Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Similarity of domain names

Bachelor's Thesis

2024
Lukáš Horňáček

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Similarity of domain names

Bachelor's Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Martin Stanek, PhD.

Bratislava, 2024
Lukáš Horňáček

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Lukáš Horňáček
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Similarity of domain names
Podobnosť doménových mien

Anotácia: Vizuálna, sémantická a iné podobnosti doménových mien sú zneužívané pri tzv. phishingových útokoch, typosquattingu a iných zlomyseľných aktivitách. Bakalárska práca preskúma metriky používané na hodnotenie podobnosti doménových mien. Tieto metriky budú porovnané a ilustrované na vhodnom súbore, spolu s popisom ich silných a slabých vlastností. V druhej časti práce má byť navrhnutý algoritmus na generovanie najpodobnejších doménových mien podľa zvolených kritérií. Implementácia bude porovnaná s analogickými nástrojmi.

Vedúci: doc. RNDr. Martin Stanek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 21.10.2023

Dátum schválenia: 26.10.2023 doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Lukáš Horňáček
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Similarity of domain names

Annotation: Visual, semantic and other similarity of domain names is abused in phishing attacks, typosquatting, and other malicious activities. The thesis will explore metrics used for evaluating similarity of domain names. The metrics are compared and demonstrated on suitable dataset, describing their strong and weak properties. The second part of the thesis should propose and implement an algorithm for generating the most similar domain names, according to chosen criteria. The implementation will be compared with analogous tools.

Supervisor:	doc. RNDr. Martin Stanek, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	21.10.2023

Approved: 26.10.2023 doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: I would like to thank my supervisor doc. RNDr. Martin Stanek, PhD. for his guidance and helpful remarks. I would also like to thank Terézia Kabátová for her advice and for always supporting and encouraging me. This thesis would not exist without her.

Abstrakt

Phishing je útok využívajúci metódy sociálneho inžinierstva, pri ktorom útočník navedie ich obeť, aby navštívila a interagovala so škodlivou webovou stránkou tým, že imituje dôveryhodný podnik, inštitúciu alebo osobu. Útočníci si môžu registrovať domény, ktoré sa podobajú na doménu webovej stránky, ktorú imitujú, aby ich webová stránka vyzerala dôveryhodnejšie. Typosquatting je prax registrovania domén, ktoré sa podobajú na imitovanú doménu, ale obsahujú jednu alebo viac typografických chýb. Keď neskôr typosquatterova obeť spraví chybu pri zadávaní originálneho doménového mena, dostane sa na typosquatterovu webovú stránku, namiesto stránky, ktorú plánoval navštíviť. Táto práca preskúmava využitie podobnosti doménových mien na detekciu phishingových a typosquattingových domén. Popisujeme a porovnávame niekoľko existujúcich funkcií, ktoré merajú podobnosť dvoch domén, a navrhujeme dve nové funkcie. Ďalej navrhujeme a implementujeme nástroj na generovanie domén, ktoré sú podobné zadanej doméne. Taktiež tento nástroj porovnávame s inými podobnými nástrojmi.

Kľúčové slová: phishing, typosquatting, podobnosť, domény

Abstract

Phishing is a social engineering attack, in which the attacker tricks their victim to visit and interact with a malicious website by imitating a legitimate business, institution or person. Attackers may register domains that look similar to domain of the website they are imitating, in order to make their website more convincing. Typosquatting is the practice of registering domains that are similar to the imitated domain but contain one or more typographical errors. Later, when the typosquatter's victim makes an error when typing the original domain name, they will be directed to the typosquatter's website instead of the website they intended to visit. This thesis explores how similarity between domain names can be used to detect phishing and typosquatting domains. We describe and compare various existing functions that measure similarity between two domains, and also propose two novel functions. Afterwards, we propose and implement a tool for generating domains that are similar to a given domain. We compare the tool with other similar tools.

Keywords: phishing, typosquatting, similarity, domains

Contents

List of Figures

List of Tables

Introduction

Phishing is a social engineering attack, in which an attacker attempts to prompt the victim to perform a certain action, such as revealing personal information or downloading malicious software, using a website, in which the attacker imposes as a legitimate business, institution or a reputable person.

For the phishing attack to be successful, the attacker first needs their victim to visit the phishing website. In order to accomplish this goal, the attacker often distributes a link to the website via email or social media. It is then in the attackers interest to make the link look as convincing as possible. For this purpose, the attacker may register domains that look similar to the domain name of the imitated website. For example, if the attacker wants to imitate the domain 'example.com', they might try to register domains such as 'exaample.com', 'exarnple.com', 'example.sk' or 'example.foundhere.com'.

This practice is closely related to another practice called typosquatting. *Typosquatting* is the practice of registering a domain name that is a variant of the imitated domain name, which contains one or more typographical errors. Later, when a user makes such error while typing the URL of the original website, they will be directed to the attacker's website instead. For example, the attacker may register the domain 'exsmple.com', because the letter 's' is adjacent to the letter 'a' on a QWERTY keyboard layout. Afterwards, anytime a user wants to visit 'example.com' but types 's' instead of 'a' by mistake, they will be directed to the typosquatting website instead of the website they intended to visit. Typosquatting domains can be a part of a phishing attack, but they can also be used in other ways. They can also be registered solely for the purpose of selling them to the owner of the original domain for a higher price.

The term phishing first appeared in 1996 [1] and since then, phishing became one of the most common cyberattacks. According to Verizon 2023 Data Breach Investigations Report [2], phishing is one of the three primary ways in which attackers access an organization. Anti-Phishing Working Group observed over one million phishing attacks in the second quarter of 2023 [3]. It is therefore not surprising that the problem of detecting phishing or typosquatting websites has been extensively studied and a variety of approaches have been proposed, see for example a literature survey by Zieni et al. [4].

One of the most widely used approaches is to use blocklists. Blocklists are lists

of URLs, domain names or IP addresses of websites, that had been confirmed to be malicious. An example of a regularly updated blocklist is Google Safe Browsing [5]. Both Google Chrome and Mozilla Firefox make use of Google Safe Browsing in order to display warnings to users when they attempt to visit a known malicious website, see Figure 1.

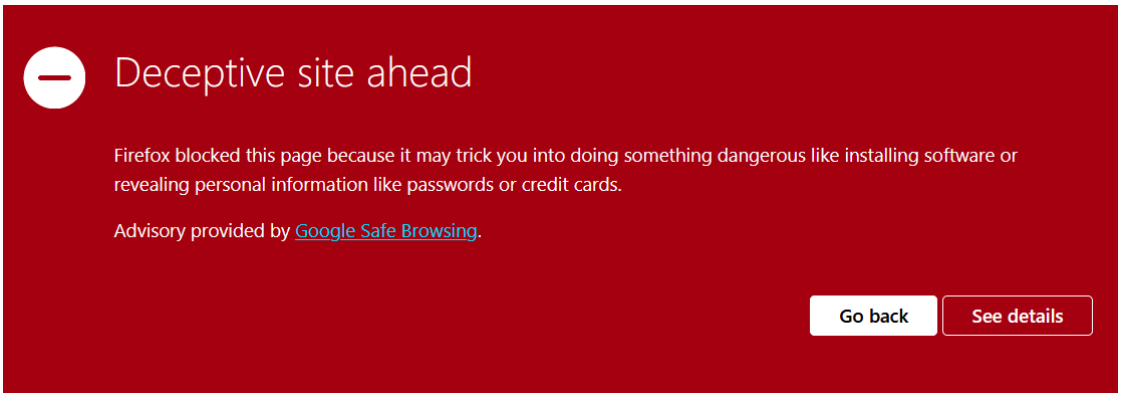

Figure 1: Warning displayed by Mozilla Firefox when user attempts to visit a URL that is present in the blocklist

Blocklists, however, cannot protect users from websites that were not previously seen and added to the list. To mitigate this vulnerability, there have been many heuristic approaches that attempted to detect phishing websites based on various characteristics that are often present in phishing websites, while not being typical for legitimate websites [6, 7].

One of the characteristics of phishing or typosquatting websites is the similarity between their domain and the domain of the legitimate website they are imitating. There have been multiple attempts to use existing functions for quantifying the similarity between two strings, such as the Levenshtein distance or the Jaro-Winkler similarity. Below we describe two different uses for this approach.

When a Google Chrome user attempts to visit a website, Google Chrome first checks whether the URL is in the Google Safe Browsing blocklist, as mentioned above. In addition to that, however, the browser also compares the domain name in the URL to a list of domain names that are either popular or were recently visited by the user [8]. If the domain is similar to any of the compared domains, Google Chrome warns the user that the website might be malicious, see Figure 2.

Moore et al. [9] used the similarity between domain names in a somewhat opposite way. Instead of comparing the domains when a user attempts to visit a website, they constructed a list of 3264 popular domains and for each of them enumerated the most similar domains. More specifically, they enumerated all strings with the Damerau-Levenshtein distance and the fat-finger distance from the domain of at most 2 and discarded any strings that were not a registered domain. The result of this process was

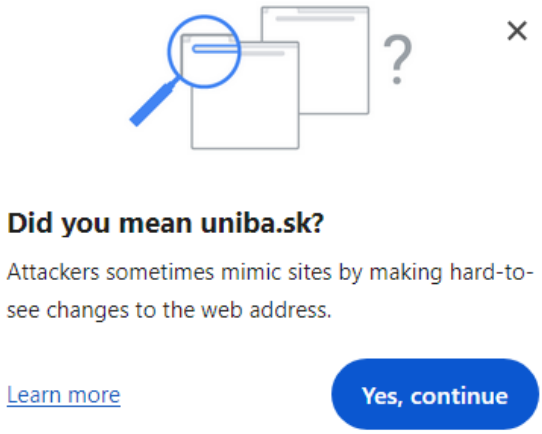

Figure 2: Warning displayed by Google Chrome when user visits a URL that is similar to another popular or previously visited URL

a list of almost 2 million registered domains. They manually checked over 2 thousand of them and found that majority of domains with either distance of 1 were typosquatting domains.

This demonstrates another use case of measuring similarity between domain names. By generating domain names, that are similar to domain names of legitimate websites, it is possible to detect phishing or typosquatting websites even before any user visits them. This process of generating domain names and verifying their maliciousness can be performed by any business or institution that wishes to prevent attackers from imitating their website. The business or institution may take action against any detected malicious website. It can also register some of the most similar domains that are not yet registered, in order to prevent their potential misuse in the future.

In the first part of this thesis we first explore existing functions for measuring similarity between two strings and then propose two novel functions for measuring similarity. We measure their effectiveness in detecting phishing and typosquatting websites, based on the similarity between their domain name and a domain name of another website. Our experiments show that one of the novel functions performs better than any of the existing functions, however, the results are not conclusive.

In the second part of the thesis we propose and implement a tool for generating domain names that are similar to a specified domain name. We also compare it with existing similar solutions and demonstrate its effectiveness. The tool is slower than the existing tools, however, it is capable of generating significantly more similar domains than other existing tools.

Chapter 1

Measuring Similarity

In this chapter we first define approximate string matching and multiple functions for measuring similarity between strings, based on existing literature. Afterwards, we propose two novel functions for measuring similarity between two domain names that are designed to be used specifically for detection of phishing and typosquatting websites. Finally, we compare the effectiveness of the described functions in detection of phishing and typosquatting domain names.

Most of the described existing functions are designed to measure similarity between two arbitrary strings and are not designed with phishing or typosquatting domain names in mind. While we describe only two functions that are designed to detect phishing or typosquatting domains, there are multiple existing functions we chose not to include, such as the patterns used in TypoGard [10], a tool for detecting typosquatting packages.

