematically similar ([www.firsttech.com](http://www.firsttech.com) and

[www.1sttech.com](http://www.1sttech.com)) [6], a common misspelling or alternate spelling ([www.greybeard.com](http://www.greybeard.com) and [www.graybeard.com](http://www.graybeard.com)) [7], or contain non-Latin characters ([www.homograph.com](http://www.homograph.com) and [www.ḡomograph.com](http://www.ḡomograph.com) where the o’s are Greek in the second example) [8] [5]. These attack vectors have been called “Web Identity Attacks,” “Phishing Attacks,” [2] and “Unicode Attacks” [2].

The second variant is commonly used in SPAM, where the attacker attempts to pass a message through a filter it is designed specifically to detect [2]. If, for example, the filter is designed to block messages containing the word “homograph,” the attacker may try “HOMOGRAPH,” “homograph” where the o’s are Greek, or even “h.o.m.o.g.r.a.p.h.” Each of these strings can be easily read by an observer but may not be detected by a filter.

## II. MITIGATING HOMOGRAPH ATTACKS

Since Homograph Attacks were first identified in the literature, a number of mitigation strategies have been described. Possibly the simplest is to prohibit homographs. In the case of URL spoofing with Unicode characters, a rule could be made prohibiting mixing alphabets. While this might prevent confusion of the Cyrillic, Greek, and Latin ‘o’ characters, it would also prohibit CNNenEspañol.com [1]. Furthermore, there would be nothing to prevent homographs consisting of a single script [6]. Consider the text “HAMBOT” consisting of Latin characters. This appears identical to the Greek string “HAMPBOT” and the Cyrillic string “HAMPBOT.” In each case, the strings are in a single alphabet yet are homographs.

### A. Punycode

A second approach is to identify text residing in a script not native to the viewer’s browser. This text is converted into a unique label called “Punycode” which will appear obviously different than the spoofed text. Thus “[www.homograph.com](http://www.homograph.com)” where the o’s are Greek is rendered as “[www.xn--hmgraph-8ofb.xn--cm-jbc](http://www.xn--hmgraph-8ofb.xn--cm-jbc)” [9]. This technique is used by the browsers Mozilla Firefox and Microsoft Internet Explorer. When the user’s browser is configured to read Greek and Latin text, however, the user will be vulnerable to Greek-Latin homograph attacks.

### B. Script Coloring

A third approach is to highlight characters of each script with a distinct color [1] [3]. This technique has been used in a

variety of anti-phishing toolbars and browser add-ins [6]. There are two problems with this approach. First, legitimate sites with mixed scripts (such as “CNNEspañol.com”) may appear suspicious to users. Second, studies have shown that such highlighting is not prominent enough to warn the user and, even when users notice the warning, they do not know what action is needed [10].

### C. Heuristics

A fourth approach is to construct a heuristic computing the probability that a given e-mail has the characteristics of a Phishing attack. This heuristic [6] includes mixed-script spoofing (mixing Greek and Latin in the same string), whole-script spoofing (text appearing Latin but consisting of Greek characters), single-script spoofing (“rn” and “m”) [11], syntax spoofing ({/} and {0x002F}), numeric spoofing (Bengali zero {0x09e6} and {0x0030} for ‘0’), invisible character spoofing ({0x200B} as the zero width space), bidirectional text spoofing ({0x202e, 0x202c} to turn on then off right-to-left reading), combining mark order spoofing ({e, 0x0300} and {0x00e8} for ‘è’), and inadequate rendering support ({e, 0x0300, 0x0300, 0x0300} ignores the last two accent marks because they are redundant). Examples of these heuristic tools include the IRI/IDN SecuChecker [12], REGAP [13], and the Quero Toolbar [6]. Each of these tools has proven successful in detecting many homographs, but none offers a generic or comprehensive solution to the problem.

### D. Visual Similarity

The final approach to detecting the presence of homograph character pairs is to compare the degree of visual similarity between the glyphs [2]. One way to measure this visual similarity is through the Earth Mover’s Distance (EMD) algorithm traditionally used to compare images. When glyphs are rendered into pixels, the EMD value can be computed directly [14]. The second way is to use the Kernel Density Estimation (KDE) to accomplish the same [15]. Using this technique, Fu et al. were able to construct the UC-SimList, a collection of homograph sets whose members meet a given visual similarity threshold. This technique is effective for detecting counterfeit Unicode-encoded text designed to appear legitimate, but does not address many other homograph attack vectors.