Throughout this chapter we use variables x and y to represent arbitrary strings. We define $|x|$ as the length of string x. For $i, j \in \{1, \ldots, n\}$ we define x_i as the i-th symbol of x and $x_{i..j}$ as substring $x_i \ldots x_j$. For technical reasons, $x_{i..j}$ is defined as an empty string for $j < i$.

1.1 Approximate String Matching

Approximate string matching is the problem of deciding whether any two given strings are similar to each other according to a specified function [11]. There are two distinct categories of functions used in approximate string matching. *Similarity functions* measure how similar two strings are to each other by giving them a value between 0 and 1, with more similar pairs having greater values. Similarity of any pair of identical strings is 1. *Edit distance functions*, on the other hand, measure how far apart any two strings are, with more similar pairs having smaller values. The distance between identical strings is 0. *Edit distance* between strings x and y is defined as the length of the shortest sequence of edit operations, such that applying the sequence on x transforms it

into y [11]. *Edit operation* is any function $e(a) = b$, where a is a string to be transformed and b is the result of the transformation. This definition can be generalized to allow for different costs for different operations and operation arguments.

1.1.1 Hamming Distance

The first function considered is the Hamming distance, named after its creator Richard Hamming. It is an edit distance function that allows only one type of operation — substitution. The Hamming distance between two strings of equal length is therefore the number of positions at which the strings differ. For example, the Hamming distance between strings 'example.com' and 'exemple.com' is 1 and the distance between 'example.com' and 'example.org' is 3.

Since the Hamming distance has no operation that can change the length of x or y, we define the distance between strings of different length to be ∞. The full definition of the Hamming distance is then as follows.

$$d(x,y) = \begin{cases} \infty, & \text{if } |x| \neq |y| \\ \left|\{i \mid i \in \{1,\ldots,|x|\} \wedge x_i \neq y_i\}\right|, & \text{if } |x| = |y| \end{cases}$$

1.1.2 Longest Common Subsequence Distance

The Longest common subsequence (LCS) of strings x, y is the longest string s, such that s is a subsequence of both x and y. The LCS distance between x and y is then defined as $d(x,y) = |x| + |y| - 2 \cdot |s|$. The distance is equal to the number of characters that need to be removed in order to transform both strings into s. For example, one LCS of strings 'example.com' and 'exemlpe.com' is 'exmpe.com' and the LCS distance between them is 4. There can be multiple common subsequences of same length.

As an edit distance function, the LCS distance allows only insertions and deletions. It is not difficult to prove that insertion of a character into x to match a character in y is equivalent to deletion of said character in y.

We also give an equivalent definition of the LCS distance as a recursive function.

$$d(x_{1..i}, y_{1..j}) = \min \begin{cases} 0, & \text{if } i = j = 0 \\ d(x_{1..i-1}, y_{1..j}) + 1, & \text{if } i > 0 \\ d(x_{1..i}, y_{1..j-1}) + 1, & \text{if } j > 0 \\ d(x_{1..i-1}, y_{1..j-1}), & \text{if } i, j > 0 \wedge x_i = y_j \end{cases}$$

1.1.3 Levenshtein Distance

The Levenshtein distance is an edit distance function that was first proposed by Vladimir I. Levenshtein [12]. It permits three single character operations — substitution, insertion and deletion. This means that the Levenshtein distance between any two strings is always less than or equal to the Hamming distance and to the LCS distance. For example, the Levenshtein distance between 'exaamble.com' and 'example.com' is 2 and the shortest sequence of edit operations consists of deleting the extra character 'a' and substituting 'b' for 'p'. On the other hand, the Hamming distance between the strings is ∞, since it is not possible to delete the extra 'a'. The LCS distance between the strings is 3, since the substitution of character 'b' for 'p' must be performed in two steps — first, the character 'b' is deleted and then the character 'p' is inserted.

Below we give a recursive definition of the Levenshtein distance.

$$d(x_{1..i}, y_{1..j}) = \min \begin{cases} 0, & \text{if } i = j = 0 \\ d(x_{1..i-1}, y_{1..j}) + 1, & \text{if } i > 0 \\ d(x_{1..i}, y_{1..j-1}) + 1, & \text{if } j > 0 \\ d(x_{1..i-1}, y_{1..j-1}), & \text{if } i, j > 0 \wedge x_i = y_j \\ d(x_{1..i-1}, y_{1..j-1}) + 1, & \text{if } i, j > 0 \wedge x_i \neq y_j \end{cases}$$

The Levenshtein distance can be normalized by dividing the result by the length of the longer of the two strings. The normalized Levenshtein distance d_n is then defined as follows.

$$d_n(x, y) = \frac{d(x, y)}{\max\{|x|, |y|\}}$$

The normalized Levenshtein distance of two identical strings is 0, same as with the Levenshtein distance, and the normalized distance of two strings that have nothing in common is 1. Normalizing the distance allows for longer pairs of strings to have the same distance as a shorter pair, even if the distance before normalization was larger.

1.1.4 Damerau-Levenshtein Distance

The Damerau-Levenshtein distance is an edit distance function that was proposed by Fred J. Damerau [13]. It is similar to the Levenshtein distance but supports one additional operation — transposition of two adjacent characters. This means that the Damerau-Levenshtein distance between any two strings is less than or equal to the Levenshtein distance between them. For example, the Damerau-Levenshtein distance between strings 'examlpe.com' and 'example.com' is 1, since the only operation required to transform the first string into the second is the transposition of the characters 'lp'. On the other hand, the Levenshtein distance between these two strings is 2, since both 'l' and 'p' must be changed by substitution.

There are two different variants of the Damerau-Levenshtein distance. In the restricted Damerau-Levenshtein distance, also called Optimal string alignment, if a character or two characters, in case of transposition, are transformed, the resulting character or characters cannot be modified again. The unrestricted Damerau-Levenshtein distance, on the other hand, has no such restriction. This most notably means that when constructing the sequence of edit operations for the unrestricted Damerau-Levenshtein distance, it is possible to insert characters between symbols that were previously transposed, while the restricted variant prohibits it. There are other sequences of operations that are prohibited in the restricted Damerau-Levenshtein distance and permitted in the unrestricted variant, however, these do not change the final distance between any two strings. For example, the unrestricted Damerau-Levenshtein distance between strings 'examlape.com' and 'example.com' is 2, while the restricted Damerau-Levenshtein distance between them is 3.

Below we give a recursive definition for the restricted Damerau-Levenshtein distance.

$$d(x_{1..i}, y_{1..j}) = \min \begin{cases} 0, & \text{if } i = j = 0 \\ d(x_{1..i-1}, y_{1..j}) + 1, & \text{if } i > 0 \\ d(x_{1..i}, y_{1..j-1}) + 1, & \text{if } j > 0 \\ d(x_{1..i-1}, y_{1..j-1}), & \text{if } i, j > 0 \wedge x_i = y_j \\ d(x_{1..i-1}, y_{1..j-1}) + 1, & \text{if } i, j > 0 \wedge x_i \neq y_j \\ d(x_{1..i-2}, y_{1..j-2}) + 1, & \text{if } i, j > 1 \wedge x_i = y_{j-1} \wedge x_{i-1} = y_j \end{cases}$$

The Damerau-Levenshtein distance can be normalized analogically to the Levenshtein distance.

1.1.5 Jaro-Winkler Similarity

The Jaro-Winkler similarity is a similarity function that was proposed by William E. Winkler [14] and is a modification of the Jaro similarity function proposed by Matthew A. Jaro [15].

For $i \in \{1, \ldots, |x|\}$, $j \in \{1, \ldots, |y|\}$, characters x_i, y_j are considered to be *matching characters* if $x_i = y_j$ and $|i - j| \leq \left\lfloor \frac{\max\{|x|,|y|\}}{2} \right\rfloor - 1$.[1] Each character x_i can be matched with only one character y_j and analogically, each character y_j can be matched with only one character x_i. We let m denote the number of matching characters in strings x and y. Next, we consider the number of transpositions, denoted as t. Two pairs of matching characters are considered to be a transposition, if they are not in the same order in both strings. Formally, characters x_i, x_j and their respective matching characters $y_{i'}$,

[1] $\lfloor x \rfloor = \max\{y \in \mathbb{Z} \mid y \leq x\}$

$y_{j'}$ are considered to be a transposition, if $i < j$ and $i' > j'$. With these terms defined we can define the Jaro similarity by the following formula.

$$s_j(x,y) = \begin{cases} 0, & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|x|} + \frac{m}{|y|} + \frac{m-t}{m}\right), & \text{if } m > 0 \end{cases}$$

The Jaro-Winkler similarity modifies the Jaro similarity to give higher values to strings that have a shared prefix. We define l as the length of the shared prefix, capped at 4. Formally, l is the largest number not greater than 4, such that $x_{1..l} = y_{1..l}$. The importance of having a shared prefix can be adjusted by changing a scaling constant p. Higher values of p correspond to greater importance of having a shared prefix. The value, however, cannot exceed 0.25. If p was greater than 0.25, the similarity value could exceed 1. Finally, we give a definition of the Jaro-Winkler similarity.

$$s_w(x,y) = s_j(x,y) + lp(1 - s_j(x,y))$$

For example, the Jaro similarity between strings 'exampel.co.uk' and 'example.com' is roughly 0.86, while the Jaro-Winkler similarity[2] between them is around 0.92, because the differences between the two strings do not occur in the beginning of the strings.

1.1.6 Gestalt Pattern Matching

The Gestalt pattern matching algorithm, also called the Ratcliff-Obershelp algorithm, is a similarity function that was developed by John W. Ratcliff and John A. Obershelp [16]. The similarity value of strings x and y is the number of *matching characters*,[3] denoted as m, multiplied by 2 and divided by $|x| + |y|$.

$$s(x,y) = \frac{2m}{|x| + |y|}$$

The number of matching characters is computed by recursively finding the longest string s, such that s is a substring of both x and y. First, the algorithm finds the longest common substring s and adds $|s|$ to the number of matching characters. Next, the algorithm is recursively called on unmatched substrings to the left and right of s. In case more than one string s exists, or s has more than one occurrence in x or y, it is not specified which should be chosen.

We demonstrate the algorithm by calculating the similarity between strings 'examalpe.com' and 'example.com'. The algorithm first identifies 'e.com' as the longest common substring. Next, the algorithm is recursively called on the remainder of both strings and the longest common substring of these remainders is identified as 'exam'.

[2] With p set to 0.1.

[3] Not related to matching characters defined in the Jaro-Winkler similarity subsection.

Afterwards, the only parts remaining are ‘alp’ and ‘pl’. The longest common substring of these can be either ‘p’ or ‘l’. Either way, the total number of matching characters is 10 and the similarity is calculated as $2 \cdot 10$ divided by $12 + 11$, which is roughly equal to 0.87.

1.1.7 Modified Levenshtein Distance

Liu et al. [17] proposed a modification of the Levenshtein distance for the purpose of detecting typosquatting domains. Their proposal is based on three observations. Firstly, users usually pay more attention to the beginning of the domain, compared to the rest. Secondly, not all substitutions are equally confusing. For example, substituting ‘l’ for ‘1’ is more likely to confuse the user, than for example substituting ‘l’ for ‘m’. Lastly, different operations cause different levels of confusion.

The proposed Modified Levenshtein distance can be defined by the following recursion.

$$d(x_{1..i}, y_{1..j}) = \min \begin{cases} 0, & \text{if } i = j = 0 \\ d(x_{1..i-1}, y_{1..j}) + \alpha^{\min(i,j)}, & \text{if } i > 0 \\ d(x_{1..i}, y_{1..j-1}) + \alpha^{\min(i,j)}, & \text{if } j > 0 \\ d(x_{1..i-1}, y_{1..j-1}), & \text{if } i, j > 0 \wedge x_i = y_j \\ d(x_{1..i-1}, y_{1..j-1}) + \alpha^{\min(i,j)} \cdot \mathcal{M}[x_i, y_j], & \text{if } i, j > 0 \wedge x_i \neq y_j \end{cases}$$

The value α is a constant that is greater than 0 and is less than or equal to 1. The constant represents how important are operations at earlier positions in the strings, compared to later positions. When α is equal to 1, each position has the same importance. Liu et al. experimented with different values of α, but for the purpose of demonstrating the results of their experiments, value 0.99997 was chosen. $\mathcal{M}$ is a symmetric matrix, where rows and columns are indexed by characters, and each value $\mathcal{M}[a, b]$ represents the base cost of substituting character a for character b.

Since all the values in the matrix $\mathcal{M}$ are less than or equal to 1, the Modified Levenshtein distance between any two strings is always less than or equal to the Levenshtein distance between them. For example, the Levenshtein distance between strings ‘exemple.com’ and ‘example.com’ is 1, while the Modified Levenshtein distance between them is around 0.8, because ‘a’ and ‘e’ are considered to be somewhat similar to each other.

1.1.8 Enhanced Levenshtein Distance

Paul E. Black designed an enhanced version of the Damerau-Levenshtein distance, which we refer to as the Enhanced Levenshtein distance [18]. It allows all four op-

erations of the Damerau-Levenshtein distance and modifies the cost of substitution of certain pairs of similar characters, similarly to the Modified Levenshtein distance. The Enhanced Levenshtein distance, however, also modifies the cost of transposition of similar characters.

Additionally, the cost of insertion is reduced if the inserted character is identical to the previous character, and further reduced if the character repeats more than once. The cost of inserting character x_i at the end of string $x_{1..i-1}$ is calculated by the following function.

$$\text{insert}(x_{1..i}, y_{1..j}) = \min \begin{cases} 1, & \text{if } i > 0 \\ 0.9, & \text{if } i > 1 \wedge j > 0 \wedge x_i = x_{i-1} \ \wedge x_{i-1} = y_j \\ 0.5, & \text{if } i > 2 \wedge j > 1 \wedge x_i = x_{i-1} \ \wedge x_{i-2..i-1} = y_{j-1..j} \\ 0.1, & \text{if } i > 3 \wedge j > 2 \wedge x_i = x_{i-1} \ \wedge x_{i-3..i-1} = y_{j-2..j} \\ 0, & \text{if } i > 4 \wedge j > 3 \wedge x_i = x_{i-1} \ \wedge x_{i-4..i-1} = y_{j-3..j} \end{cases}$$

Similarly to the Modified Levenshtein distance, we define a symmetric matrix $\mathcal{S}$, where each value $\mathcal{S}[a, b]$ represents the cost of substitution or transposition of characters a and b. The Enhanced Levenshtein distance is then defined as follows.

$$d(x_{1..i}, y_{1..j}) = \min \begin{cases} 0, & \text{if } i = j = 0 \\ d(x_{1..i-1}, y_{1..j}) + \text{insert}(x_{1..i}, y_{1..j}), & \text{if } i > 0 \\ d(x_{1..i}, y_{1..j-1}) + \text{insert}(y_{1..j}, x_{1..i}), & \text{if } j > 0 \\ d(x_{1..i-1}, y_{1..j-1}), & \text{if } i, j > 0 \wedge x_i = y_j \\ d(x_{1..i-1}, y_{1..j-1}) + \mathcal{S}[x_i, y_j], & \text{if } i, j > 0 \wedge x_i \neq y_j \\ d(x_{1..i-2}, y_{1..j-2}) + \mathcal{S}[x_{i-1}, x_i], & \text{if } i, j > 1 \wedge x_i = y_{j-1} \\ & \qquad \wedge x_{i-1} = y_j \end{cases}$$

For example, the Enhanced Levenshtein distance between ‘example.sk’ and ‘exaample.sk’ is 0.9 and the distance between ‘exaample.sk’ and ‘exaaample.sk’ is 0.5. For another example, the Enhanced Levenshtein distance between strings ‘jigsaw.com’ and ‘ijgsaw.com’ is 0.5, since the transposed characters ‘j’ and ‘i’ are similar to each other, while the distance between ‘jigsaw.com’ and ‘jisgaw.com’ is 1, since the transposed characters are not similar to each other.

1.2 Modified Damerau-Levenshtein Distance

We begin this section by making a few observations about the functions described above, which led us to proposing two novel functions. Afterwards, we describe and define the two functions.

Firstly, we observed that distance functions are generally easier to describe and analyze than similarity functions, since they are defined by sets of simple operations. This also makes them easier to modify, since adding, removing or modifying an operation does not affect the rest of operations. Furthermore, all operations of distance functions described above can be mapped to common typographical mistakes a user can make when typing a domain name. This makes distance functions arguably better suited for detection of typosquatting domains. Even though this is the case, the Enhanced Levenshtein distance is instead designed to detect visually similar phishing domains. The Modified Levenshtein distance is claimed to be designed to detect typosquatting domain, however, the modified costs in matrix $\mathcal{M}$ are in fact the same as costs used by the Enhanced Levenshtein distance. Because of this, we propose the Vanilla distance function, which is designed to detect typosquatting domains and uses different costs of operations.

One limitation of distance functions we identified is that the cost of edit operations is the same regardless of which part of a string they are applied to. In contrast, the Jaro-Winkler similarity is higher if the compared strings share a common prefix, because it may be less likely that users will notice differences further from the beginnings of the strings. The Modified Levenshtein distance achieves a similar effect by decreasing the cost of operations when they are applied further from the beginnings of the compared strings. The second proposed function, called the Caramel distance, expands upon this idea by using the fact that the compared strings are domains, and separating them into multiple parts, which are then compared separately.

1.2.1 Vanilla Distance

The Vanilla distance modifies the restricted Damerau-Levenshtein distance similarly to the Enhanced Levenshtein distance. The key difference is that while the Enhanced Levenshtein distance is designed to detect visually similar phishing domains, the Vanilla distance is designed to detect typosquatting domains.

The Vanilla distance is based on the fat-finger distance proposed by Moore et al. in Measuring the Perpetrators and Funders of Typosquatting [9]. The fat-finger distance between two strings is the minimum number of insertions, deletions, substitutions and transpositions using letters adjacent on a QWERTY keyboard layout. The distance is not defined for pairs of strings, for which it is not possible to transform one of them into the other using only operations with adjacent characters. The Vanilla distance expands on the fat-finger distance by including characters adjacent on a QWERTZ, AZERTY and Dvorak keyboard layout in addition to the QWERTY layout. Additionally, instead of not allowing operations with nonadjacent characters, the Vanilla distance allows them with double the cost of an operation with adjacent characters. This means that

the Vanilla distance is defined for any pair of strings.

Below we give a recursive definition of the Vanilla distance, where A is a set of pairs of adjacent characters. For technical reasons, each character is adjacent with itself.

$$d_V(x_{1..i}, y_{1..j}) = \min \begin{cases} 0, & \text{if } i = j = 0 \\ d(x_{1..i-1}, y_{1..j}) + 1, & \text{if } i > 0 \\ d(x_{1..i}, y_{1..j-1}) + 1, & \text{if } j > 0 \\ d(x_{1..i-1}, y_{1..j}) + 0.5, & \text{if } i > 1 \wedge (x_i, x_{i-1}) \in A \\ d(x_{1..i}, y_{1..j-1}) + 0.5, & \text{if } j > 1 \wedge (y_j, y_{j-1}) \in A \\ d(x_{1..i-1}, y_{1..j-1}) + 1, & \text{if } i, j > 0 \\ d(x_{1..i-1}, y_{1..j-1}) + 0.5, & \text{if } i, j > 0 \wedge (x_i, y_j) \in A \\ d(x_{1..i-1}, y_{1..j-1}), & \text{if } i, j > 0 \wedge x_i = y_j \\ d(x_{1..i-2}, y_{1..j-2}) + 1, & \text{if } i, j > 1 \wedge x_i = y_{j-1} \wedge x_{i-1} = y_j \\ d(x_{1..i-2}, y_{1..j-2}) + 0.5, & \text{if } i, j > 1 \wedge x_i = y_{j-1} \wedge x_{i-1} = y_j \\ & \qquad \wedge (x_i, x_{i-1}) \in A \end{cases}$$

For example, the Vanilla distance between ‘example.com’ and ‘examole.com’ is 0.5, because the first string can be transformed into the second by substituting character ‘p’ for character ‘o’, which is adjacent to ‘p’ on the QWERTY keyboard layout. On the other hand, the Vanilla distance between strings ‘example.com’ and ‘exaxple.com’ is 1, because the substituted character ‘x’ is not adjacent to ‘m’ on any of the supported keyboard layouts.

1.2.2 Caramel Distance

The Caramel distance was designed to capture a specific type of phishing domains that were present in the phishing dataset PhishTank mentioned in the following section. This means that while the Caramel distance might not be best suited to detecting similar domains in general, it might perform better than other functions when detecting a specific type of similar domains.

The Caramel distance modifies the Enhanced Levenshtein distance described in the previous section by splitting domains into multiple parts in an attempt to identify the most important part of each domain, which we call the core. The cores are then compared separately from the rest of the domains, and the distance between the cores has greater weight than the distance between the remaining parts of the domains. For example, the Damerau-Levenshtein distance between domains ‘get-mcafee.tech’ and ‘mcafee.com’ is 8, however, the most important part of both domains is arguably ‘mcafee’ and the distance between the cores of the domains is therefore 0. This means that if we

consider the distance between cores to be more important than the rest of the domains, the domains themselves are similar to each other.

The Caramel distance attempts to identify the cores of the domains using the following method. First, the public suffix of both domains is separated from the rest of the domain. The public suffix, also called effective top-level domain, is a domain suffix under which users can directly register domain names. This includes top-level domains, such as 'com', or domains with more levels, such as 'co.uk'. Domain suffixes, such that the domain owner does not have control over their subdomains, such as 'github.io', are also considered public suffixes. A list of public suffixes initially created by the Mozilla Foundation [19] is publicly available at `https://publicsuffix.org`.

After removing the public suffix, the remainder of the legitimate domain is declared to be the legitimate core. Identifying the core of the phishing domain is, however, more complicated. After removing the public suffix, the remainder of the phishing domain is further split into three disjoint parts — prefix, core and suffix.

The phishing core is chosen from multiple potential cores as the one with the lowest Enhanced Levenshtein distance from the legitimate core. A potential core is any substring c of the phishing remainder x, such that the following condition holds.

$$\begin{aligned}
&\exists i, j \in \mathbb{N}\ x = x_{1..i-1} c x_{j+1..|x|} \wedge \\
&(x_{i-1} = \text{`-'} \vee x_{i-1} = \text{`.'} \vee \forall k \in \{1, \ldots, i-1\}\ x_k \in \{\text{`1'}, \ldots, \text{`9'}, \text{`.'}, \text{`-'}\}) \wedge \\
&(x_{j+1} = \text{`-'} \vee x_{j+1} = \text{`.'} \vee \forall k \in \{j+1, \ldots, |x|\}\ x_k \in \{\text{`1'}, \ldots, \text{`9'}, \text{`.'}, \text{`-'}\})
\end{aligned}$$

The condition ensures that the prefix to any potential core consists of only digits, dots and dashes, or ends with a dot or a dash, while the suffix consists of only digits, dots and dashes, or begins with a dot or a dash. The reason for this is that we found these to be the most common types of prefixes and suffixes of phishing domains present in the PhishTank dataset.