## III. THEORETICAL FRAMEWORK

The first step in mitigating homograph attacks is to precisely define a homograph. We define a homograph as “a set of encodings that, when displayed to the user, are perceived as being the same.” There are several components to this definition:

$e$       *Encoding*: A representation of some presentation. Examples include Unicode for text, HTML for documents, MP3 for sound, and JPG for images.

$r$       *Rendition*: How a given encoding  $e$  appears to the observer. This could be pixels on the screen or music in the ears.

$R(e_1) \rightarrow r_1$       *Rendering function*: How a given encoding is rendered or displayed into the rendition format. This could be an edit control in a browser’s address textbox, Adobe Acrobat® reader, or a music player.

$O(r_1, r_2) \rightarrow p$       *Observer function*: The probability that a given observer will consider two renditions the same. This probability is called the threshold of belief. An alternate form of this is  $O(r_1, r_2, p)$  which returns true if the observer considers  $r_1$  and  $r_2$  the same at  $p$  probability and false otherwise.

The relationship between these definitions can be represented graphically in Figure 1:

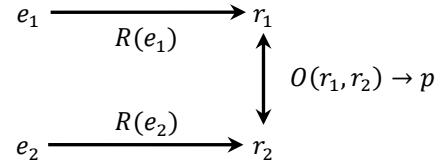


Figure 1: The relationship between the rendering function and the observer function

From these components, we can define the homograph function, homograph sets, and homographs.

$H(e_1, e_2) \rightarrow p$       *Homograph function*: The probability that a pair of encodings will be perceived as being the same to an observer. An alternate form of this is  $H(e_1, e_2, p)$  which returns true if the observer considers  $e_1$  and  $e_2$  the same at  $p$  probability and false otherwise. The homograph function is defined in terms of the observer function and the rendering function:

$$H(e_1, e_2, p) = O(R(e_1), R(e_2), p)$$

Equation 1: A homograph is a pair of encodings considered by the observer to be the same

$h$       *Homograph set*: A set of unique encodings perceived by the observer as being the same. In other words, two encodings  $e_1$  and  $e_2$  are considered homographs if an observer  $O()$  considers them to be in the same set  $h$ . This definition can be represented mathematically with:

$$\forall e_i, e_j \quad e_i \in h \wedge e_j \in h \leftrightarrow H(e_i, e_j) \geq p$$

Equation 2: A homograph set is a set of all encodings which are homographs of each other

This allows us to define homographs. Homographs are two or more encodings belonging to the same homograph set. Conversely, anti-homographs are two or more encodings belonging to different homograph sets. In other words, the threshold of belief as set by the observer function is below that computed by the homograph function.

#### A. Rendering Function and Observer Function

From these definitions and associated equations, we can see that several components need to be understood in a given homograph scenario. First, the rendering function  $R(e)$  must be well-defined. The encodings  $\{0x48, 0x44, 0x4d, 0x4c\}$  and  $\{0x0397, 0x0422, 0xff2d, 0x13DE\}$  may appear the same in an edit control using Ariel Unicode as a font but appear quite different when rendered in Cooper Black or another non-Unicode font. In order to completely identify homograph sets, the full capabilities of the rendering function must be incorporated.

The second component that must be understood is the observer function:  $O(r_1, r_2) \rightarrow p$ . In some attack scenarios such as phishing attacks, the renditions must be identical:  $O(r_1, r_2) = 1.0$ . However, most filter-avoiding attacks require a much lower level of user scrutiny: it is only necessary for the observer to be able to understand the message rather than believe they are the same. In this case, the observer function represents the probability that the observer will believe the two encodings represent the same meaning, not that they have identical presentations. In SPAM scenarios, the probability level can be set quite low in a successful attack because the number of sent messages is very high:  $O(r_1, r_2) = p_e$ , where  $p_e < 0.1$ .

### IV. DUAL CANONICALIZATION

We present a novel method for detecting if two encodings are homographs called “dual canonicalization.” First, we will formally define dual canonicalization as associated terms, then we will present this technique by example, and finally discuss implications of the technique.

#### A. Canonicalization

Two relevant concepts in the dual canonicalization method are that of the canon and the process of deriving a canon called canonicalization.

$c$  *Canon:* A canon is “a general rule, fundamental principle, aphorism, or axiom governing the systematic or scientific treatment of a subject” [16]. The canonical form is “in its simplest or standard form” [17]. In the context of homographs, a canon is defined as a unique representation of a homograph set. Note that the format of the canonical token  $c$  may or may not be the same format as the encoding  $e$  or the rendition format  $r$ .

$C(e) \rightarrow c$

*Canonicalization function:*

Canonicalization is “the process by which various equivalent forms of a name are resolved to a single, standard name – the canonical name” [18]. We will define this canonicalization process with the function  $C()$ , taking an encoding  $e$  as input and returning a canonical token  $c$ .

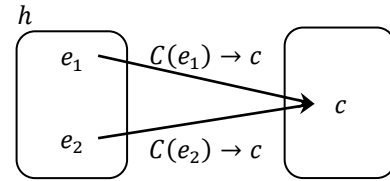
#### B. Canonicalization Properties

The canonicalization function must have two properties:

1. *Reliable canons property:* The canonicalization function will always yield identical canonical tokens for any homograph pair [19].

$$\forall e_1, e_2 \quad C(e_1) = C(e_2) \leftrightarrow H(e_1, e_2) \geq p$$

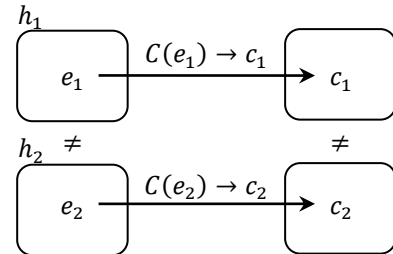
Equation 3: If the canons are the same for two encodings, they must be members of the same homograph set



2. *Unique canons property:* Any pair of non-homograph encodings will yield different canonical tokens.

$$\forall e_1, e_2 \quad C(e_1) \neq C(e_2) \leftrightarrow H(e_1, e_2) < p$$

Equation 4: If the canons are not the same for two encodings, they must be anti-homographs



We present a straightforward proof that any canonicalization function  $C(e)$  meeting these two requirements will be sufficient to detect whether or not a given pair of encodings  $e_1, e_2$  are homographs. In other words, we show that if a canonicalization function is both Reliable and Unique, then when applied to two encodings it will determine that either the encodings belong to the same homograph set, or else that they belong to different homograph sets.

Let  $C(e)$  be a canonicalization function that satisfies the Reliable Canons Property and the Unique Canons Property. Let  $e_1$  and  $e_2$  be two encodings, with  $e_1 \neq e_2$ . By the law of the excluded middle, either  $C(e_1) = C(e_2)$  or else  $C(e_1) \neq C(e_2)$ . Suppose that  $C(e_1) = C(e_2)$ . Then by Reliability,  $H(e_1, e_2) \geq p$ , which by definition means that  $e_1 \in h_1$  and  $e_2 \in h_2$  and  $h_1 = h_2$  (they're in the same homograph set). Suppose that  $C(e_1) \neq C(e_2)$ . Then by Uniqueness,  $H(e_1, e_2) < p$ , which by definition means that  $e_1 \in h_1$  and

$e_2 \in h_2$  and  $h_1 \neq h_2$  (they're in different homograph sets).

### C. Dual Canonicalization

Observe how, for a given pair of encodings, it is necessary to canonicalize both members to determine if they are homographs. This necessity is the source of the name “dual canonicalization.”

Consider the classic programming algorithm to perform case-insensitive string compare. Note that this is a simple homograph problem: the encoding is ASCII or Unicode text. The rendering function is the mapping of ASCII or Unicode text into the associated Ariel Unicode glyphs. The observer function is that the rendered glyphs have the same semantic meaning. Note that for the text “Homograph”, there are 512 members of the homograph set  $(\{H,h\} \cdot \{O,o\} \cdot \dots) = 2^9$  or  $2^n$  where  $n$  is the length of the string. Despite the  $2^n$  growth function, case-insensitive string compare is  $O(n)$  because the lowercase versions of both strings are compared. This is accomplished by converting both the input strings to lowercase using the `tolower()` function before the individual characters are compared. In other words, the  $n$ -to- $n$  problem can be reduced to a  $2 \times (n-1)$  problem [19]. The classic case-insensitive string compare is accomplished by canonicalizing both the right-hand-side and the left-hand-side of the search equation.

Observe how the canonicalization function must be consistent with the homograph function, and the homograph function is defined in terms of the rendering function and the observer function. Thus, in order to counter a homograph attack in a given context, it is necessary to take into account the full rendering capability of the rendering function as well as the degree of observer scrutiny as represented in the observer function. If either of these is not fully taken into account, a risk exists that some homographs will remain undetected.