For example, the core of legitimate domain 'example.com' is 'example' and the public suffix is '.com'. The public suffix of phishing domain 'not-example.sk' is '.sk' and the potential cores are 'not', 'example' and 'not-example'. Since 'example' has the lowest distance from the legitimate core, it is chosen as the phishing core, with 'not-' as prefix, empty string as suffix and '.sk' as public suffix.

The value that the prefix and suffix add to the final Caramel distance is determined using a modified length function d_L, which is defined as follows.

$$d_L(x) = |x| - \frac{|\{i \mid i \in \{1, \ldots, |x|\} \wedge x_i \in \{\text{`0'}, \ldots, \text{`9'}, \text{`.'}, \text{`-'}\}\}|}{2}$$

The modified length counts digits, dots and dashes as only half of a character. Finally, the Caramel distance is defined by the following function, where d_E is the Enhanced

Levenshtein distance.

$$d(x,y) = d_E(\text{core}(x), \text{core}(y)) + \frac{d_E(\text{publicsuffix}(x), \text{publicsuffix}(y))}{3} + \frac{d_L(\text{prefix}(y)) + d_L(\text{suffix}(y))}{3}$$

The constants used in the modified length function and the Caramel distance were chosen based on intuition.

To give an example, we demonstrate the Caramel distance on domains 'example.com' and '12-examlpe.com', where '12-examlpe.com' is the phishing domain and 'example.com' is the legitimate domain. First, the legitimate domain is split into two parts — legitimate core 'example' and public suffix '.com'. Next, the phishing domain is split into four parts — phishing core 'examlpe', public suffix '.com', prefix '12-' and empty string as suffix. The Enhanced Levenshtein distance between the cores is equal to 1 and the modified length of the prefix is equal to 1.5. The Caramel distance between the two domains is then equal to 1.5.

1.3 Experiments

In this section we compare the effectiveness of multiple existing functions, namely the Levenshtein distance, Damerau-Levenshtein distance, Jaro-Winkler similarity, Gestalt pattern matching, Modified Levenshtein distance, Enhanced Levenshtein distance and the two novel functions — Vanilla distance and Caramel distance. The Hamming Distance and the LCS distance were excluded, because both of them are edit distance functions whose sets of operations are a strict subset of the set of operations of the Levenshtein distance.

For the Levenshtein distance, Damerau-Levenshtein distance, Jaro similarity and Jaro-Winkler similarity we used implementations available in a Python library named jellyfish [20]. The implementation of the Gestalt pattern matching algorithm was available in a built-in Python library difflib [21]. We implemented the rest of the functions in Python. Implementations of the Vanilla distance and the Caramel distance are included in Appendix A.

We conducted two separate experiments. First, we compared the selected functions using the PhishTank dataset of phishing URLs [22]. We found, however, that this dataset did not contain enough phishing domains that were similar to a legitimate domain. We then compared the functions using a list of domains registered directly under top-level domain 'sk', that is publicly available at `https://sk-nic.sk/subory/domains.txt`. The list is maintained by SK-NIC, the manager of the domain. This experiment showed that most of registered similar domains did not contain malicious websites.

Instead, most of the domains were unavailable, parked or redirected to another unrelated website.

In the rest of this section we describe both experiments in more detail.

1.3.1 PhishTank Dataset

For the first experiment we obtained a dataset of 58107 phishing URLs from PhishTank [22]. PhishTank is a service where registered users can submit and verify suspected phishing URLs they encounter. PhishTank provides access to an hourly updated database of the verified phishing URLs.

We also obtained a dataset of the one million most popular legitimate domains from the Tranco list[4] [23]. Tranco is a service that provides free access to a daily updated ranking list of the most popular websites. The list is created by aggregating data from several other publicly available ranking lists with the purpose of making the list more stable and less prone to malicious manipulation. The Tranco website also contains an archive of previous lists. Both datasets were obtained on April 7, 2024.

To evaluate the effectiveness of the functions we selected the 5000 most popular domains from the legitimate dataset and 5000 unique randomly selected phishing domains extracted from the URLs in the phishing dataset.

For each distance function we chose a threshold and considered each pair of domains (x, y) with distance under that threshold to be similar. The number of true positives for each function and threshold was then computed as the number of phishing domains y, such that there existed a similar legitimate domain x. The number of false positives was computed analogically with pairs of different legitimate domains x and y. The number of true and false positives for similarity functions was computed analogically, with the difference that the domains were considered similar if the similarity value between them was higher than the threshold.

However, the initial results of this experiment showed that the functions performed significantly worse than expected. For example, the Jaro-Winkler similarity with 0.98 as threshold correctly identified only 2 phishing domains, while incorrectly declaring 52 legitimate domains as phishing. While the precision increased with lower thresholds, this was mostly due to the fact, that the majority of legitimate domains were already declared as phishing with higher thresholds. The rest of the functions followed a similar pattern.

One issue that contributed to this result was that the phishing dataset contained long domains that did not look similar to any popular legitimate domain. To mitigate this issue, domains longer than 20 characters were excluded from both datasets. This filter removed around half of domains in the phishing dataset, while removing only 90

[4]The list used in this thesis is available at `https://tranco-list.eu/list/XJJ7N`.

domains from the 5090 most popular legitimate domains.

We repeated the experiment with new lists of 5000 phishing and 5000 legitimate domains that passed through the filter. For comparison, the results with and without the filter for the Jaro-Winkler similarity can be seen in Figure 1.1. For each function the points in the graph represent the number of true and false positives for different thresholds. The thresholds are multiples of 0.01, starting with 1 in the bottom left corner, where both the number of true and false positives is zero, and ending with 0 in the top right corner, where both the number of true and false positives is 5000.

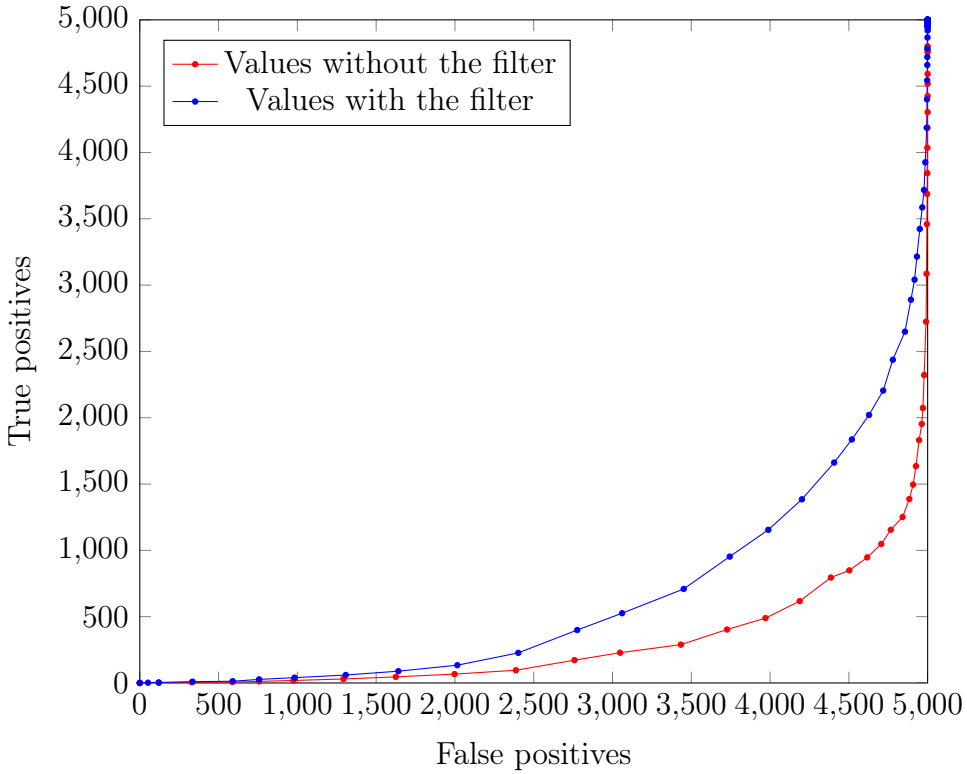

Figure 1.1: Number of true positives in relation with the number of false positives for the Jaro-Winkler similarity

The results of the experiment are shown in Figure 1.2. Again, for each function the points in the graph represent values for different thresholds. The thresholds for similarity functions are multiples of 0.01 between 1 and 0, while the thresholds for distance functions are multiples of 0.1 between 0 and 20. The values for each function and threshold are included in Appendix B. The results showed that all the previously existing functions performed very similarly, however, the Gestalt pattern matching function performed slightly better than the rest. More importantly, the Caramel distance performed better than any other function. This might be due to the fact that it was designed with the phishing dataset in mind. However, the results still suggest that the type of similarity, which the Caramel distance was designed to detect, does not commonly occur in legitimate domains.

Overall, all functions had less than 50% precision, which suggested that legitimate domains on average look more similar to other legitimate domains than to phishing domains contained in the dataset. To verify this, we manually reviewed the phishing dataset and confirmed that the dataset contained very few domains that look similar to legitimate domains, such as typosquatting domains. The dataset, therefore, did not contain phishing domains of types that the selected functions were designed to detect, which explains the poor results. Instead, the PhishTank dataset contained mostly domains that either did not resemble any meaningful word or words, such as '90281721.xyz', or were domains of legitimate services, such as 'docs.google.com' or 'tinyurl.com', where the domain by itself was not malicious. Instead, the legitimate service either unknowingly hosted malicious content, or was used to redirect to another malicious domain.

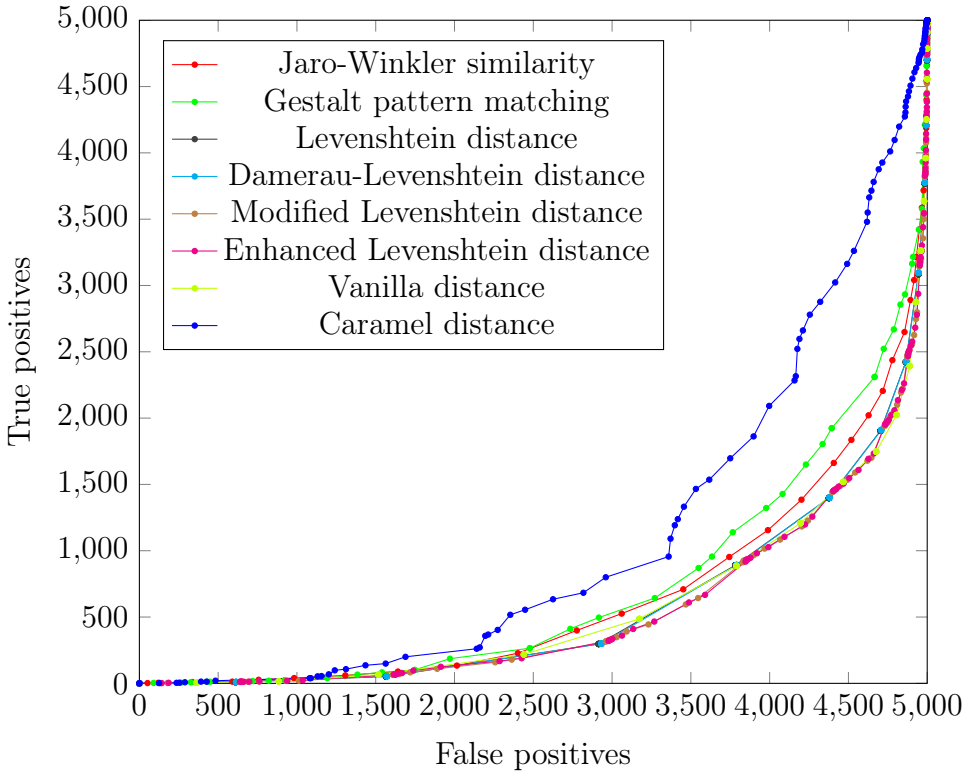

Figure 1.2: Number of true positives in relation with the number of false positives for selected functions

1.3.2 Top-Level Domain sk

Because of the poor results of the experiment with the PhishTank dataset, we decided to conduct another experiment with a publicly available list of domains registered directly under the top-level domain 'sk'. The list was obtained on April 9, 2024.