An interesting fallout of this relationship is that it is possible for homographs to be detected to a given degree of confidence regardless of the encoding or presentation format as long as the observer and rendering function are known. Thus the dual canonicalization technique can be applied in many more contexts than the traditional Unicode homograph scenario.

## V. EXPERIMENTAL RESULTS

The dual canonicalization homograph detection technique can be applied to a wide variety of homograph attacks. To illustrate this point, three classic attacks will be discussed: perfect Unicode homographs in a URL phishing attack, near Unicode homographs used in body-text phishing attacks, and HTML homographs in a SPAM filter-avoidance attack.

### A. Classic Unicode Attack

The classic Unicode homograph attack, first described in 2002 [1], occurs when the attacker creates a counterfeit URL that appears like a legitimate URL yet consists of non-Latin character encodings. In this case, the rendering function is the

Arial Unicode font rendered in a Unicode-enabled edit control in a browser window. We will initially set the observer scrutiny level to 100%, two presentations will be considered the same iff they are pixel-for-pixel identical. Thus the observer function  $O(r_1, r_2) \rightarrow p$  can be reduced to  $r_1 = r_2$  because  $p \rightarrow 1.0$ . The final step is to identify the canonicalization function.

Fu et al. [2] identified a Unicode Similarity List (UC-SimList) describing homograph sets for the Arial Unicode font. A similar list can be generated from any TrueType through the `cmap` table, a mapping of character codes to glyph indices [20]. From this collection of homograph sets, the canonicalization function can be defined:  $c_1$  is the member of  $h_1$  with the lowest Unicode value.

To verify this program, a homograph generation  $HG()$  function was created that converts an encoding  $e$  into another random encoding  $e'$  such that both are members of the same homograph set  $h_1$ :

$$HG(e) \rightarrow e'$$

From here, 1,000 homographs as well as 1,000 anti-homographs were generated for a given input stream “homograph”. Each member of the homograph set was compared against all other members and 100% recognition was achieved. The 1000 “anti-homographs” were generated by substituting a single character from the string “homograph”. Each of these were compared against the 1,000 homographs with 0% recognition. Please see Appendix A for the first 10 items in the homograph and anti-homograph sets.

### B. Near Homograph Attack

Next we relax the level of observer scrutiny to include the set of “near homographs” or the set of characters that the observer will *probably* recognize as the same. Here the rendering function remains the same but the observer function is relaxed. The canonicalization function will then be determined by how similar rendered glyphs appear to the user (visual edit distance or VED computed using the Knuth-Morris-Pratt algorithm) and the semantic differences between the characters (SED). Fu et al. created the UC\_SimList<sub>v</sub> to represent these homograph sets [15]. Again, a set of 1,000 homographs were generated and all were detected. Similarly, 1,000 random terms did not register as homographs. Please see Appendix B for the first 10 items in the homograph and anti-homograph sets.

### C. HTML Filter Avoiding Attack

The final class of attack is the SPAM filter-avoidance scenario where the attacker attempts to pass a word (“Homograph”) through a filter yet remain readable to the observer. In this case, the encoding format is HTML. The rendering function is a web browser because most HTML clients use browsers for rich text rendering. Note that the number of renderings for the word “Homograph” is extremely large [21]. A few examples include “Homograph,” “Homo<b></b>graph”, “Hom<img src=“o.gif” />graph,” etc. [12]. The observer function is set to be

extremely lax because the SPAM is successful if the reader is simply able to understand the message; it is not necessary for it to appear identical to the reader [22].

We will define a canonical token for this scenario as ASCII text. The canonicalization function will generate a stream of canonical tokens from a given input HTML message through the following process:

- 1) The HTML input message will be rendered as an image through a browser (Internet Explorer 9 was used in this experiment).
- 2) The image is converted into Unicode text through an Optical Character Recognition (OCR) function (Free Online OCR from [www.free-online-ocr.com](http://www.free-online-ocr.com)).
- 3) All white spaces and punctuation are removed
- 4) Unicode homographs are canonicalized using UC\_SimList<sub>0.8</sub> as the homograph set.

To verify this process, 100 HTML-encoded homographs were created and 100 HTML-encoded anti-homographs were created. 91% of the homographs were detected (the OCR recognized 'm' as "rn" 4 times, 'p' was recognized as 'o' twice, 'o' was recognized as 'a' twice, and 'g' was recognized as 'a' once) and none of the non-homographs were detected. Please see Appendix C for the first 10 items in the homograph and anti-homograph sets.