We selected 5 popular domains registered under the top-level domain 'sk' — 'aktuality.sk', 'bazos.sk', 'dennikn.sk', 'google.sk' and 'zoznam.sk'. For each of the 5 domains we selected the 20 most similar domains registered under 'sk', according to the Jaro-Winkler similarity, Gestalt pattern matching, Damerau-Levenshtein distance, Enhanced Levenshtein distance, Vanilla distance and Caramel distance. For each of the selected domains we manually verified the content of the website accessible on the domain. We separated the websites into 4 categories.

- **Malicious websites** — Websites that either imitate the website their domain is similar to, or contain malicious content. Websites that redirect to other websites containing malicious content are also considered to be malicious.
- **Typosquatting websites** — Typosquatting websites, which do not appear to contain malicious content, but instead redirect to other unrelated websites.
- **Legitimate websites** — Websites that are not related to the target website and do not appear to be intentionally imitating the popular website their domain is similar to.
- **Neutral websites** — Websites that do not fit into any of the other categories, for example because they were unreachable or parked.

The number of websites in each category for each function is listed in Table 1.2. To provide a concrete example, the 20 domains most similar to 'google.sk' according to the Caramel distance, along with their distances and categories, are shown in Table 1.3.

The numbers show that all functions performed very similarly. In fact, there was a significant overlap between the most similar domains according to each function. In total, the functions detected 192 unique domains.[5] For each of the 192 domains we counted the number of functions which detected it, and computed the average number of functions for each category of domains, see Table 1.1. The average numbers show that there was more overlap between malicious and typosquatting domains detected by each function, compared to legitimate and neutral domains.

Type	Number of functions
Malicious	4.2
Typosquatting	4.0
Legitimate	2.0
Neutral	3.1

Table 1.1: Average number of functions for each domain

[5]If there was no overlap between the functions, the total number of unique domains would be 600.

In regard to the previous experiment, each function detected more malicious and typosquatting domains than legitimate domains, which confirms that the poor results of the first experiment can be explained by the fact that the PhishTank dataset did not contain the types of domains this thesis focuses on.

Function	Malicious	Typosquatting	Legitimate	Neutral
Jaro-Winkler similarity	10	16	8	66
Gestalt pattern matching	11	14	7	68
Damerau-Levenshtein distance	14	15	9	62
Enhanced Levenshtein distance	13	14	10	63
Vanilla distance	13	15	6	66
Caramel distance	14	13	8	65

Table 1.2: Number of domains detected by each function

Domain	Caramel distance	Type
googie.sk	0.0	neutral
gooogle.sk	0.5	neutral
coogle.sk	0.8	neutral
gooqle.sk	0.8	neutral
ggoogle.sk	0.9	typosquatting
gogle.sk	0.9	typosquatting
gogole.sk	0.9	malicious
googgle.sk	0.9	malicious
googlee.sk	0.9	malicious
boogie.sk	1.0	legitimate
boogle.sk	1.0	neutral
goodie.sk	1.0	legitimate
googe.sk	1.0	typosquatting
googel.sk	1.0	malicious
googi.sk	1.0	neutral
googke.sk	1.0	malicious
googler.sk	1.0	typosquatting
googlr.sk	1.0	malicious
googlw.sk	1.0	malicious
goole.sk	1.0	malicious

Table 1.3: The domains most similar to ‘google.sk’ according to the Caramel distance

Chapter 2

Similar Domain Generator

In this chapter we propose and implement a tool for generating similar domain names called Similar Domain Generator (SDG). In the simplest scenario, SDG takes a domain name as an input from the user and returns a list of domain names that are similar to the given domain name. SDG also has multiple features designed to make the tool more practical, such as using DNS to return only registered domains.

We first explore existing tools with similar functionality. Next, we identify limitations of these tools and describe the functionality and implementation of SDG, which improves upon them. Finally, we compare SDG with the existing tools.

Throughout this chapter we use the term *modification* to informally refer to any operation that alters a part of a string in a specific way and has a cost associated with it. More formally, modification is any function $e(a) = (b, c)$, where a is a string to be transformed, b is the result of the transformation and c is the cost associated with the transformation. For example, insertion is a modification that inserts any single character at any position in a string with a constant cost of 1. The term modification mirrors the term edit operation defined in the previous chapter.

2.1 Existing Solutions

In this section we explore three of the most notable available tools with functionality similar to SDG. While more tools exist, most of them either offer less functionality than the described tools, such as Domain Check's Typo Generator [24] or Catphish [25], or provide no information about how the similar domains are generated, such as Have I Been Squatted? [26], Domain Name Typo Generator [27] or Cloudflare's Brand Protection [28].

Although we defined modification as an operation with a cost, we found no existing tools which implement operations with variable costs. This is equivalent to each modification having the same constant cost. However, SDG has operations with variable

costs, which allows it to generate domains in order of their similarity.

Typosquatting Finder

Typosquatting Finder [29] is a website and a Python package created by David Cruciani, that generates domains similar to the domain given by the user.

Typosquatting Finder generates similar domains using various types of modifications. This includes broad modifications, such as insertion, deletion, substitution or transposition of any character or characters, and more targeted modifications, such as insertion or deletion of specifically '-' or '.', substitution of similar Unicode characters, or detection of certain words inside the domain, which are then replaced with their misspellings obtained from Wikipedia's list of common misspellings. The tool also uses modifications specific to domain names, such as swapping the top-level domain with another top-level domain, or adding another top-level domain after the original one. The Python package also allows the user to specify a limit to the number of generated domains. It also contains a modification named combo, which generates domains by applying a combination of any two different modifications. The user can select which modifications to use to generate domains.

Typosquatting Finder uses DNS to filter the list of generated domains to contain only registered domains. Additionally, the user can provide a list of NS records and a list of MX records, which will be used to mark matching domains as known. The user can also provide a file containing a list of domains that will be excluded from the result.

URLCrazy

URLCrazy [30] is a command line tool created by Andrew Horton that generates similar domains using mostly the same modifications as the Typosquatting Finder. The most notable difference is that URLCrazy does not generate domains by inserting or substituting any arbitrary character. Instead, these two operations use only characters to the immediate left and right of the original character on keyboard.

Dnstwist

Dnstwist [31] is a command line tool created by Marcin Ulikowski. Although it is also available as a website, the website does not offer all the functionality of the command line tool.

Most of the modifications that dnstwist uses to generate similar domains are also covered by the previous two tools. One notable exception is that the command line version of dnstwist can be supplied with a dictionary file. Dnstwist then uses words from the dictionary to add prefixes and suffixes to the original domain. For example, if the

dictionary includes word ‘fish’ and the given domain is ‘example.com’, dnstwist uses the word ‘fish’ to generate domains ‘fishexample.com’, ‘fish-example.com’, ‘examplefish.com’ and ‘example-fish.com’.

2.2 Functionality

All three described tools offer very similar functionality and use similar modifications to generate domains. SDG contains most of the modifications used in the described tools and combines most of their additional functionality. It allows the user to specify a limit to the number of generated domains. It is capable of filtering domains using DNS and can be supplied with various lists that are used to exclude certain domains from the result. It can also be supplied with a custom word list that is used with certain modifications. We also identified a few ways to improve upon the existing tools.

One limitation of these tools is their limited capability of combining multiple modifications. While Typosquatting Finder allows combining any pair of different modifications, it is not possible to apply more than two modifications to a domain. It is also not possible to apply the same modification twice. Dnstwist and URLCrazy do not allow combining modifications at all. During our experiments in the previous section we found that some phishing domains are created by modifying the original domain multiple times. To address this limitation, SDG is designed to allow repeatedly applying certain compatible modifications without any limit to the number of repetitions.

Another way to improve upon the existing tools is to provide more modifications. This improvement, together with the ability to more freely combine modifications, greatly increases the number of generated domains.

Although increasing the number of generated domains is an improvement, it could also make it infeasible for the user or another automated tool to review the generated list of domains. In fact, depending on configuration of SDG, the length of the list may not be finite. This makes it necessary to limit the number of generated domains. However, if the domains are not generated in any specific order, this means that the limited list might not necessarily contain the most similar domains. To address this issue, SDG implements an additional feature that is not available in any of the existing solutions — a variable cost for each modification. This cost represents how much will the similarity between the original domain and the new domain decrease by applying that modification. SDG uses the variable costs to generate domains in order of their similarity, starting with the most similar domain. This, combined with the limit, allows SDG to generate a list of the most similar domains that is still short enough to be reviewed by the user or a more complex automated tool, which for example analyzes contents of websites accessible on the generated domains.

2.2.1 Modifications

Below we describe how the modifications used by SDG can be combined, and list all modifications, with related modifications grouped together.

The original domain is split into two parts — the public suffix and the rest of the domain, which we refer to as the core. Most modifications can be applied only to the core of the domain. The only exceptions to this are modifications in the public suffix group, which can be applied only to the public suffix.

Any modification can be chained together with any other modification to modify the domain multiple times. For example, the domain 'exaamplee.com' can be generated from 'example.com' by first inserting the extra character 'a', and then inserting the character 'e', or vice versa. The only exceptions to this are modifications in the typosquatting group, which can be chained together with modifications from the same group, but not with modifications from different groups. The reason for this is that the typosquatting modifications generate domains based on the likelihood of them being used as typosquatting domains, while the rest of the modifications generate domains based on their visual similarity to the original domain.

Additionally, once a character in the domain has been modified, it cannot be modified again. For example, if the domain 'example.com' is modified by transposing the first two characters, it is no longer possible to delete the character 'x' from the resulting domain 'xeample.com'. On the other hand, it is still possible to delete a character in another position, for example, to create the domain 'xeamle.com'.

General

The first group of modifications contains four modifications — insertion, deletion, substitution and transposition. A single application of these modifications generates a domain with the Damerau-Levenshtein distance from the original domain of 1. Domains with greater Damerau-Levenshtein distance can be created by chaining these rules.

Typosquatting

The typosquatting group contains two subgroups of modifications — repeat modifications and adjacent modifications. Repeat modifications are repeat insertion and repeat deletion. They reflect a common type of mistake, where a user presses the same keyboard button twice instead of once, or vice versa. Adjacent modifications are adjacent substitution, adjacent insertion, adjacent deletion and adjacent transposition. These modifications represent a similar mistake, where a user accidentally presses a keyboard button that is adjacent to the button they intended to press.

Similar

The similar group contains similar substitution, similar transposition and multicharacter substitution. These modifications represent operations with characters that are visually similar to each other and therefore have a lower cost than the general operations in the general group. For example, the cost of similar substitution of character 'j' for character 'i' is 0.5, while the cost of general substitution for the same pair of characters is 1. The values used for these modifications were obtained from a project of Paul E. Black [18].