## VI. CONCLUSION

This paper presented a mathematical model describing homograph attacks and a methodology called dual canonicalization which can be used to detect such attacks. This methodology was applied to three common homograph scenarios: Unicode URL spoofing, phishing attacks, and SPAM filter avoidance attacks. In each case, the dual canonicalization method accurately classified a large number of homographs with zero false positives and acceptable false negatives.

Because of the general homograph definition presented

herein, the dual canonicalization technique can be applied to a wide variety of contexts not currently addressed in the literature. One such area of future research is plagiarism detection. In this context, the encoding format is the written word. This could be ASCII-encoded text, Unicode-encoded text, a rich document format, or even printed text. The rendition function is reading, converting written words into ideas in the reader's mind. A canonicalization function must be able to map text into tokenized ideas for the homographs to be detected. Hoad and Zobel [19] identified such a method which they called a "fingerprinting process." By adapting the Hoad and Zobel method to the canonicalization function format and by ensuring the resulting function meets the reliable canons and unique canons properties, a robust plagiarism detection function should be achievable.

Another area for future research may be programming source-code plagiarism. In this context, the encoding format is a given source code language (C++, Java, C#, etc.). The rendition function is the way the program is compiled. The canonicalization function will need to not only tokenize assembly statements (a nearly trivial task), but also recognize variations having no influence on the outcome of the program (such as transposing the order of two independent assignment statements). A candidate canonicalization function may be a compiler parse tree. If such a canonicalization function can be built, then it should be easily adaptable to metamorphic virus detection.

A final area for future research may be song plagiarism. In this context, the encoding format is music in MP3, WAV, or WMA format. The rendition function is the process of converting the data into music through speakers. The observer function is relevant here because copy write laws allows artists to adapt the music of others. One possible canonicalization function may be to convert the music sheet music with varying degrees of fidelity. If such a canonicalization function can be built, the aforementioned Nabster homograph attack [4] and similar variations could finally be addressed.

## APPENDIX A

Encoding	Rendition
&#x4bb;omog&#xff52;&#x430;&#x440;&#x4bb;	homog r aph
h&#x43e;mo&#x261;&#xff52;a&#x440;&#x4bb;	homog r aph
&#x4bb;o&#xff4d;&#x3bf;gr&#xff41;&#xff50;&#xff48;	homogr a p h
&#x4bb;&#xff4f;m&#xff4f;gra&#xff50;&#x4bb;	h o m o gra p h
&#xff48;&#x3bf;m&#x3bf;&#x261;&#xff52;a&#x440;&#x4bb;	h omog r aph
h&#x43e;m&#xff4f;&#xff47;&#xff52;&#xff41;p&#x4bb;	hom o g r a ph
&#xff48;o&#xff4d;o&#xff47;ra&#x440;h	h o m o graph
&#xff48;&#xff4f;&#x217f;&#x3bf;&#xff47;&#xff52;a&#xff50;&#xff48;	h o m o g r a p h
hom&#x3bf;&#x261;&#xff52;&#x430;&#xff50;&#xff48;	homog r a p h
&#x4bb;&#x43e;&#xff4d;&#xff4f;&#x261;r&#xff41;p&#x4bb;	hom o gr a ph

Table 1: Perfect (100%) homographs for the word "homograph"

Encoding	Rendition
&#x4bb;&#x43e;m&#xff4f;&#x3a5;r&#x430;&#x440;&#x4bb;	hom o Yraph
&#x4bb;&#x43e;h&#x3bf;&#x261;&#xff52;a&#x440;h	hohog r aph
&#xff48;&#x43e;mo&#xff18;&#xff52;&#x430;&#xff50;h	h o m o 8 r a p h
h&#x43e;&#x217f;&#xff4f;&#xff47;&#xff52;&#x430;&#x440;&#x217c;	hom o g r apl
h&#x43e;m&#x43e;&#xff47;&#xff52;&#xff38;ph	homo g r Xph
&#x4bb;&#xff4f;m&#xff37;&#xff47;r&#x430;&#x440;&#xff48;	h o mW graph
&#xff48;&#x3bf;&#xff4d;&#xff4f;g&#xff52;A&#x440;h	h om o g r Aph
&#x4bb;o&#xff4d;&#xff4f;gr&#xff41;&#xff41;&#x4bb;	hom o gr a a h
&#x4bb;&#x43e;&#x217f;&#x43e;&#xff47;ra&#x399;&#xff48;	homo gral h
&#x4bb;&#xff4f;&#xff4d;&#x43e;gr&#xff41;&#xff4e;h	h o mogr a n h

Table 2: Perfect (100%) homographs of the word "homograph"  
with one character substituted

## APPENDIX B

Encoding	Rendition
h&#x217f;&#xff4f;&#x261;&#xff52;&#x430;&#x440;&#xff48;	hom o g r a p h
h&#x1e43;&#x43e;&#xff47;&#xff52;&#x430;&#xff50;h	hom o g r a p h
h&#xff4f;&#x1e43;&#xff4f;g&#xff52;&#xff41;p&#x4bb;	h o m o g r a p h
&#x4bb;o&#xff4d;o&#xff47;r&#x430;ph	hom o g r a p h
h&#x3bf;&#x1e43;&#x3bf;&#x261;ra&#xff50;&#x4bb;	homogra p h
&#xff48;&#x3bf;m&#x3bf;&#xff47;rap&#x4bb;	h o m o g r a p h
h&#x43e;&#x1e43;&#x3bf;&#x261;&#xff52;&#xff41;&#x440;h	homog r a p h
h&#xff4f;&#x217f;&#x3bf;g&#xff52;&#xff41;p&#x4bb;	h o m o g r a p h
&#x4bb;&#xff4f;&#xff4d;o&#xff47;r&#xff41;&#xff50;&#x4bb;	h o m o g r a p h
h&#x43e;&#x217f;&#xff4f;&#x261;ra&#x440;&#x4bb;	hom o g r a p h

Table 3: Close (95%) homographs for the word "homograph"

Encoding	Rendition
&#xff48;&#x43e;m&#x3bf;&#xff47;&#x2cb;&#x430;&#xff50;&#x4bb;	h o m o g ` a p h
&#xff48;&#xff4f;&#xff4d;&#xff20;gr&#xff41;&#xff50;&#x4bb;	h o m @ g r a p h
&#xff48;&#x3bf;&#x217f;&#x3bf;ur&#x430;&#x440;h	h o m o u r a p h
&#xff48;&#x43e;&#x1e43;&#x43e;&#xff47;&#xff52;&#xff41;&#xff33;h	h o m o g r a S h
&#xff48;&#x3bf;&#xff4d;&#x3bf;&#x261;r&#xff41;&#xff50;q	h o m o g r a p q
hom&#xff57;&#x261;&#xff52;&#xff41;&#x440;h	homw g r a p h
h&#x3bf;&#xff4d;x&#x261;ra&#xff50;&#xff48;	homxgra p h
h&#x43e;&#x3bf;&#xff4f;&#x261;&#xff52;&#xff41;ph	hoo g r a p h
&#xff48;&#x43e;&#x217f;&#x3bf;&#x261;&#xff52;&#xff41;&#x440;;	h o m o g r a p ,
&#xff48;&#xff4f;m&#x3bf;g&#xff52;ap&#x2164;	h o m o g r a p V

Table 4: Close (95%) homographs of the word "homograph" with one character substituted

## APPENDIX C

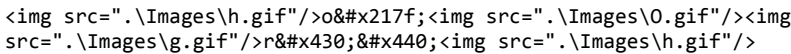
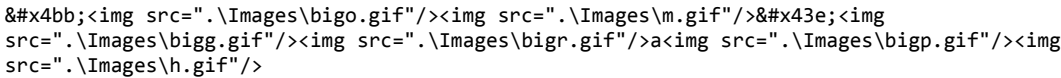
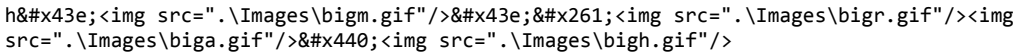
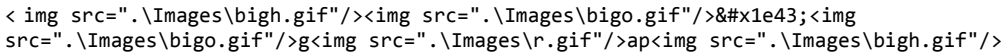
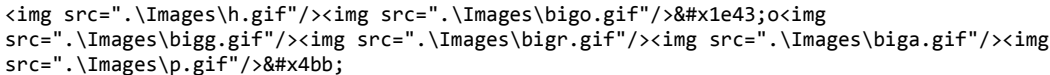
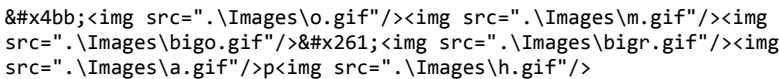
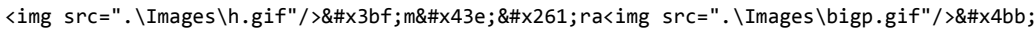
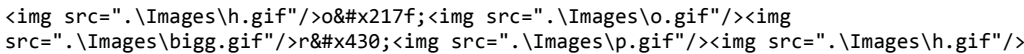
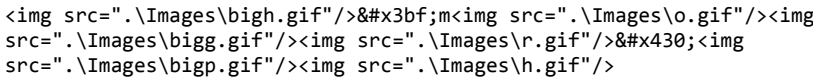
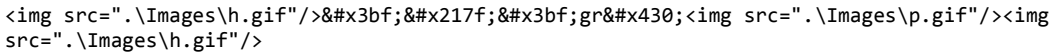
Encoding	Rendition
 homo9raph	
 hōmoGRaPh	
 hoMogR A pH	
 HōmōgraphH	
 hōmōGR A pH	
 homōgR aph	
 homograPh	
 homoGraPh	
 HomoG r aPh	
 homograPh	