Public Suffix

The public suffix group contains modifications that alter the public suffix of a domain. These are TLD[1] insertion, TLD deletion, TLD substitution and TLD transposition. These modifications work analogically to modifications in the general group, however, instead of transforming one or two characters, they transform a single level of a domain. For example, the domain 'example.com' can be transformed into 'examle.com.org' using the TLD insertion. For another example, the domain 'example.co.uk' can be transformed into 'example.uk.co' using the TLD transposition.

Extended Similar

The extended similar group contains several more advanced modifications. These are level transposition, digit substitution, three prefix modifications and three suffix modifications.

The level transposition is similar to TLD transposition, however, it transposes levels that are part of the core of a domain, not part of the public suffix. For example, the domain 'site.example.com' can be transformed into 'example.site.com'.

Digit substitution is a substitution of a digit for its English word representation or vice versa. For example 'exampleone.com' can be transformed into 'example1.com'.

The prefix modifications are user-provided prefix, common prefix and custom prefix. All three modifications take a list of words and insert them to the beginning of the domain. However, they differ in how the list of words is obtained. The user-provided prefix, as the name suggests, uses a list of words that is provided by the user. The common prefix uses a list of the most common subdomains obtained from the SecLists collection of lists.[2] Finally, the custom prefix uses a list of words that are often used on the website accessible on the original domain. The list is obtained by crawling the website with a tool named CeWL (Custom Word List generator) [32]. This list is

[1] top-level domain

[2] The list of subdomains is available at `https://github.com/danielmiessler/SecLists/blob/master/Discovery/DNS/subdomains-top1million-5000.txt`.

filtered using a list of stop words. Stop words are words that hold little or no meaning on their own, such as 'the' or 'is'.

All three modifications insert the word directly before the domain, for example 'worddomain', and separated from it by a dash, for example 'word-domain'.

The three suffix modifications are user-provided suffix, common suffix and custom suffix, and are analogous to the prefix modifications.

2.3 Implementation

Since the implementation makes use of formal grammars, we first need to define a few terms from formal language theory. However, giving a comprehensive introduction to the field of formal language theory is out of scope of this thesis, therefore we provide the definitions without giving further context. For a comprehensive introduction to formal language theory we recommend for example Introduction to Automata Theory, Languages, and Computation [33], which is the source for the following subsection.

2.3.1 Formal Grammars

An *unrestricted grammar* is defined as tuple (N, T, P, S), where N and T are disjoint finite sets of *nonterminal* and *terminal symbols*, S is a nonterminal symbol named *start symbol* and P is a finite set of *productions*, where $P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$. Production (x, y) can also be denoted as $x \to y$. Two or more productions that share the same left-hand side can be grouped together by separating the right-hand sides with character $|$. For example, productions (x, y) and (x, z) can be denoted as $x \to y \mid z$.

A *string* is a finite sequence of nonterminal and terminal symbols. We define a binary relation $\Rightarrow$ (*derives in one step*) between two strings x and y as follows.

$$x \Rightarrow y \Leftrightarrow \exists \alpha, \beta, x_1, x_2 \in (N \cup T)^* \colon x = x_1 \alpha x_2 \wedge y = x_1 \beta x_2 \wedge \alpha \to \beta \in P$$

Relation $\Rightarrow^*$ (*derives*) between strings x and y is defined as the reflexive transitive closure of $\Rightarrow$.

Language generated by grammar G is a set of strings that can be derived from the start symbol S. Formally, language is defined as $L = \{w \mid w \in T^* \wedge S \Rightarrow^* w\}$.

In some grammars, there are multiple possible derivations for a single string. This could mean that there is more than one multiset of productions that can be used to derive the string, or that productions from same multiset can be used in multiple different orders. The *leftmost derivation* is defined as a derivation, such that in each step, the leftmost nonterminal symbol is used in the applied production. The leftmost derivation then serves as a representative derivation for all derivations that use the same multiset of productions.

2.3.2 Weighted Grammar

For our purposes we define *weighted grammar* as an unrestricted grammar that also has a cost associated with each production. This cost represents how much will the distance between the original string and the derived string increase by applying that production. We also allow only nonterminal symbols on the left-hand side of productions.

Formally, weighted grammar is defined as a tuple (N, T, P, S), where N and T are disjoint finite sets of nonterminal and terminal symbols, S is a *start symbol* and P is a finite set of *productions*, where $P \subseteq N^+ \times (N \cup T)^* \times \mathbb{R}_0^+$, and the following condition holds.

$$\forall x, y \in (N \cup T)^* \; \forall a, b \in R\colon (x, y, a) \in P \wedge (x, y, b) \in P \Rightarrow a = b$$

Production (x, y, a) can also be denoted as $x \to y \; [a]$. The third element of a production is the *production cost*.

We define *derivation cost* as the sum of costs of productions used in a derivation. The *cost* of a string is then defined as the minimum derivation cost for all derivations for that string.

For convenience, we use uppercase alphabetic strings to denote nonterminal symbols, and all other strings to denote terminal symbols. For example, *A*, *NINE* and *TLD* are nonterminal symbols, while *a*, *9* and *.com* are terminal symbols. This convention allows us to provide examples of productions without first needing to define the sets of terminal and nonterminal symbols. Figure 2.1 shows a small example of a weighted grammar defined using this notation. We use this example grammar throughout the rest of this section.

$$\begin{aligned}
&S \to A \;\; B \;\; C \; [0.0] \\
&A \to a \; [0.0] \mid C \;\; a \; [1.0] \\
&B \to b \; [0.0] \\
&C \to c \; [0.0] \mid d \; [0.5]
\end{aligned}$$

Figure 2.1: An example of a weighted grammar

2.3.3 Architecture and Design

The most important component of the tool is the SimilarDomainGenerator class. It directly or indirectly utilizes several other components, as can be seen in Figure 2.2. We first describe each of these components separately.[3] Afterwards, we describe how

[3]We do not describe classes, which represent objects from formal language theory, such as terminals and productions.

SimilarDomainGenerator uses the other components to generate similar domains.

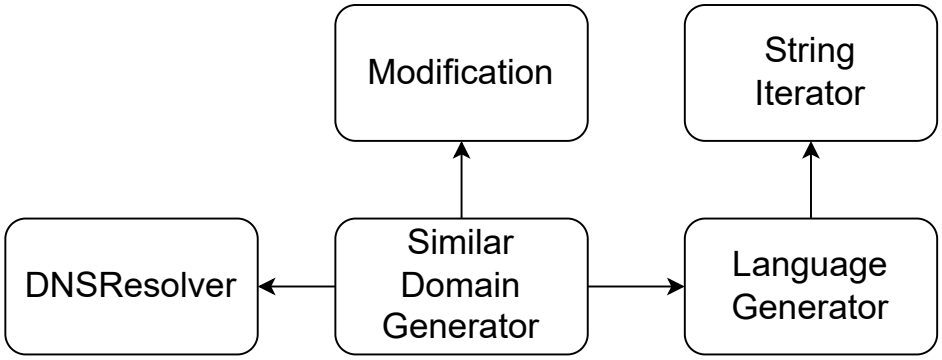

Figure 2.2: A diagram displaying the architecture of SDG

Modification

Instances of this class encapsulate a modification in form of a set of productions. These productions are then used to create a grammar, which generates a language of strings that can be created by applying the modification to a domain.

For example, one modification available in SDG is insertion. The modification is represented by two productions for each pair of characters that are valid in a domain name. For each pair (a, b), there are productions $X \to X\ y\ [c]$ and $X \to y\ X\ [c]$, where X is a nonterminal symbol representing character a, y is a terminal symbol representing character b and c is the cost of the modification.

When the productions are first generated, their cost is calculated to mirror the Caramel distance described in the previous chapter. First, the cost of a production is calculated as the Enhanced Levenshtein distance between the left-hand side and the right-hand side of the production. More precisely, the inputs to the distance function are symbols from the left-hand side concatenated together as the first string and the symbols from the right-hand side concatenated together as the second. Furthermore, each nonterminal symbol is converted to its corresponding terminal symbol. For example, A is converted to a. Nonterminal symbols that do not have a dedicated terminal symbol, such as *PREFIX*, are discarded. For example, in order to calculate the cost of production $PREFIX\ X \to www\text{-}\ X$, it is first converted to $x \to www\text{-}\ x$. Next, the symbols are concatenated together, and the inputs to the distance function are strings 'x' and 'www-x'. The distance between these two strings is 4, therefore the initial cost of the production is also 4.

If the production is applicable to the core of the domain, this is the actual cost of the production. On the other hand, if it is applicable only to the prefix, suffix or public suffix of the domain, the initial cost is further divided by 3. This reflects how the Caramel distance gives greater importance to the core of the domain.

StringIterator

StringIterator is an iterator class that is initialized with a grammar and a string. Afterwards, the StringIterator generates strings on demand. It iterates over all strings that can be derived from the provided string in a single step, using only productions from the provided grammar, which use the leftmost nonterminal in the string. The condition to use only the leftmost nonterminal ensures that all strings, which are later generated by LanguageGenerator, are generated using only leftmost derivations.

The productions are ordered by their cost, starting with the least expensive one. The cost of a generated string is equal to the cost of the applied production added to the cost of the string it was initialized with.

For example, a StringIterator object initialized with the grammar in Figure 2.1 and string $a\ A\ B$ with cost 2 would first generate string $a\ a\ B$ with cost 2. The next string would be $a\ C\ a\ B$ with cost 3, and afterwards there would be no more strings to be generated.

LanguageGenerator

LanguageGenerator is an iterator class that is initialized with a grammar. The iterator generates strings from language generated by the grammar. The strings are generated in order of their cost, starting with the string with the lowest cost.

During iteration, LanguageGenerator uses a priority queue of StringIterator objects, where the priority of a StringIterator is the cost of the string it will return next, with the StringIterator the lowest cost having the highest priority. In the beginning, the queue contains a single StringIterator initialized with string, that consists only of the start symbol and has the cost of 0. Each time a new string from the language needs to be generated, the following function is called.

```
def next():
    # queue: priority queue of StringIterators
    while not queue.empty():
        it = queue.get()
        string = it.next()
        if it.has_next():
            queue.put(it)
        if string.contains_nonterminal():
            new_it = StringIterator(string)
            queue.put(new_it)
        else if string not in previously_generated:
            previously_generated.add(string)
            return string
```

```
    # there are no more strings to be generated
    raise Exception
```

Since each StringIterator generates strings ordered by their distance and the priority queue orders the iterators in such a way, that the iterator, which will generate the string with the lowest distance will be used each time, the algorithm ensures that LanguageGenerator generates strings ordered by their distance.

For example, LanguageGenerator initialized with the grammar in Figure 2.1 with S as the start symbol would first generate $a\ b\ c$, then $a\ b\ d$, $c\ a\ b\ c$, $c\ a\ b\ d$ and $d\ a\ b\ c$. The last generated string would be $d\ a\ b\ d$.

DNSResolver

The primary purpose of this class is to test whether a domain is registered using DNS queries. It is initialized with a list of IP addresses and networks, a list of NS records and a list of MX records, based on which to exclude domains.

DNSResolver has a single function named `resolves`. The function first looks up the SOA record for the domain to check, if it is registered. Afterwards, if the domain is registered, it looks up A, AAAA, MX and NS records for the domain and checks if any of the records are present in the corresponding lists. If the domain is registered and no record is present in its corresponding list, the function returns True. Otherwise, it returns False.

SimilarDomainGenerator

The main class of the package. It is initialized with the following parameters provided by the user. All except the first are optional.

- **Target domain** — The only required parameter. SDG generates domains similar to this domain.
- **Limit** — The maximum number of generated similar domains.
- **Selected modifications** — By default, SDG generates domain names using every modification available. Optionally, the user can specify a list of modifications to be used. For convenience, the user can also select whole groups of modifications instead of selecting them individually.
- **Word list** — A list of words used by the user-provided prefix and suffix modifications. These should be words associated with the individual, business or institution that owns the target domain.

- **Excluded domains** — A list of domains to exclude when generating domains. These can be domains that the user owns, or domains that the user knows about and does not want them to clutter the output.
- **Excluded IP addresses and networks** — A list of IP addresses and IP networks, based on which to exclude domains.
- **Excluded NS records** — A list of NS records, based on which to exclude domains.
- **Excluded MX records** — A list of MX records, based on which to exclude domains.
- **Registered only** — A boolean value that determines if SDG should return only registered domains. If this parameter is set to False, SDG will not make any DNS requests and the lists of excluded IP addresses and networks, NS records and MX records will not be used.

The selected modifications are used to create a grammar, which is in turn used to construct a LanguageGenerator object. The lists of IP addresses and networks, NS and MX records are used to construct a DNSResolver object. After the initialization is completed, the list of similar domains can be obtained by calling the following function.

```
def generate():
    domains = []
    while len(domains) < limit:
        if not language_generator.has_next():
            break
        domain = language_generator.next()
        if domain in excluded_domains:
            continue
        if not dns_resolver.resolves(domain):
            continue
        domains.append(domain)
    return domains
```

2.4 Results

We compared SDG and the three previously described tools based on three properties — the time it takes to generate the list of similar domains,[4] how many of the generated

[4]The time was measured on an AMD Ryzen 5 5600H processor with 3.30 GHz clock speed.

domains were registered, and the average Damerau-Levenshtein and Caramel distance of the generated domains. The time was measured without taking into account the time it takes to check if the domains are registered. Since dnstwist does not allow the user to disable this functionality, we had to disable it directly in source code.

The number of registered domains is an indicator of how closely the generated domains reflect domains used on the Internet. If a tool generated no registered domains, not only would it mean that the tool failed to detect any phishing or typosquatting domains, it would also indicate that the types of domains generated by the tool are not being used, and it might not be worth it to register them as a preventative measure.

The properties were measured using the same 5 popular domains registered under the top-level domain 'sk' that were used in the previous chapter. The resulting values were computed as averages of values measured for each of the domains.

Existing Tools

The properties of the existing tools were measured with the following configurations. Typosquatting Finder and URLCrazy were used with their default configurations, which means that all their modifications were used. Since dnstwist can be provided with a list of top-level domains and a custom list of words, it was used with two different configurations — one with all modifications that do not require the two lists and one with all modifications, including the modifications that require the lists.[5] The results of the measurements are shown in Tables 2.1, 2.2 and 2.3, along with selected values from the measurements of SDG described below.

The results show that Typosquatting Finder generates domains in the shortest time, generates the most registered domains and the generated domains have the lowest average Vanilla and Caramel distance. On the other hand, while domains generated by URLCrazy have higher average distance and there are less registered domains in total, the percentage of generated domains that are registered is much higher. For Typosquatting Finder, around 1.10% of generated domains were registered, while around 4.73% of domains generated by URLCrazy were registered.

Similar Domain Generator

The properties of SDG were measured with 5 different configurations:

- **General**, which used modifications from the general group.
- **Typosquatting**, which used modifications from the typosquatting group.
- **Similar**, which used modifications from the similar group.

[5]The provided lists were the same as the lists used by SDG.

Tool	Time
dnstwist (default)	0.45s
dnstwist (with lists)	1.14s
Typosquatting Finder	0.28s
URLCrazy	0.35s
SDG Default (5000)	7.98s
SDG General (5000)	2.67s
SDG Default (10000)	9.10s
SDG General (10000)	4.45s
SDG Default (25000)	23.23s
SDG General (25000)	9.53s

Table 2.1: Average time needed to generate domains by existing tools and SDG

Tool	Generated domains	Registered domains
dnstwist (default)	3256.6	17.8
dnstwist (with lists)	20619.2	31.2
Typosquatting Finder	10600.0	117.0
URLCrazy	2030.6	96.0
SDG Default (5000)	5000	26.0
SDG General (5000)	5000	38.0
SDG Default (10000)	10000	65.2
SDG General (10000)	10000	70.4
SDG Default (25000)	25000	271.0
SDG General (25000)	25000	115.6

Table 2.2: Average number of registered domains generated by existing tools and SDG

- **Extended similar**, which used modifications from both the similar and the extended similar group.
- **Default**, which used all modifications.

The results of the measurements are shown in Figures 2.3, 2.4 and 2.5. All configurations, except the similar configuration were capable of generating at least 100000 domains. The modifications in the similar configuration were capable of changing only few selected characters, which limited the highest possible number of generated domains.

The results show that the default configuration generated the most registered domains, followed by the general configuration. The default configuration was also the slowest configuration, while the general was the second slowest. The rest of the

Tool	Damerau-Levenshtein	Caramel
dnstwist (default)	2.27	2.15
dnstwist (with lists)	5.01	3.08
Typosquatting Finder	2.39	1.01
URLCrazy	4.82	1.63
SDG Default (5000)	3.30	2.12
SDG General (5000)	1.89	1.76
SDG Default (10000)	3.48	2.20
SDG General (10000)	1.95	1.86
SDG Default (25000)	3.68	2.22
SDG General (25000)	1.98	1.86

Table 2.3: Average distance of domains generated by existing tools and SDG

configurations generated significantly less registered domains.

In comparison with the existing tools, SDG with both the default and the general configuration takes longer than Typosquatting Finder or URLCrazy to generate the same number of domains, however, it is capable of generating a significantly larger number of both registered and unregistered domains. At 100000 generated domains, the percentage of registered domains with the default configuration is around 1.78%, which is higher than the percentage of registered domains generated by Typosquatting Finder. With the general configuration the percentage is only around 0.56%.

We conclude that SDG is not better suited than the existing tools for generating a small list of unregistered domains for preventative registration, or for generating a small list of potential phishing and typosquatting domains for manual verification by the user. However, since SDG is capable of generating significantly more domains, it is better suited for a more extensive detection of phishing and typosquatting domains, where the generated list is further filtered using another automated tool, which for example analyzes contents of websites accessible on the generated domains, such as Cantina [6].