Table 5: Close (95%) homographs with HTML-encoded images

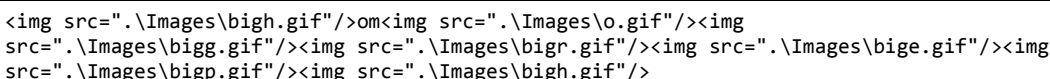
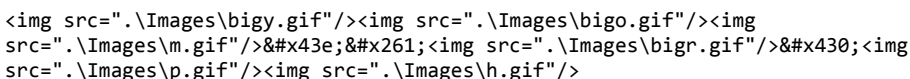
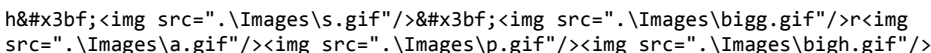
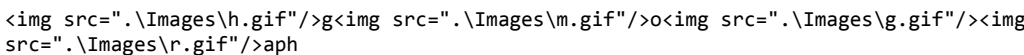
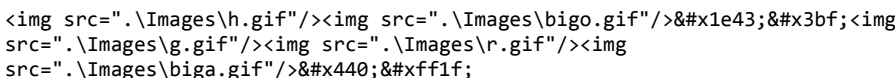
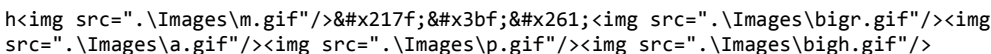
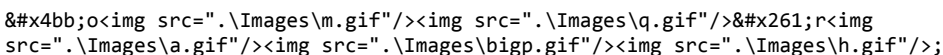
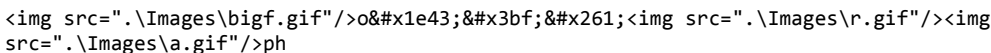
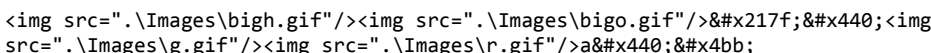
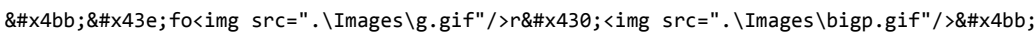
Encoding	Rendition
 HōmōGREPH	
 YōmogR aPh	
 ho <sup>s</sup> oGr aPh	
 hgmo9r aph	
 hōmō9r Ap?	
 hmogR aPh	
 hōm <sup>q</sup> gr aPh	
 fōmōgr aph	
 Hōmp9raph	
 hōfo9raPh	