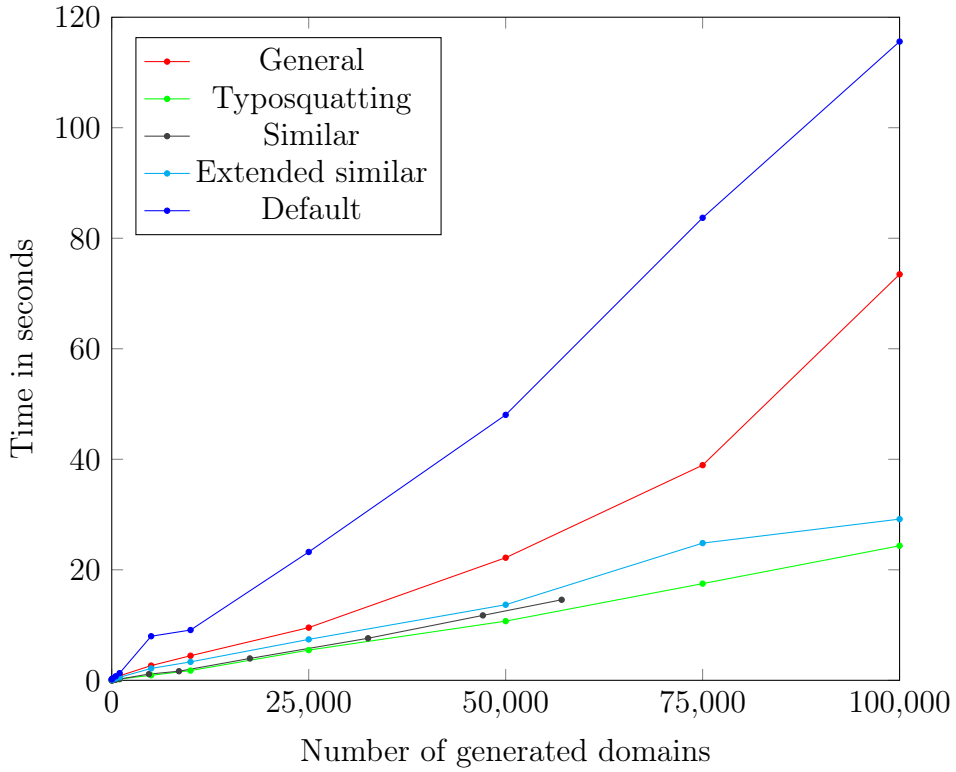

Figure 2.3: Average time needed to generate domains by SDG

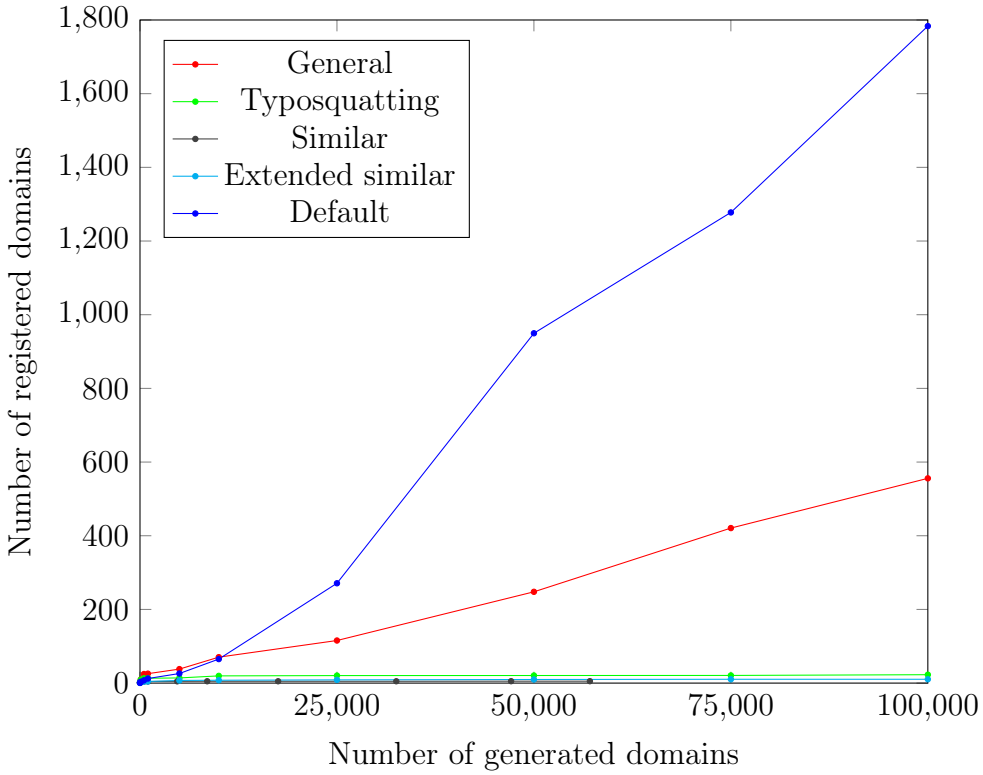

Figure 2.4: Average number of registered domains generated by SDG

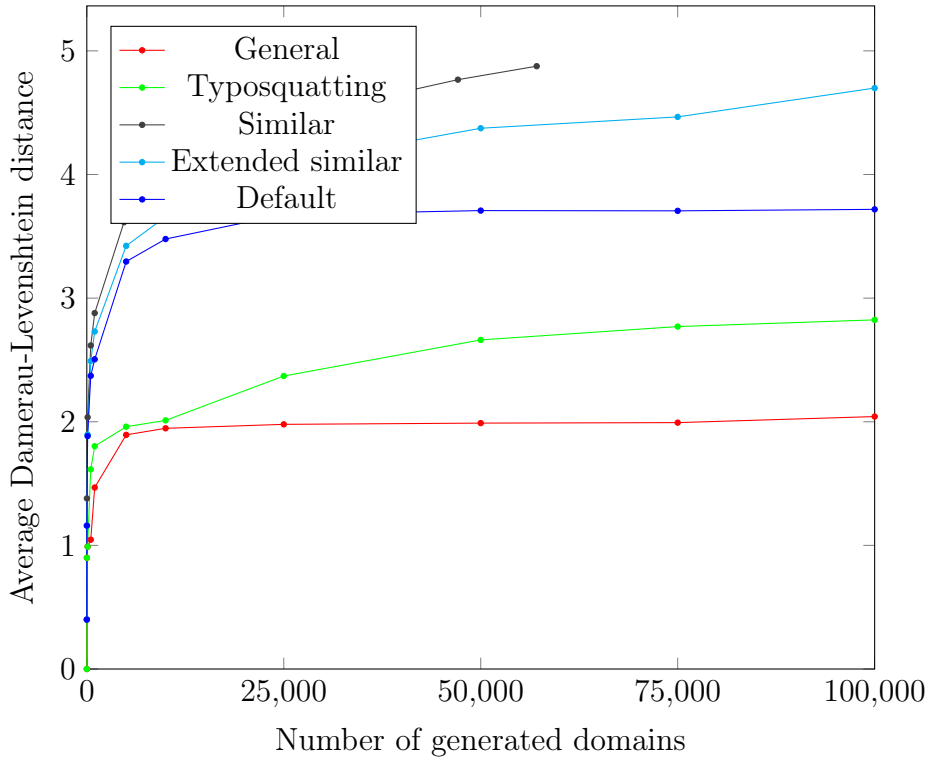

Figure 2.5: Average Damerau-Levenshtein distance of domains generated by SDG

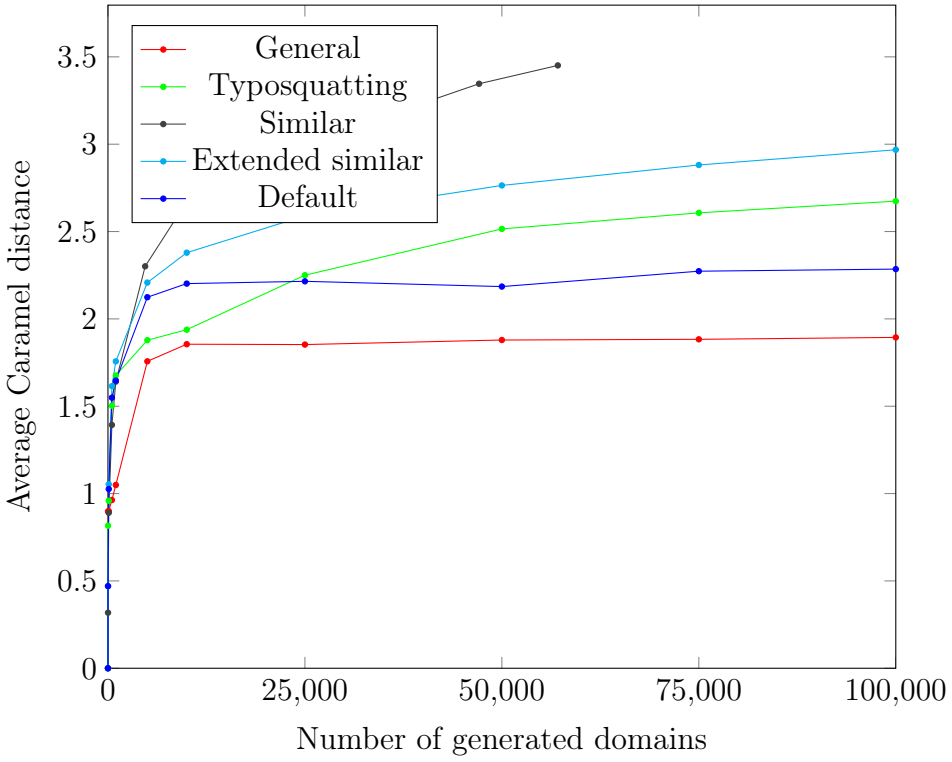

Figure 2.6: Average Caramel distance of domains generated by SDG

Conclusion

This thesis explored the usage of similarity between domain names for the purpose of detecting phishing and typosquatting websites. We described several existing functions that measure similarity between two domains, and proposed two novel functions — the Vanilla distance and the Caramel distance. The Vanilla distance was designed to detect typosquatting domains, while the Caramel distance was designed to detect a specific type of visually similar phishing domains. Both of them were based on already existing functions.

We compared the functions using a publicly available dataset of phishing domains and a list of the most popular legitimate domains. The experiment, however, did not yield conclusive results, due to the fact, that the phishing dataset did not contain many domains that appeared similar to a legitimate domain. We conducted another experiment using domains registered under the top-level domain 'sk', which confirmed that the results of the first experiment were most likely not representative of the real world situation.

In the second part of the thesis we proposed and implemented Similar Domain Generator, a tool for generating domains that are similar to a given domain. SDG can be used for detection of phishing and typosquatting domains, or preventative registration of the most similar domains, so they cannot be misused in the future. SDG uses various modifications, such as insertion or substitution, in order to transform the original domain and generate new similar domains. Each of the modifications has a variable cost associated with it. The cost represents how much will the similarity between the original and the generated domain decrease by applying the modification.

We compared SDG with other existing tools and concluded, that although SDG is slower than the other tools, it is capable of generating significantly more of both unregistered and registered domains. This makes it suitable for a more extensive detection of phishing and typosquatting domains, where the generated domains are further analyzed by another automated tool, which for example analyzes contents of websites accessible on the domains generated by SDG.

Further work on this subject would include obtaining a dataset of phishing and typosquatting domains that appear similar to legitimate domains they are targeting. This dataset could then be used to repeat the experiments in Section 1.3, in order to

obtain more reliable results.

In regard to SDG, further work would include adding more modifications. It could also be beneficial to more extensively experiment with various configurations and change the cost of modifications in order to generate more relevant domains first.

Bibliography

[1] Paul Gillin. The History of Phishing. Retrieved December 16, 2023, from https://www.verizon.com/business/resources/articles/s/the-history-of-phishing/.

[2] Verizon 2023 Data Breach Investigations Report, 2023. Retrieved December 13, 2023, from https://www.verizon.com/business/resources/reports/dbir/.

[3] Anti-Phishing Working Group. Phishing Attacks Trends Report, 2nd quarter 2023, 2023. Retrieved December 13, 2023, from https://docs.apwg.org/reports/apwg_trends_report_q2_2023.pdf.

[4] Rasha Zieni, Luisa Massari, and Maria Carla Calzarossa. Phishing or Not Phishing? A Survey on the Detection of Phishing Websites. *IEEE Access*, 11:18499–18519, 2023.

[5] Google Safe Browsing. Retrieved December 16, 2023, from https://safebrowsing.google.com.

[6] Yue Zhang, Jason I Hong, and Lorrie F Cranor. Cantina: A Content-Based Approach to Detecting Phishing Websites. In *Proceedings of the 16th international conference on World Wide Web*, pages 639–648, 2007.

[7] Mouad Zouina and Benaceur Outtaj. A Novel Lightweight URL Phishing Detection System Using SVM and Similarity Index. *Human-centric Computing and Information Sciences*, 7(1):1–13, 2017.

[8] Lookalike Warnings in Google Chrome. Retrieved December 16, 2023, from https://chromium.googlesource.com/chromium/src/+/master/docs/security/lookalikes/lookalike-domains.md.

[9] Tyler Moore and Benjamin Edelman. Measuring the Perpetrators and Funders of Typosquatting. In *International Conference on Financial Cryptography and Data Security*, pages 175–191. Springer, 2010.

[10] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Defending Against Package Typosquatting. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14*, pages 112–131. Springer, 2020.

[11] Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[12] Vladimir I Levenshtein et al. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[13] Fred J Damerau. A Technique for Computer Detection and Correction of Spelling Errors. *Communications of the ACM*, 7(3):171–176, 1964.

[14] William E Winkler. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. 1990.

[15] Matthew A Jaro. Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[16] John W Ratcliff, David Metzener, et al. Pattern Matching: The Gestalt Approach. *Dr. Dobb's Journal*, 13(7):46, 1988.

[17] Tingwen Liu, Yang Zhang, Jinqiao Shi, Ya Jing, Quangang Li, and Li Guo. Towards Quantifying Visual Similarity of Domain Names for Combating Typosquatting Abuse. In *MILCOM 2016-2016 IEEE Military Communications Conference*, pages 770–775. IEEE, 2016.

[18] Paul E Black. Compute Visual Similarity of Top-Level Domains. Retrieved Februrary 12, 2024, from `https://hissa.nist.gov/~black/GTLD/`.

[19] Public Suffix List. Retrieved February 12, 2024, from `https://publicsuffix.org/`.

[20] James Turk. jellyfish, June 2023. Retrieved February 12, 2024, from `https://github.com/jamesturk/jellyfish`.

[21] difflib. Retrieved February 12, 2024, from `https://docs.python.org/3/library/difflib.html`.

[22] PhishTank. Retrieved February 12, 2024, from `https://phishtank.org/`.

[23] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[24] Typo Generator. Retrieved February 27, 2024, from `https://domaincheckplugin.com/typo`.

[25] Catphish. Retrieved February 27, 2024, from `https://github.com/ring0lab/catphish`.

[26] Have I Been Squatted? Retrieved February 27, 2024, from `https://www.haveibeensquatted.com/`.

[27] Domain Name Typo Generator. Retrieved February 27, 2024, from `https://www.internetmarketingninjas.com/tools/domain-typo-generator/`.

[28] Brand Protection. Retrieved February 27, 2024, from `https://developers.cloudflare.com/security-center/brand-protection/`.

[29] David Cruciani. Domain Name Typo Generator. Retrieved February 27, 2024, from `https://www.internetmarketingninjas.com/tools/domain-typo-generator/`.

[30] Andrew Horton. URLCrazy. Retrieved February 27, 2024, from `https://morningstarsecurity.com/research/urlcrazy`.

[31] Ulikowski Marcin. dnstwist. Retrieved February 27, 2024, from `https://dnstwist.it/`.

[32] Robin Wood. CeWL. Retrieved April 12, 2024, from `https://github.com/digininja/CeWL`.

[33] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to Automata Theory, Languages, and Computation. *Acm Sigact News*, 32(1):60–65, 2001.

Appendix A

Source Code

The electronic attachment contains source code files for Python implementations of the Vanilla and Caramel distances, as well as the implementation of Similar Domain Generator. Instructions for use of Similar Domain Generator are included in file `README.md`.

Appendix B

Experiment Results

The results of the experiment with the PhishTank dataset are included in the form of CSV files as an electronic attachment.