Table 6: Close (95%) homographs with HTML-encoded images



## REFERENCES

- [1] E. Gabrilovich and A. Gontmakher, "The homograph attack," *Communications of the ACM*, vol. 45, no. 2, p. 128, February 2002.
- [2] A. Fu, W. Zhang, X. Deng and L. Wenxin, "Safeguard against unicode attacks: Generation and application of UC-SimList," in *Proceedings of the 15th International Conference on World Wide Web (WWW '06)*, New York, NY, 2006.
- [3] L. Wenxin, A. Fu and X. Deng, "Exposing homograph obfuscation intentions by coloring unicode strings," *Lecture Notes in Computer Science*, vol. 4976, pp. 275-286, 2008.
- [4] C. Barnes, "Napster fans squeeze through loopholes," 6 March 2001. [Online]. Available: <http://news.cnet.com/2100-1023-253658.html>. [Accessed 10 May 2012].
- [5] A. Fu, X. Deng and L. Wenxin, "A potential IRI based phishing strategy," in *Proc. Sixth Int'l Conf. Web Information Systems Eng. (WISE '05)*, 2005.
- [6] V. Krammer, "Phishing defense against IDN address spoofing attacks," in *Proceedings of the 2006 International Conference on Privacy, Security, Trust: Bridge the Gap Between PST Techniques and Business Services (PST '06)*, 2006.
- [7] Y. Wang, D. Beck, J. Wang, C. Verbowski and B. Daniels, "Strider typo-patrol: Discovery and analysis of systematic typo-squatting," in *Proceedings of the 2nd Conference of Steps to Reducing Unwanted Traffic on the Internet (SRUTI'06)*, 2006.
- [8] P. Faltstrom, Cisco, P. Hoffman, I. &. VPNC, A. Costello and U. Berkeley, "Internationalizing domain names in applications," March 2003. [Online]. Available: <http://www.ietf.org/rfc/rfc3490.txt>. [Accessed 24 April 2012].
- [9] V. Gupta, "International domain names in IE7," 19 December 2005. [Online]. Available: <http://blogs.msdn.com/b/ie/archive/2005/12/19/505564.aspx>. [Accessed 26 April 2012].
- [10] M. Wu, R. Miller and S. Garfinkel, "Do security toolbars actually prevent phishing attacks?," in *Conference on Human Factors in Computing Systems*, 2006.
- [11] K. Yee and K. Sitaker, "Passpet: Convenient password management and phishing protection," in *Symposium On Usable Privacy and Security (SOUPS)*, Pittsburgh, PA, 2006.
- [12] C. Liu and S. Stamm, "Fighting unicode-obfuscated spam," in *APWG eCrime Researchers Summit*, Pittsburgh, PA, 2007.
- [13] A. Fu, X. Deng and L. Wenxin, "REGAP: A tool for unicode-based web identity fraud detection," *Journal of Digital Forensic Practice*, pp. 83-97, 2006.
- [14] A. Fu, L. Wenxin and X. Deng, "EMD based visual similarity for detection of phishing webpages," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 301-311, 2006.
- [15] A. Fu, X. Deng, L. Wenxin and G. Little, "The methodology and an application to fight against unicode attacks," in *Symposium on Usable Privacy and Security (SOUPS)*, Pittsburgh, 2006.
- [16] Oxford University Press, 2009. [Online]. Available: <http://dictionary.oed.com>.
- [17] Random House, Random House Webster's College Dictionary, 1999.
- [18] M. Howard and D. LeBlanc, Writing secure code, Redmond, WA: Microsoft Press, 2002.
- [19] T. Hoad and J. Zobel, "Methods for identifying versioned and plagiarised documents," *J. Am. Soc. In. Sci. Technology*, vol. 54, no. 3, pp. 203-215, February 2003.
- [20] Apple Inc., "The TrueType Font File," 18 December 2002. [Online]. Available: <https://developer.apple.com/fonts/TTRefMan/RM06/Chap6.html>. [Accessed 24 May 2012].
- [21] R. Cockerham, "There are 600,426,974,379,824,381,952 ways to spell Viagra," 2007. [Online]. Available: <http://cockeyed.com/lessons/viagra/viagra.html>. [Accessed 26 April 2012].
- [22] Spammer-X and J. Posluns, Inside the SPAM cartel: Trade secrets from the dark side, Rockland, MA: Syngress Publishing, Inc., 2004.

**James N. Helfrich.** Dr. Helfrich earned his Bachelors and Masters degrees from the University of Pennsylvania in 1994 and 1995 respectively. He completed his PhD from Idaho State University in 2011 with his dissertation exploring the relationship between interactivity and learning performance among university computer science students.

He spent eleven years working for Microsoft Corporation in the Office suite of applications. Though his area of emphasis was user interface design, Dr. Helfrich has worked on most aspects of Word and Publisher. He served as Software Development Engineer and Program Manager during his tenure at Microsoft. Dr. Helfrich currently serves on the faculty of Brigham Young University – Idaho in Rexburg, Idaho. Current areas of research interest include computer security, human computer interaction, and instructional design.



**Rick Neff.** Dr. Neff earned his Masters degrees from the University of Utah in 1986. He completed his PhD in Computer Science from University of Utah in 1995.

He spent several years working in industry before becoming a professor of computer science at Brigham Young University – Idaho. Dr. Neff's areas of research interest include discrete mathematics, algorithms, and computational theory.

